

Parallel pointer analysis for large-scale software

Author:

Su, Yu

Publication Date:

2015

DOI:

<https://doi.org/10.26190/unsworks/18164>

License:

<https://creativecommons.org/licenses/by-nc-nd/3.0/au/>

Link to license to see what you are allowed to do with this resource.

Downloaded from <http://hdl.handle.net/1959.4/54374> in <https://unsworks.unsw.edu.au> on 2024-05-03

Parallel Pointer Analysis for Large-Scale Software

by

Yu Su

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
IN THE SCHOOL
OF
COMPUTER SCIENCE AND ENGINEERING
THE UNIVERSITY OF
NEW SOUTH WALES



SYDNEY • AUSTRALIA

April 29, 2015

All rights reserved. This work may not be
reproduced in whole or in part, by photocopy
or other means, without the permission of the author.

© Yu Su 2015

PLEASE TYPE

THE UNIVERSITY OF NEW SOUTH WALES
Thesis/Dissertation Sheet

Surname or Family name: **Su**

First name: **Yu**

Other name/s:

Abbreviation for degree as given in the University
calendar: **PhD**

School: **School of Computer Science and
Engineering**

Faculty: **Faculty of Engineering**

Title: **Parallel Pointer Analysis for Large-Scale
Software**

Abstract 350 words maximum: (PLEASE TYPE)

Pointer analysis, a process that statically computes the possible runtime values of a pointer in a program, enables the understanding of program behaviours. It lies in the heart of software engineering and has laid foundations for extensive applications, such as compiler optimisation, software bug detection and program verification. The long existing challenge of the analysis, however, is to improve its efficiency while maintaining high precision, especially when applied to large programs.

Parallel platforms, which are prevalent nowadays, provide a great opportunity to enhance the efficiency of pointer analysis. Yet, it is challenging to parallelise this analysis, which is essentially an irregular graph algorithm. In general, pointer analysis comes in two styles: whole-program and demand-driven. Whole-program analysis, which computes the points-to information of all variables in a program, is often formulated as a graph-rewriting problem that makes extensive modifications to data structures representing the graph. Demand-driven analysis, which only targets the variables requested by queries, is solved in terms of a graph traversal problem.

This thesis presents the design and implementation of a parallel pointer analysis framework that enables efficient pointer analysis for large-scale software. This framework consists of three parts, each targeting one of today's most popular parallel platforms, and is implemented with a combination of Java, C++ and CUDA. The first part is a parallel solution to pointer analysis driven by queries, on multicore CPUs. It has achieved significant speedups over the sequential solution, since a large amount of unnecessary graph traversals have been eliminated by information sharing and query scheduling. The second part is an efficient GPU solution to whole-program pointer analysis. With effective load balancing and reduced redundant computation, it demonstrates considerable speedups over the state-of-the-art GPU implementation. The third part is a heterogeneous CPU-GPU solution to whole-program pointer analysis. It prioritises the distribution of different workloads to CPU/GPU according to the processing unit's ability for processing them, and therefore has achieved speedups over the corresponding CPU-only and GPU-only solutions. The effectiveness of each part of the framework is demonstrated via an evaluation with a set of open-source Java/C programs.

Declaration relating to disposition of project thesis/dissertation

I hereby grant to the University of New South Wales or its agents the right to archive and to make available my thesis or dissertation in whole or in part in the University libraries in all forms of media, now or here after known, subject to the provisions of the Copyright Act 1968. I retain all property rights, such as patent rights. I also retain the right to use in future works (such as articles or books) all or part of this thesis or dissertation.

I also authorise University Microfilms to use the 350 word abstract of my thesis in Dissertation Abstracts International (this is applicable to doctoral theses only).



Signature



Witness

29/04/2015

Date

The University recognises that there may be exceptional circumstances requiring restrictions on copying or conditions on use. Requests for restriction for a period of up to 2 years must be made in writing. Requests for a longer period of restriction may be considered in exceptional circumstances and require the approval of the Dean of Graduate Research.

FOR OFFICE USE ONLY

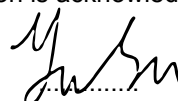
Date of completion of requirements for Award:

THIS SHEET IS TO BE GLUED TO THE INSIDE FRONT COVER OF THE THESIS

ORIGINALITY STATEMENT

'I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, or substantial proportions of material which have been accepted for the award of any other degree or diploma at UNSW or any other educational institution, except where due acknowledgement is made in the thesis. Any contribution made to the research by others, with whom I have worked at UNSW or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project's design and conception or in style, presentation and linguistic expression is acknowledged.'

Signed

A handwritten signature in black ink, appearing to be 'J. Smith', written over a dotted line.

Date

29/04/2015
.....

COPYRIGHT STATEMENT

'I hereby grant the University of New South Wales or its agents the right to archive and to make available my thesis or dissertation in whole or part in the University libraries in all forms of media, now or here after known, subject to the provisions of the Copyright Act 1968. I retain all proprietary rights, such as patent rights. I also retain the right to use in future works (such as articles or books) all or part of this thesis or dissertation.

I also authorise University Microfilms to use the 350 word abstract of my thesis in Dissertation Abstract International (this is applicable to doctoral theses only).

I have either used no substantial portions of copyright material in my thesis or I have obtained permission to use copyright material; where permission has not been granted I have applied/will apply for a partial restriction of the digital copy of my thesis or dissertation.'

Signed



Date

29/04/2015

AUTHENTICITY STATEMENT

'I certify that the Library deposit digital copy is a direct equivalent of the final officially approved version of my thesis. No emendation of content has occurred and if there are any minor variations in formatting, they are the result of the conversion to digital format.'

Signed



Date

29/04/2015

Abstract

Pointer analysis, a process that statically computes the possible runtime values of a pointer in a program, enables the understanding of program behaviours. It lies in the heart of software engineering and has laid foundations for extensive applications, such as compiler optimisation, software bug detection and program verification. The long existing challenge of the analysis, however, is to improve its efficiency while maintaining high precision, especially when applied to large programs.

Parallel platforms, which are prevalent nowadays, provide a great opportunity to enhance the efficiency of pointer analysis. Yet, it is challenging to parallelise this analysis, which is essentially an irregular graph algorithm. In general, pointer analysis comes in two styles: whole-program and demand-driven. Whole-program analysis, which computes the points-to information of all variables in a program, is often formulated as a graph-rewriting problem that makes extensive modifications to data structures representing the graph. Demand-driven analysis, which only targets the variables requested by queries, is solved in terms of a graph traversal problem.

This thesis presents the design and implementation of a parallel pointer analysis framework that enables efficient pointer analysis for large-scale software. This framework consists of three parts, each targeting one of today's most popular par-

allel platforms, and is implemented with a combination of Java, C++ and CUDA. The first part is a parallel solution to pointer analysis driven by queries, on multi-core CPUs. It has achieved significant speedups over the sequential solution, since a large amount of unnecessary graph traversals have been eliminated by information sharing and query scheduling. The second part is an efficient GPU solution to whole-program pointer analysis. With effective load balancing and reduced redundant computation, it demonstrates considerable speedups over the state-of-the-art GPU implementation. The third part is a heterogeneous CPU-GPU solution to whole-program pointer analysis. It prioritises the distribution of different workloads to CPU/GPU according to the processing unit's ability for processing them, and therefore has achieved speedups over the corresponding CPU-only and GPU-only solutions. The effectiveness of each part of the framework is demonstrated via an evaluation with a set of open-source Java/C programs.

Publications

- **Yu Su**, Ding Ye, Jingling Xue and Xiangke Liao. An Efficient GPU Implementation of Inclusion-based Pointer Analysis. *IEEE Transactions on Parallel and Distributed Systems (TPDS '15)*. To Appear.
- Ding Ye, **Yu Su**, Yulei Sui and Jingling Xue. WPBound: Enforcing Spatial Memory Safety Efficiently at Runtime with Weakest Preconditions. *IEEE International Symposium on Software Reliability Engineering (ISSRE '14)*.
- **Yu Su**, Ding Ye and Jingling Xue. Parallel Pointer Analysis with CFL-Reachability. *IEEE International Conference on Parallel Processing (ICPP '14)*.
- **Yu Su**, Ding Ye and Jingling Xue. Accelerating Inclusion-based Pointer Analysis on Heterogeneous CPU-GPU Systems. *IEEE International Conference on High Performance Computing (HiPC '13)*.
- Peng Di, Ding Ye, **Yu Su**, Yulei Sui and Jingling Xue. Automatic Parallelization of Tiled Loop Nests with Enhanced Fine-Grained Parallelism on GPUs. *IEEE International Conference on Parallel Processing (ICPP '12)*.

Acknowledgements

First and foremost, I would like to express my deepest gratefulness to my supervisor, Prof. Jingling Xue. He is always there, ready to help me in all aspects, from the very beginning when I applied for a PhD. Through all stages of my doctoral study, his broad knowledge and sharp insight have kept me on track of an interesting and challenging research path. He spares no effort in teaching me how to think and write. Looking back at the three and half years at the CORG group, I feel amazed how much time and energy he has spent in inspiring and guiding me. My sincere gratitude also goes to the funding I received towards PhD: the UNSW Tuition Fee Scholarship, and a research stipend from my supervisor.

I am very much thankful to Ding Ye, my husband and colleague, for his unconditional love and support through my entire doctoral study. Without his great patience and effort, my path towards PhD would not have been so smooth and delightful. Without his continuous encouragement, I would not have had such advances in my personal and career development.

I would like to thank Dr. Peng Di for his dedicated collaboration on my work. He surprises me by how good he is at coming up with new ideas. Thank Dr. Yulei Sui for his willingness to share and help. His strength in programming and thirst for communication have inspired me a lot. I also give my sincere thanks to other colleagues from the CORG group, Xinwei Xie, Lei Shang, Sen Ye, Yue Li, Hao

Zhou, Xiaokang Fan, Tian Tan, Hua Yan, and Feng Zhang. It was a pleasant experience to work with them all during these unforgettable days.

Thank Yu Chen, my best friend, for accompanying me through my PhD study. She encourages and supports me whenever I am down and shows me the positive side. She is always ready to share her great experience to me, from which I benefit greatly.

Lastly, I would like to give my gratefulness and love to my mom and dad for their unconditional love that gives me everything I need.

Contents

Abstract	i
Publications	iii
Acknowledgements	iv
List of Figures	xi
List of Tables	xii
List of Algorithms	xiii
1 Introduction	1
1.1 Challenges	2
1.2 Our Approaches	4
1.3 Contributions	6
1.4 Thesis Organisation	7
2 Background	9
2.1 Whole-Program Pointer Analysis	9
2.1.1 Andersen’s Inclusion-based Pointer Analysis	9
2.1.2 State-of-the-Art	15

2.2	Demand-Driven Pointer Analysis	18
2.2.1	Program Representation	19
2.2.2	CFL-Reachability-based Pointer Analysis	20
3	Demand-Driven Analysis on Multi-CPU	26
3.1	Methodology	26
3.1.1	A Naive Parallelisation Strategy	27
3.1.2	Data Sharing	27
3.1.3	Query Scheduling	32
3.2	Evaluation	36
3.2.1	Implementations	36
3.2.2	Experimental Settings	37
3.2.3	Methodology	37
3.2.4	Performance Results	39
3.3	Related Work	46
3.4	Chapter Summary	46
4	Whole-Program Analysis on GPU	48
4.1	Challenges and Motivation	48
4.1.1	Load Imbalance	49
4.1.2	Redundant Traversals	50
4.2	Methodology	51
4.2.1	Imbalance-Aware Workload Partitioning	51
4.2.2	Adaptive Group Propagation	59
4.3	Evaluation	65
4.3.1	Methodology and Benchmarks	66
4.3.2	Experimental Settings	67

4.3.3	Speedups	69
4.3.4	Effectiveness of Workload Balancing	70
4.3.5	Effectiveness of Adaptive Group Propagation	72
4.3.6	Comparing with a Parallel CPU Implementation	74
4.4	Related Work	75
4.4.1	Sequential Andersen’s Analysis	75
4.4.2	Parallel Pointer Analysis	76
4.4.3	Parallel Graph Algorithms	77
4.5	Chapter Summary	78
5	Whole-Program Analysis on Heterogeneous CPU-GPU	79
5.1	Motivation	80
5.2	Method	81
5.2.1	Graph Data Representation	82
5.2.2	Managing Communication between CPU and GPU	82
5.2.3	Partitioning Computation for CPU and GPU	85
5.3	Optimisations	91
5.3.1	Optimisation I: On Hiding Communication Overhead	91
5.3.2	Optimisation II: On ΔP -Equivalence and R_{store}	92
5.3.3	Optimisation III: On Adaptive Heterogeneity	92
5.4	Evaluation	92
5.4.1	Implementations	93
5.4.2	Speedups	95
5.4.3	Dynamic Workload Balancing	97
5.4.4	Optimisations	98
5.4.5	Overall Effectiveness	100
5.5	Related Work	100

5.6	Chapter Summary	101
6	Conclusions and Future Work	102
6.1	Conclusions	102
6.2	Future Work	106
	Bibliography	108

List of Figures

2.1	An example illustrating Andersen’s analysis.	14
2.2	Sparse bit vector representing $\{958, 1920\}$	17
2.3	Syntax of PAG (pointer assignment graph).	19
2.4	A Java example and its PAG.	21
3.1	Adding jmp edges by graph rewriting.	29
3.2	Syntax of extended PAG.	30
3.3	An example of query scheduling.	35
3.4	Speedups of our parallel implementation.	40
3.5	Histograms of jmp edges.	43
3.6	Speedups of our parallel modes with different numbers of threads.	44
4.1	Execution times of all iterations in vim	49
4.2	The points-to edges for vim	50
4.3	Distributions of non-zero outgoing degree $ E_{\mathcal{P}}(x) $ and $ E_{\mathcal{C}}(x) $ in vim	50
4.4	Two lock-free atomic operations for merging an element.	55
4.5	Comparing individual and group propagation schemes.	61
4.6	Effectiveness of load balancing achieved by the two proposed schemes.	71
5.1	A cost-benefit analysis for the svn benchmark.	85
5.2	The shared worklist \mathcal{W} used in Algorithm 11.	87

5.3	Sizes of points-to and copy edges for svn	88
5.4	The shared worklist \mathcal{W} used in Algorithm 12.	90
5.5	Speedups of our parallel Andersen’s analysis and their CPU-only and GPU-only versions.	96

List of Tables

2.1	Initialisation: mapping program statements to constraint edges. . .	10
2.2	Constraint resolution: graph-rewriting rules.	11
3.1	Benchmark information.	38
3.2	Statistics for data sharing.	42
3.3	Statistics for query scheduling.	45
4.1	Benchmark statistics: sizes of initial constraint graphs.	66
4.2	Kernel configurations.	68
4.3	Analysis times and speedups.	69
4.4	Effectiveness of adaptive group propagation.	73
4.5	The speedups of <i>Ref</i> , <i>IG</i> and <i>CPU</i> -16 over the baseline.	74
5.1	Constraint resolution: graph-rewriting rules.	83
5.2	Benchmark suite: sizes of initial constraint graphs.	93
5.3	Analysis of our CPU-GPU solution (including its key method employed).	97
5.4	Analysis of our CPU-GPU solution (including its key method and optimisations employed).	99
6.1	Comparing different parallel pointer analysis.	103

List of Algorithms

1	Sequential Andersen’s Analysis.	12
2	CFL-reachability-based pointer analysis.	24
3	REACHABLENODES with data sharing.	31
4	Imbalance-Aware Workload Partitioning.	52
5	Imbalance-Aware Propagation.	53
6	Performing $dst \leftarrow dst \uplus src$ by a warp.	57
7	Atomic insertion of one element from src into dst	57
8	Atomic disjunction of an element from src with another in dst	58
9	Implementing $\pi_{src} \leftarrow \pi_{src \rightarrow next}$ for a virtual sparse bit vector.	64
10	A CPU-GPU solution of Andersen’s analysis.	81
11	A naive workload distribution scheme.	87
12	A DI-based dynamic workload distribution.	90

Chapter 1

Introduction

Pointers are an essential feature of mainstream programming languages (e.g., C, C++, Java). Pointer analysis, which determines statically what a pointer may point to at runtime, is a fundamental analysis for understanding the behaviours of a program. This analysis provides points-to information, which enables many applications such as compiler optimisation [7], bug detection [9, 66] and security analysis [6]. With a rich literature, it has been one of the most popular research areas in computer science [18]. Depending on how points-to information is provided, two types of pointer analyses have been studied for long: (1) whole-program analysis, which computes points-to information for all variables in the program; and (2) demand-driven analysis, which determines the points-to information only for the variables that are requested by clients. Both types of analyses are important, with the whole-program analysis incorporated in production-quality compilers for all kinds of analysis and transformation, and the demand-driven analysis laying foundations for clients such as software debugging [43, 45, 48] and alias disambiguation [52].

The main problem with developing a practical pointer analysis tool is enhanc-

ing its efficiency while maintaining its precision. Great efforts have been made on making tradeoffs between precision and efficiency across several dimensions, including field-sensitivity (by matching field accesses) [37], context-sensitivity (by distinguishing calling contexts) [44, 45, 52, 60, 62, 68] and flow-sensitivity (by considering control flow) [15, 23, 57, 68]. Over the years, many performance improvements have been made, in the sequential setting, on accelerating both whole-program analysis [10, 11, 14, 34, 36, 38, 52] and demand-driven analysis [26, 42, 43, 45, 48, 65, 71]. However, precise pointer analysis is still costly when applied to large programs. In recent years, with the ubiquity of parallel platforms, including multicore CPUs [8, 29, 31, 33, 40] and GPUs [28, 32], boosting the performance of pointer analysis by exploring parallelisation techniques has become an increasingly popular research topic.

1.1 Challenges

Pointer analysis, which is essentially an irregular graph problem, is very challenging to parallelise efficiently on multicore CPUs, GPUs or heterogeneous CPU-GPU systems. Whole-program pointer analysis, which often comes in Andersen’s inclusion-based style (by considering pointer assignments as subset constraints) [2], is a graph algorithm that makes extensive modifications to the underlying graph representing the program being analysed. Demand-driven pointer analysis, which is driven by queries from clients and founded on Context-Free Language (CFL) reachability [41], is a graph traversal problem with massive parallelism-inhibiting dependences.

CFL-reachability-based demand-driven pointer analysis has never been parallelised in the literature. Previous studies are focused on its sequential algorithm, by resorting to refinement [45, 48, 71], summarisation [43, 65], incrementalisa-

tion [26, 42], pre-analysis [64] and ad-hoc caching [45, 64]. On parallel platforms, however, it is crucial but difficult to avoid the redundant graph traversals both within and across the queries, in order to fully utilise the computational resources. The new points-to/alias information discovered in answering some queries during graph traversals is not directly available to other queries on the (read-only) graph representation of the program. For precise CFL-reachability-based analysis with field- and context-sensitivity, there are massive parallelism-inhibiting dependences as well. These dependences are introduced into its various analysis stages since processing a query involves matching calling contexts and handling heap accesses throughout the program.

Andersen’s inclusion-based pointer analysis is promising yet difficult to parallelise on GPUs. In recent years, several parallel implementations of Andersen’s analysis have been introduced to enhance its performance on multicore CPUs [29, 40] and GPUs [28]. The latter platform is chosen for such an analysis in this thesis since a GPU is possible to deliver better performance than a multicore CPU at a lower price [28]. However, existing work [28] is still inefficient due to extensive modifications to the underlying graph representing the program being analysed. These modifications are highly irregular, input-dependent and statically unpredictable. The skewed and dynamically-changing distribution of edges in a graph leads to a highly imbalanced workload distribution across its nodes, where the higher-degree nodes induce larger processing tasks. As a result, GPUs can perform poorly when their computational resources are severely underutilised.

It is another challenge to parallelise Andersen’s pointer analysis on a heterogeneous CPU-GPU system. The obstacle lies in making full use of computational resources of both CPU and GPU. It is important to notice that CPUs behave better for some applications while GPUs prevail for others, indicating that the relative

suitability of CPUs or GPUs for applications varies depending on their workload characteristics. The challenges in exploring a heterogeneous CPU-GPU system are two-fold. First, the workload distribution between the CPU and GPU must be balanced with negligible runtime overhead. This is non-trivial since different programs give rise to different graphs to be analysed and the structure of a graph changes unpredictably during the analysis. Second, the CPU-GPU communication must be minimised in terms of the amount of data exchanged and the degree of overlap with computation on CPU and GPU. This is also non-trivial because the graphs being analysed are dynamically changing, making it hard to extract the “right” amount of information to communicate between the CPU and GPU with reasonable overhead.

1.2 Our Approaches

In order to address the above challenges, this thesis proposes a series of parallelisation techniques to speed up both CFL-reachability-based pointer analysis and Andersen’s inclusion-based pointer analysis.

Reducing Unnecessary Graph Traversals for CFL-Reachability-based Analysis Redundant graph traversals can be the major cost to CFL-reachability-based pointer analysis. We present two novel techniques, *data sharing* and *query scheduling*, to reduce redundant graph traversals. With data sharing, the original graph traversal problem is recast into a graph rewriting problem. By adding new edges to the graph representation to shortcut the paths traversed in a query, re-traversing (redundantly) the same paths can be avoided by taking their short-cuts instead when handling subsequent queries. With query scheduling, the queries to be issued (in batch mode) are prioritised according to their statically estimated

dependences so that more redundant graph traversals can be further reduced.

Balancing Workload for Whole Program Analysis on GPU Load imbalance causes underutilisation in GPU cores and results in severe performance issues. We propose an imbalance-aware workload partitioning scheme that divides the workload dynamically among the concurrent warps, initially in a warp-centric manner as in [28] (during its coarse-grain stage) but later switches to a task-pool-based model (during its fine-grain stage) as soon as a workload imbalance is detected. The coarse-grain stage makes use of a so-called pull-based method so that addition of edges can be performed in a synchronisation-free manner. This stage is efficient as long as workload is balanced and the computational resources on the GPU are fully utilised. The fine-grain stage tackles situations where load imbalance is detected by further decomposing long tasks. When propagating edge sets for nodes during the analysis, there can be redundant traversals and computations. We propose an adaptive group propagation scheme to reduce unnecessary graph traversals and edge set propagations. This scheme also facilitates load balancing when redundant traversals and propagations involving high-degree nodes are reduced.

Workload Distribution and Communication Management for Heterogeneous Systems CPUs and GPUs behave in different manners in processing workloads with distinct characteristics. We prioritise the distribution of different types of graph-rewriting rules (i.e., workloads) to CPU or GPU according to the degrees of the processing unit’s suitability for processing them. In this way, both CPU and GPU are more likely to get the workload which they are able to process more efficiently. With separate memory spaces, communication can be a major cost for the analysis. We adopt difference propagation to transfer new points-to

information between CPU and GPU and overlap this process with some computations.

1.3 Contributions

Demand-Driven Pointer Analysis on Multicore CPUs This is the first parallel implementation of context- and field-sensitive pointer analysis with CFL-reachability for Java programs. This implementation uses a data sharing scheme to reduce redundant graph traversals (in all query-processing threads) by graph rewriting, and a query scheduling scheme to eliminate unnecessary traversals further, by prioritising queries according to their statically estimated dependences. Evaluated with a set of 20 Java benchmarks, this parallel solution achieves an average speedup of 16.2X over a state-of-the-art sequential implementation with 16 threads on 16 CPU cores.

Efficient Whole-Program Pointer Analysis on GPU While [28] focuses on producing the first-ever GPU implementation of Andersen’s analysis via graph-rewriting, this thesis leverages that implementation but addresses its load balancing problem, thereby accelerating this analysis significantly further as well as providing insights for parallelising other graph algorithms that also make modifications to their input graphs. This is achieved by an imbalance-aware workload partitioning scheme that includes a warp-centric stage (coarse-grain) and a task-pool-based stage (fine-grain), and an adaptive group propagation scheme to avoid some redundant graph traversals and computations. The evaluation in terms of 14 C benchmarks used in prior work shows that this parallel implementation achieves a significant speedup of 46% on average over the state-of-the-art GPU implementation [28].

Whole-Program Pointer Analysis on Heterogeneous CPU-GPU Systems

This is the first implementation of Andersen’s analysis on a heterogeneous CPU-GPU system. It takes advantage of a previous formulation of this analysis in terms of graph-rewriting rules [28] to ensure that all rule applications on CPU and GPU are synchronisation-free. It proposes a dynamic workload distribution scheme that dispatches a particular type of workload to the processor, CPU or GPU, that is better suited for the workload. It employs a difference propagation scheme for transferring new points-to information discovered between CPU and GPU to reduce communication cost. The evaluation using seven C benchmarks shows that this CPU-GPU solution outperforms (on average) (1) the CPU-only solution by 50.6%, (2) the GPU-only solution by 78.5%, and (3) an oracle that behaves as the faster of (1) and (2) on every benchmark by 34.6%, where (1) and (2) are variants of state-of-the-art implementations introduced in [28].

1.4 Thesis Organisation

The rest of this thesis is organised as follows:

Chapter 2 provides the general background of pointer analysis. As the targets of our parallel solutions in this thesis, Andersen’s inclusion-based pointer analysis and CFL-reachability-based pointer analysis are introduced in details.

Chapter 3 presents the first parallel solution to CFL-reachability-based pointer analysis. With a data sharing scheme and a query scheduling scheme, it successfully reduces redundant traversals in the analysis.

Chapter 4 describes an efficient GPU solution to Andersen’s inclusion-based pointer analysis, which is based on an state-of-the-art GPU implementation and improves its performance with an imbalance-aware workload balancing scheme and

a group propagation scheme.

Chapter 5 introduces the first parallel solution to Andersen’s inclusion-based pointer analysis on a heterogeneous CPU-GPU system. It presents techniques for matching suitable workload to CPU or GPU and minimising communications between them.

Chapter 6 concludes the thesis and discusses future work.

Chapter 2

Background

This thesis focuses on both whole-program and demand-driven analyses, given their importance in program analysis and software engineering. We provide some preliminaries on these two types of pointer analyses in Sections 2.1 and 2.2 respectively, before introducing our parallel solutions to them.

2.1 Whole-Program Pointer Analysis

Due to its scalability, Andersen’s pointer analysis, an inclusion-based flow- and context-insensitive whole-program analysis, has been adopted by production compilers such as Open64, LLVM and GCC, and also has become the focus of this thesis. Section 2.1.1 introduces this analysis with its sequential algorithm and Section 2.1.2 describes its state-of-the-art parallel implementation on a GPU.

2.1.1 Andersen’s Inclusion-based Pointer Analysis

Andersen’s analysis for a program is often formulated by solving a set-constraint problem over a directed graph, $G = (V, E)$, called a *constraint graph*. We first

discuss its initialisation and constraint resolution, then present its algorithm, and finally, illustrate it with an example.

Initialisation Given a C program, its constraint graph $G = (V, E)$ is created with its node set V being the variables in the program and its edge set E representing five different types of pointer-manipulating statements in the program, as shown in Table 2.1. At this stage, there is one edge for every such a statement in the program.

Name	Statement	Edge
points-to (\mathcal{P})	$x = \&y$	$x \xrightarrow{\mathcal{P}} y$
copy (\mathcal{C})	$x = y$	$x \xrightarrow{\mathcal{C}} y$
load (\mathcal{L})	$x = *y$	$x \xrightarrow{\mathcal{L}} y$
store (\mathcal{S})	$*x = y$	$x \xrightarrow{\mathcal{S}} y$
offset (\mathcal{F})	$x = y + o$	$x \xrightarrow[\mathcal{F}]{o} y$

Table 2.1: Initialisation: mapping program statements to constraint edges.

For example, $x \xrightarrow{\mathcal{P}} y$ represents a points-to (i.e., \mathcal{P}) edge directing out of node x into node y . The four other types of edges are understood similarly. Offset edges are introduced for enabling field-sensitivity so that a field of a struct is treated as a separate variable. However, an array is usually analysed as a whole (with its elements collapsed).

We use E_t to represent a subset of E containing all the edges of type t . For example, $E_{\mathcal{P}}$ is the set of all \mathcal{P} edges in E and $E_{\mathcal{C}}$ is the set of all \mathcal{C} edges in E . This notation will be used heavily in many algorithms presented later in the thesis.

Constraint Resolution Once the constraint graph $G = (V, E)$ for a program is initialised, we can start discovering new points-to information in the program

during a so-called constraint resolution process. V (as the set of program variables) will obviously stay unchanged. However, E may grow as new points-to (\mathcal{P}) and copy (\mathcal{C}) edges are added by applying graph-rewriting rules iteratively until a fixed-point is reached. Note that the load (\mathcal{L}), store (\mathcal{S}) and offset (\mathcal{F}) edges will remain the same.

Propagating the points-to information initially available in G along its edges iteratively is time-consuming. In practice, we can avoid some redundant computations by propagating only the new information discovered to where it is needed. This can be achieved with *difference propagation* [11, 36, 46], in which the newly added $\delta\mathcal{P}$ and $\delta\mathcal{C}$ edges to G in one iteration are distinguished from the \mathcal{P} and \mathcal{C} edges available before this iteration.

Rule	Semantics
$R_{\text{copy}}(x)$	$x \xrightarrow{\mathcal{C}} y \wedge y \xrightarrow{\Delta\mathcal{P}} z \Rightarrow x \xrightarrow{\delta\mathcal{P}} z$
$R_{\text{load}}(x)$	$x \xrightarrow{\mathcal{L}} y \wedge y \xrightarrow{\Delta\mathcal{P}} z \Rightarrow x \xrightarrow{\delta\mathcal{C}} z$
$R_{\text{store}}(x)$	$x \xrightarrow{\Delta\mathcal{P}^{-1}} y \wedge y \xrightarrow{\mathcal{S}} z \Rightarrow x \xrightarrow{\delta\mathcal{C}} z$
$R_{\delta\text{copy}}(x)$	$x \xrightarrow{\delta\mathcal{C}} y \wedge y \xrightarrow{\mathcal{P}} z \Rightarrow x \xrightarrow{\delta\mathcal{P}} z$
$R_{\text{offset}}(x)$	$x \xrightarrow[\mathcal{O}]{\mathcal{F}} y \wedge y \xrightarrow{\Delta\mathcal{P}} z \Rightarrow x \xrightarrow{\delta\mathcal{P}} z + \mathcal{O}$

Table 2.2: Constraint resolution: graph-rewriting rules.

Table 2.2 lists five types of graph-rewriting rules used in constraint resolution. Note that there are two types of newly added points-to edges, $\Delta\mathcal{P}$ and $\delta\mathcal{P}$. For each iteration, the $\Delta\mathcal{P}$ edges in the premise signify the new points-to edges generated in the previous iteration and being used in the current iteration, and the $\delta\mathcal{P}$ edges in the conclusion represent the new points-to edges being produced in the current iteration. For new copy edges, only one version, i.e., $\delta\mathcal{C}$ is maintained, because they are produced by R_{load} and R_{store} and used only later by $R_{\delta\text{copy}}$ in the same iteration

(by Algorithm 1).

Algorithm 1 Sequential Andersen's Analysis.

```

Procedure ANDERSEN()
  begin
1   $G = (V, E) \leftarrow \text{CREATEGRAPH}();$ 
2  repeat
3     $\text{APPLY}(\mathcal{C}, \Delta\mathcal{P}, \delta\mathcal{P}, V);$ 
4     $\text{APPLY}(\mathcal{L}, \Delta\mathcal{P}, \delta\mathcal{C}, V);$ 
5     $\text{APPLY}(\Delta\mathcal{P}^{-1}, \mathcal{S}, \delta\mathcal{C}, V);$ 
6     $\text{APPLY}(\delta\mathcal{C}, \mathcal{P}, \delta\mathcal{P}, V);$ 
7     $\text{APPLYOFFSET}(\mathcal{F}, \Delta\mathcal{P}, \delta\mathcal{P}, V);$ 
8     $E_{\Delta\mathcal{P}} \leftarrow E_{\delta\mathcal{P}} \setminus E_{\mathcal{P}}; \quad // \text{ update}$ 
9     $E_{\mathcal{P}} \leftarrow E_{\mathcal{P}} \cup E_{\delta\mathcal{P}}; \quad // \text{ update}$ 
10    $E_{\mathcal{C}} \leftarrow E_{\mathcal{C}} \cup E_{\delta\mathcal{C}}; \quad // \text{ update}$ 
  until fixed-point;

  Procedure APPLY( $t1, t2, t3, V$ )
  begin
11   foreach  $x \in V$  do
12     foreach  $y \in E_{t1}(x)$  do
13        $E_{t3}(x) \leftarrow E_{t3}(x) \cup E_{t2}(y);$ 

  Procedure APPLYOFFSET( $t1, t2, t3, V$ )
  begin
14   foreach  $x \in V$  do
15     foreach  $y \in E_{t1}(x)$  do
16       Let this offset edge  $(x, y)$  be  $x \xrightarrow[o]{\mathcal{F}} y;$ 
17        $E_{t3}(x) \leftarrow E_{t3}(x) \cup \{z + o \mid z \in E_{t2}(y)\};$ 
  
```

Algorithm Algorithm 1 gives a sequential procedure for performing Andersen's analysis. Given a program, CREATEGRAPH is called to initialise its constraint graph $G = (V, E)$ as discussed earlier and the graph-rewriting rules in Table 2.2 are applied in sequence until a fixed-point is reached as discussed above. Recall that we use E_t to represent a subset of E containing all the edges of type t . In this algorithm, for example, $E_{\mathcal{P}}$ is the set of all \mathcal{P} edges in E and $E_{t1}(x)$ denotes the

set of edges of type $t1$ associated with the node x being processed.

For each rule of the form of $x \xrightarrow{t1} y \wedge y \xrightarrow{t2} z \Rightarrow x \xrightarrow{t3} z$ in Table 2.2, we call **APPLY** or **APPLYOFFSET**, whichever is appropriate, to apply it to every node x in the program to discover any new edges of type $t3$ for node x . At the end of each iteration (lines 8 – 10), the $\Delta\mathcal{P}$, \mathcal{P} and \mathcal{C} edges are updated to prepare for the next iteration.

Example Let us apply Algorithm 1 to a program given in Figure 2.1(a). The analysis starts with the constraint graph created by **CREATEGRAPH** in Figure 2.1(b). This graph is then modified into the ones in Figures 2.1(c) – 2.1(e) during iterations 1 – 3, respectively. A fixed-point is reached at the end of iteration 3. In each iteration, the newly discovered edges are highlighted in dashed arrows. Figure 2.1(f) illustrates the difference propagation in terms of the $\Delta\mathcal{P}$, $\delta\mathcal{P}$ and $\delta\mathcal{C}$ edges during each iteration.

In the first iteration, $E_{\Delta\mathcal{P}}$ contains $x \xrightarrow{\Delta\mathcal{P}} a$ and $a \xrightarrow{\Delta\mathcal{P}} o$, which were added during initialisation. R_{copy} is applied to y , resulting in $y \xrightarrow{\delta\mathcal{P}} a$ added. At this stage, none of the rules is applicable. Then $y \xrightarrow{\delta\mathcal{P}} a$ is included to the $\Delta\mathcal{P}$ and \mathcal{P} edges (lines 8 – 9).

In the second iteration, applying R_{load} to b causes $b \xrightarrow{\delta\mathcal{C}} a$ to be discovered. Then $R_{\delta\text{copy}}$ is applied to b with $b \xrightarrow{\delta\mathcal{P}} o$ added. Next, R_{offset} is applied to z , with $z \xrightarrow{\delta\mathcal{P}} a + 2$ discovered. Finally, the $\Delta\mathcal{P}$, \mathcal{P} and \mathcal{C} edges are updated accordingly (lines 8 – 10).

In the last iteration, R_{store} is applied to $a + 2$, giving rise to $a + 2 \xrightarrow{\delta\mathcal{C}} b$, and resulting in $a + 2 \xrightarrow{\delta\mathcal{P}} o$ to be discovered by $R_{\delta\text{copy}}$. These two new edges become $a + 2 \xrightarrow{\mathcal{C}} b$ and $a + 2 \xrightarrow{\mathcal{P}} o$ (as $a + 2 \xrightarrow{\Delta\mathcal{P}} o$ initially due to difference propagation) at the end of this iteration. A fixed-point is then reached since no more rules can be applied.

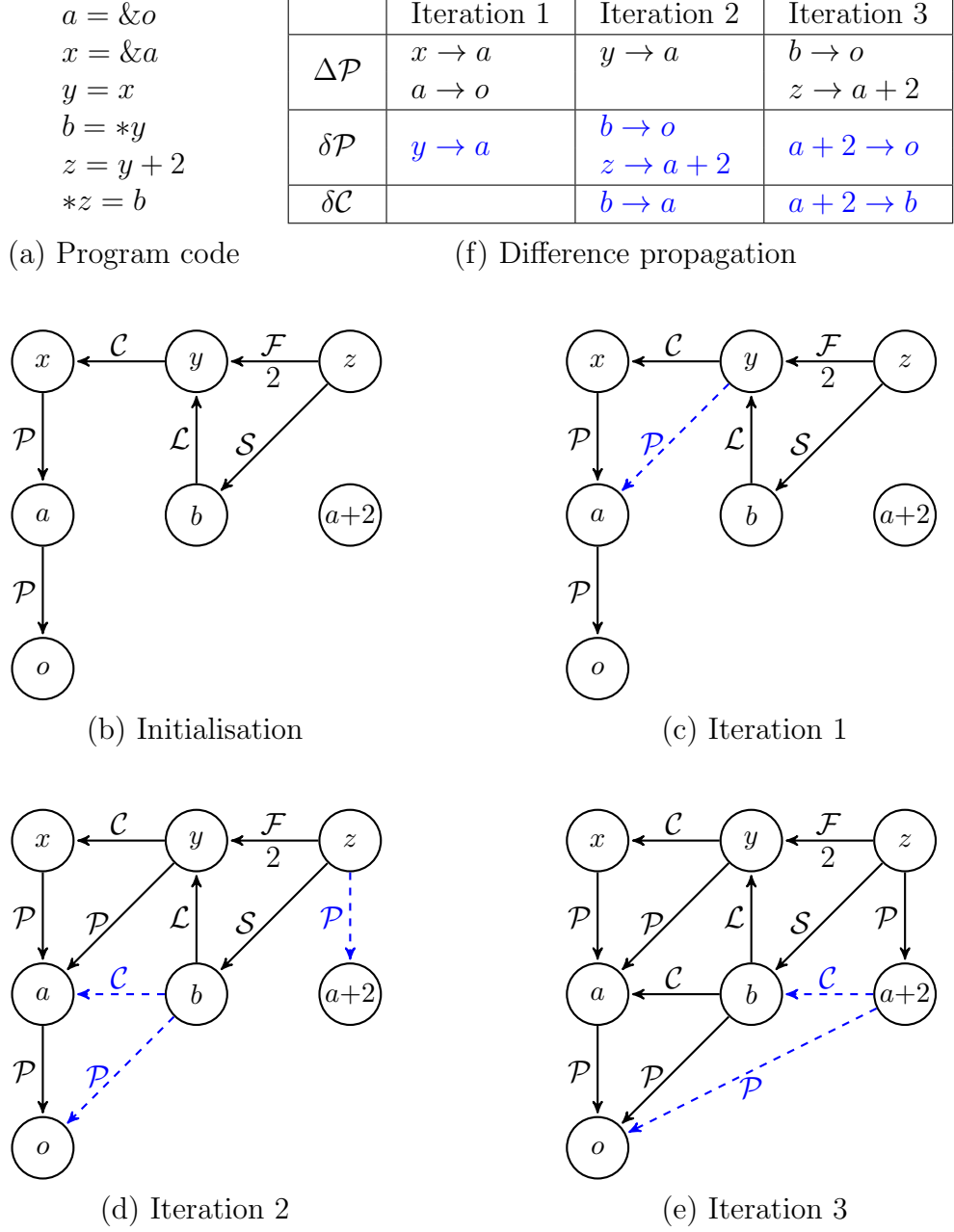


Figure 2.1: An example illustrating Andersen’s analysis. The initial graph is the one in (b), where a is a variable representing the address of a struct and the address of its first field, and $a+2$ denotes its third field. Note that for illustration purposes, this graph only shows variables directly relevant to the example. For instance, the second field $a+1$, which never occurs in the program, is not shown in the graph. The new edges added in an iteration are shown in dashed arrows. The fixed-point is reached after iteration 3.

2.1.2 State-of-the-Art

We explain the first GPU implementation of Andersen’s analysis [28]. We start by reviewing the GPU architecture and the CUDA programming model. We then describe a sparse bit vector representation that is often used in pointer analysis but adapted for this GPU implementation. Finally, we examine its workload partitioning scheme used, together with its inherent load balancing issues.

GPU Architecture We use an NVIDIA TESLA K20c GPU in our evaluation, but the general techniques introduced in this thesis are not tied to any specific GPU architecture. This GPU consists of 13 streaming multiprocessors (SMXs), each containing 192 cores, giving rise to 2496 GPU cores in total. In addition, the GPU has a global memory (i.e., off-chip memory) bandwidth of 208GB/s. However, each GPU core clocks at only 0.71GHz and the global memory access latency reaches 400 – 800 cycles. This suggests that a GPU is well suited for regular, balanced workloads with abundant data parallelism when its massive number of cores and high memory bandwidth are fully utilised. However, a GPU can perform poorly when its cores and memory bandwidth are underutilised, which usually happens for irregular, imbalanced workloads.

In the CUDA programming model, threads are organised in *thread blocks*. All the threads in one block are executed on one SMX, while one SMX can have multiple concurrent blocks. One block has several *warps*, each containing 32 parallel threads.

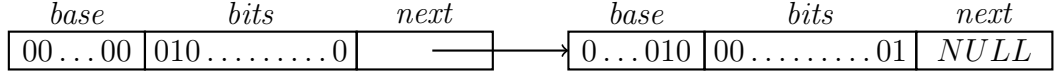
In the case of imbalanced workloads, the computational resources in a GPU may be underutilised in two scenarios. First, all the threads in a warp execute the same instruction concurrently in an SIMD (Single Instruction Multiple Data) manner, but can be severely serialised on divergent control flows (known as *intra-warp divergence*). Second, all the warps in a thread block running on an SMX are

time-shared to hide memory latency, but a few long-running warps may lead to underutilisation of its cores (known as *inter-warp load imbalance*).

Sparse Bit Vectors A sparse bit vector, implemented as a singly linked list, represents a set of integers. Each element in the list consists of three fields: *bits* (several words) represents whether the corresponding integer belongs to the set, *base* (1 word) indicates the range of integers that is represented in this element, and *next* (1 word) points to the next element. The prior work in pointer analysis [14] has demonstrated that sparse bit vectors are compact for representing constraint graphs while facilitating efficient applications of graph-rewriting rules. In their GPU implementation of Andersen’s analysis, [28] used 32-word elements, where the *bits* field spans 30 words (960 bits). This helps mitigate intra-warp divergence, since all 32 threads in a warp can perform operations (e.g., a coalesced global memory access [61, 70] and a bitwise *OR* for \cup) on the 32 words in parallel.

In a constraint graph $G = (V, E)$, all the variables, i.e., nodes in V are mapped to consecutive integers, starting from 0. Given a node x , all its (outgoing) edges in $E_t(x)$ of type t are stored in a separate list. Let us consider an example when x has two outgoing points-to edges $x \xrightarrow{P} y$ and $x \xrightarrow{P} z$, of type t , where y and z are identified by integers 958 and 1920, respectively. The sparse representation of x ’s (outgoing) edges of type t is illustrated in Figure 2.2. It is a linked list with two elements. The first element’s *base* is 0 and the 958^{th} bit in *bits* is 1. So it contains the integer $960 \times 0 + 958 = 958$, i.e., y . The second element’s *base* is 2 and the 0^{th} bit in *bits* is 1. As a result, it contains z , identified by integer $960 \times 2 + 0 = 1920$. Note that a sparse bit vector is implemented as a singly sorted linked list, with its elements sorted in increasing order of their *bases*. In addition, different elements on the same list will always have different *bases*.

With sparse bit vectors, the set union operation in line 13 in Algorithm 1 can

Figure 2.2: Sparse bit vector representing $\{958, 1920\}$.

be implemented as follows. If $z \in E_{t2}(y)$ has a *base* that is not available in $E_{t3}(x)$, then z is inserted into $E_{t3}(x)$ wherever appropriate. Otherwise, the *bits* field of z is merged (via a bitwise *OR*) with the *bits* part of the element in $E_{t3}(x)$ that shares the same *base*. The set union in line 17 in Algorithm 1 is handled similarly.

Warp-Centric Workload Partitioning In the GPU implementation introduced in [28], Andersen’s analysis (given in Algorithm 1) is parallelised in a warp-centric manner [19]. During each iteration (of its **repeat** loop), the five types of rules (in lines 3 – 7) are applied sequentially, with synchronisation enforced between every two consecutive rule applications. However, each rule is applied in parallel to all the nodes in the program. So the **foreach** loops in lines 11 and 14 are essentially DOALL loops. When applied to every individual node x (lines 12 – 13 in **APPLY** and lines 15 – 17 in **APPLYOFFSET**), each rule is executed by one warp, $warp_x$, in a warp-centric manner. As $warp_x$ will add new edges only at node x and such new edges are never used by the other concurrent warps (due to difference propagation), different applications of the same rule at different nodes do not require any synchronisation.

To apply R_{store} , the pointed-by $(\Delta\mathcal{P}^{-1})$ edges are required. For space considerations, $E_{\Delta\mathcal{P}^{-1}}$ is not explicitly stored as it can be obtained from its inverse $E_{\Delta\mathcal{P}}$ efficiently. For this rule, the set $\Delta\mathcal{P}$ involved is small since the number of store edges is relatively small and remains unchanged during the analysis. In addition, pairs (x, y) are extracted from $E_{\Delta\mathcal{P}}$ only when both y has some outgoing store edges and $y \xrightarrow{\mathcal{P}} x$ holds. All pairs with the same first component are assigned to

the same warp.

For this particular GPU implementation, intra-warp divergence is mitigated by letting the 32 threads in a warp run in parallel on a 32-word element in a sparse bit vector, with one word per thread. However, this simple warp-centric approach suffers from underutilisation of warp resources, since some warps processing low-degree nodes finish much earlier than those processing high-degree nodes. Several heuristics are employed in [28] to alleviate this inter-warp imbalance problem. First, due to the absence of read-write conflicts between $R_{\text{copy}}(x)$ (line 3) and $R_{\text{load}}(x)$ (line 4), no synchronisation is necessary in between. Second, $R_{\delta\text{copy}}(x)$ (line 6) is combined with both $R_{\text{load}}(x)$ and $R_{\text{store}}(x)$ (so that $R_{\text{copy}}(x)$ is applied immediately after each of these two rules at a node). Finally, $R_{\text{store}}(x)$ is applied in a block- rather than warp-centric manner so that a node is processed by all warps in a thread block. Despite these heuristics, the GPU implementation [28] still suffers from extremely imbalanced workloads for many programs.

2.2 Demand-Driven Pointer Analysis

Demand-driven pointer analysis is founded on CFL-reachability [41] [26, 42, 43, 45, 48, 65, 71]. By computing the points-to information of some variables (instead of all variables as in whole-program Andersen’s analysis), demand-driven analysis can be performed both context- and field-sensitively to achieve better precision more scalably than Andersen’s analysis, especially for Java programs. Section 2.2.1 describes the intermediate representation used for analysing Java programs. Section 2.2.2 reviews the standard formulation of pointer analysis in terms of CFL-reachability.

2.2.1 Program Representation

We focus on Java although this thesis applies equally well to C [71]. A Java program is represented as a *Pointer Assignment Graph (PAG)*, as defined in Figure 2.3.

n	$:=$	$v \mid o$	Node
v	$:=$	$l \mid g$	Variable
e	$:=$	$l_1 \xleftarrow{\text{new}} o$	Allocation
		$l_1 \xleftarrow{\text{assign}^l} l_2$	Local Assignment
		$g \xleftarrow{\text{assign}^g} v \mid v \xleftarrow{\text{assign}^g} g$	Global Assignment
		$l_1 \xleftarrow{\text{ld}(f)} l_2$	Load
		$l_1 \xleftarrow{\text{st}(f)} l_2$	Store
		$l_1 \xleftarrow{\text{param}_i} l_2$	Parameter
		$l_1 \xleftarrow{\text{ret}_i} l_2$	Return
			$l \in \text{Local} \quad g \in \text{Global} \quad o \in \text{Object}$
			$i \in \text{CallSite} \quad f \in \text{Field}$

Figure 2.3: Syntax of PAG (pointer assignment graph).

A node n represents a variable v or an object o in the program, where v can be local (l) or global (g). An edge e represents a statement in the program oriented in the direction of its value flow. An edge connects only to local variables (l_1 and/or l_2) unless it represents an assignment involving at least one global variable (**assign** ^{g}). Let us look at the seven types of edges in detail. $l_1 \xleftarrow{\text{new}} o$ captures the flow of object o to variable l_1 , indicating that l_1 points directly to o . $l_1 \xleftarrow{\text{assign}^l} l_2$ represents a local assignment ($l_1 = l_2$). A global assignment is similar except that one or both variables at its two sides are static variables in a class (i.e. g). $l_1 \xleftarrow{\text{ld}(f)} l_2$ represents a load of the form $l_1 = l_2.f$ and $l_1 \xleftarrow{\text{st}(f)} l_2$ represents a store of the form $l_1.f = l_2$. $l_1 \xleftarrow{\text{param}_i} l_2$ models parameter passing, where l_2 is an actual parameter and l_1 is its corresponding formal parameter, at call site i . Similarly, $l_1 \xleftarrow{\text{ret}_i} l_2$ indicates an

assignment of the return value in l_2 to l_1 at call site i .

Figure 2.4 gives an illustrating example and its PAG representation. Note that o_i denotes the object created at the allocation site in line i and v_m represents variable v declared in method m . Loads and stores to array elements are modeled by collapsing all elements into a special field, denoted **arr**.

2.2.2 CFL-Reachability-based Pointer Analysis

CFL-reachability [41] is an extension of traditional graph reachability. Let G be a directed graph with edges labelled by letters over an alphabet Σ and L be a CFL over Σ . Each path p in G is composed of a string $s(p)$ in Σ^* , formed by concatenating in order the labels of edges along p . A path p is an L -path iff $s(p) \in L$. Node x is L -reachable to y iff a path p from x to y exists, such that $s(p) \in L$. For a single-source reachability analysis, the worst-case time complexity is $O(\Gamma^3 N^3)$, where Γ is the size of a normalised grammar for L and N is the number of nodes in G .

Field-Sensitivity Let us start with a field-sensitive formulation without context-sensitivity. When calling contexts are ignored, there is no need to distinguish the four types of assignments, **assign**^{*l*}, **assign**^{*g*}, **param**_{*i*} and **ret**_{*i*}; they are all identified as assignment edges of type **assign**.

For a PAG G with only **new** and **assign** edges, the CFL L_{FT} (FT for flows to) is a regular language, meaning that a variable v is L_{FT} -reachable from an object o in G iff o can flow to v . The rule for L_{FT} is defined as:

$$\text{flowsTo} \rightarrow \text{new} (\text{assign})^* \quad (2.1)$$

with *flowsTo* as the start symbol. If o *flowsTo* v , then v is L_{FT} -reachable from o .

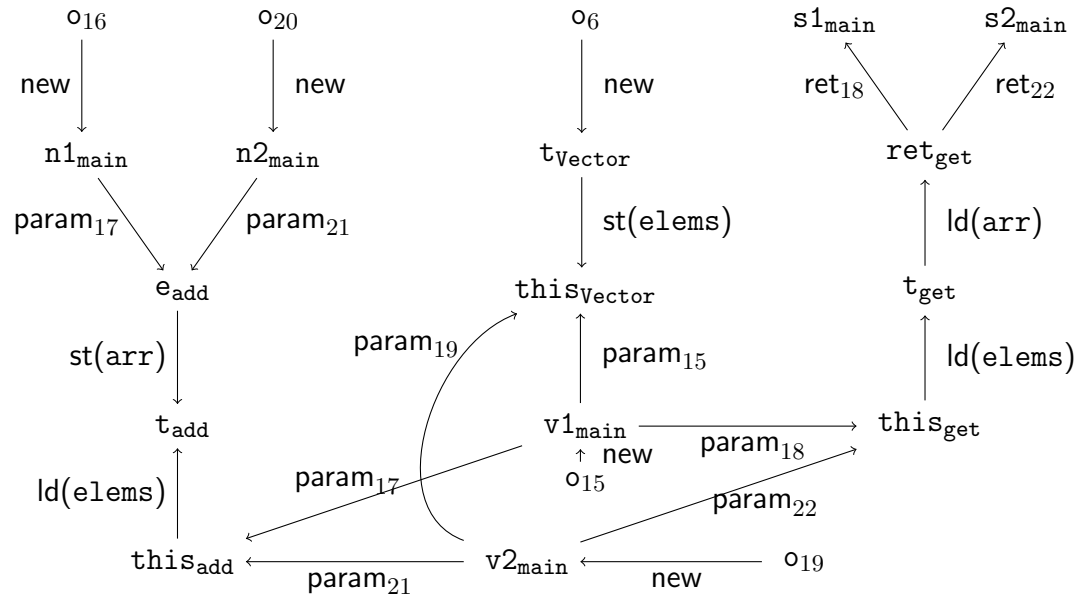
For example, in Figure 2.4(a), since $o_{15} \xrightarrow{\text{new}} v1_{\text{main}} \xrightarrow{\text{param}_{15}} \text{this}_{\text{vector}}$, o_{15} flows


```

1 class Vector {
2   Object[] elems;
3   int count;
4   Vector(){
5     count = 0;
6     t = new Object[MAXSIZE];
7     this.elems = t;}
8   void add(Object e){
9     t = this.elems;
10    t[count++] = e;} // W t.arr
11   Object get(int i){
12     t = this.elems;
13     return t[i];} // R t.arr
14 static void main(String[] args){
15   Vector v1 = new Vector();
16   String n1 = new String("N1");
17   v1.add(n1);
18   Object s1 = v1.get(0);
19   Vector v2 = new Vector();
20   Integer n2 = new Integer(1);
21   v2.add(n2);
22   Object s2 = v2.get(0);}

```

(a) Java Code.



(b) PAG.

Figure 2.4: A Java example and its PAG.

to `thisvector`.

When field accesses are also handled, the CFL L_{FS} (FS for field-sensitivity) is defined as follows:

$$\begin{aligned}
 \text{flowsTo} &\rightarrow \text{new} (\text{assign} \mid \text{st}(f) \text{ alias } \text{ld}(f))^* \\
 \text{alias} &\rightarrow \overline{\text{flowsTo}} \text{ flowsTo} \\
 \overline{\text{flowsTo}} &\rightarrow (\overline{\text{assign}} \mid \overline{\text{ld}(f)} \text{ alias } \overline{\text{st}(f)})^* \overline{\text{new}}
 \end{aligned} \tag{2.2}$$

The rule for *flowsTo* matches fields via $\text{st}(f) \text{ alias } \text{ld}(f)$, where $\text{st}(f)$ and $\text{ld}(f)$ are like a pair of parentheses [48]. For a pair of statements $q.f = y (\text{st}(f))$ and $x = p.f (\text{ld}(f))$, if p and q are aliases, then an object o that flows into y can flow first through this pair of parentheses (i.e. $\text{st}(f)$ and $\text{ld}(f)$) and then into x . For example, in Figure 2.4(b), as $o_{15} \xrightarrow{\text{new}} v1_{\text{main}} \xrightarrow{\text{param}_{15}} \text{this}_{\text{vector}}$ and $o_{15} \xrightarrow{\text{new}} v1_{\text{main}} \xrightarrow{\text{param}_{18}} \text{this}_{\text{get}}$, we have $\text{this}_{\text{vector}} \text{ alias } \text{this}_{\text{get}}$. Thus $o_6 \xrightarrow{\text{new}} t_{\text{vector}} \xrightarrow{\text{st}(\text{elems})} \text{this}_{\text{vector}} \text{ alias } \text{this}_{\text{get}} \xrightarrow{\text{ld}(\text{elems})} t_{\text{get}}$, indicating that o_6 flows to t_{get} .

To allow the *alias* relation in the language, $\overline{\text{flowsTo}}$ is introduced as the inverse of the *flowsTo* relation. For a *flowsTo*-path p , its corresponding $\overline{\text{flowsTo}}$ -path \bar{p} can be obtained using inverse edges, and vice versa. For each edge $x \xrightarrow{e} y$ in p , its inverse edge is $y \xrightarrow{\bar{e}} x$ in \bar{p} . Therefore, $o \text{ flowsTo } x$ iff $x \overline{\text{flowsTo}} o$, indicating that $\overline{\text{flowsTo}}$ actually represents the points-to relation. To find the points-to set of a variable, we use the CFL given in (2.2) with $\overline{\text{flowsTo}}$ as the start symbol.

Context-Sensitivity When context-sensitivity is considered, param_i and ret_i are treated as `assign` as before in L_{FS} . However, assign^l and assign^g are now distinguished.

A context-sensitive CFL requires param_i and ret_i to be matched, also in terms of balanced parentheses, by ruling out the unrealisable paths in a program [45].

The CFL R_{CS} (CS for context-sensitivity) for capturing realisable paths is:

$$\begin{aligned}
 c &\rightarrow \text{entry}_i \ c \ \text{exit}_i \mid c \ c \mid \epsilon \\
 \text{entry}_i &\rightarrow \text{param}_i \mid \overline{\text{ret}_i} \\
 \text{exit}_i &\rightarrow \text{ret}_i \mid \overline{\text{param}_i}
 \end{aligned} \tag{2.3}$$

When traversing along a *flowsTo* path, after entering a method via param_i from call site i , a context-sensitive analysis requires exiting from that method back to call site i , via either ret_i to continue its traversal along the same *flowsTo* path or $\overline{\text{param}_i}$ to start a new search for a $\overline{\text{flowsTo}}$ path. Traversing along a $\overline{\text{flowsTo}}$ path is similar except that the direction of traversal is reversed. Consider Figure 2.4(a). s1_{main} is found to point to o_{16} as o_{16} reaches s1_{main} along a realisable path by first matching param_{17} and $\overline{\text{param}_{17}}$ and then param_{18} and ret_{18} . However, s1_{main} does not point to o_{20} since o_{20} cannot reach s1_{main} along a realisable path.

Let L_{PT} (PT for points-to) be the CFL for computing the points-to information of a variable field- and context-sensitively. Then L_{PT} is defined in terms of (2.2) and (2.3): $L_{PT} = L_{FS} \cap R_{CS}$, where $\overline{\text{flowsTo}}$ is the start symbol.

Algorithm With CFL-reachability, a query that requests for a variable’s points-to information can be answered on-demand, according to Algorithm 2. This algorithm makes use of three variables: (1) E represents the edge set of the PAG of the program, (2) B is the (global) budget defined as the maximum number of steps that can be traversed by any query, with each node traversal being counted as one step [48], and (3) *steps* is query-local, representing the number of steps that has been traversed so far by a particular query.

Given a query (l, c) , where l is a local variable and c is a context, POINTSTO computes the points-to set of l under c . It traverses the PAG with a work list W maintained for variables to be explored. *pts* is initialised with an empty set and W

Algorithm 2 CFL-reachability-based pointer analysis, where POINTSTO computes $\overline{flowsTo}$ and FLOWSTO is analogous to its inverse POINTSTO and thus omitted.

Global E ; **Const** B ; **QueryLocal** $steps$; // initially 0

Procedure POINTSTO(l, c)

begin

```

1  |  $pts \leftarrow \emptyset$ ;
2  |  $W \leftarrow \{<l, c>\}$ ;
3  | while  $W \neq \emptyset$  do
4  |    $<x, c> \leftarrow W.pop()$ ;
5  |    $steps \leftarrow steps + 1$ ;
6  |   if  $steps > B$  then OUTOFBUDGET(0);
7  |   foreach  $x \xleftarrow{new} o \in E$  do  $pts \leftarrow pts \cup \{<o, c>\}$ ;
8  |   foreach  $x \xleftarrow{assign^l} y \in E$  do  $W.push(<y, c>)$ ;
9  |   foreach  $x \xleftarrow{assign^g} y \in E$  do  $W.push(<o, \epsilon>)$ ;
10 |   foreach  $<y, c'> \in \text{REACHABLENODES}(x, c)$  do
11 |      $W.push(<y, c'>)$ ;
12 |   foreach  $x \xleftarrow{param_i} y \in E$  do
13 |     if  $c = \epsilon$  or  $c.top() = i$  then
14 |        $W.push(<y, c.pop()>)$ ; //  $\epsilon.pop() \equiv \epsilon$ 
15 |   foreach  $x \xleftarrow{ret_i} y \in E$  do  $W.push(<y, c.push(i)>)$ ;
16 | return  $pts$ ;

```

Procedure REACHABLENODES(x, c)

begin

```

17 |  $rch \leftarrow \emptyset$ ;
18 | foreach  $x \xleftarrow{ld(f)} p \in E$  do
19 |   foreach  $q \xleftarrow{st(f)} y \in E$  do
20 |      $alias \leftarrow \emptyset$ ;
21 |     foreach  $<o, c'> \in \text{POINTSTO}(p, c)$  do
22 |        $alias \leftarrow alias \cup \text{FLOWSTO}(o, c')$ ;
23 |     foreach  $<q', c''> \in alias$  do
24 |       if  $q' = q$  then  $rch \leftarrow rch \cup \{<y, c''>\}$ ;
25 | return  $rch$ ;

```

Procedure OUTOFBUDGET(BDG)

begin

```

26 |  $\text{exit}()$ ;

```

with $\langle l, c \rangle$ (lines 1 – 2). By default, *steps* for this query is initialised as 0. Each variable x with its context c , i.e., $\langle x, c \rangle$ obtained from W is processed as follows: *steps* is updated, triggering a call to `OUTOFBUDGET` if the remaining budget is 0 (lines 5 – 6), and the incoming edges of x are traversed according to (2.2) and (2.3) (lines 7 – 15).

Field-sensitivity is handled by `REACHABLENODES(x, c)`, which searches for the reachable variables y to x in context c , due to heap accesses by matching the load ($x = p.f$) with every store ($q.f = y$), where p and q are aliases (lines 17 – 25). Both `POINTSTO` and `FLOWSTO` are called (recursively) to ensure that p and q are aliased base variables.

To handle context-sensitivity, the analysis stays in the same context c for `assign`^{*l*} (line 8), clears c for `assign`^{*g*} as global variables are treated context-insensitively (line 9), matches the context ($c.top() = i$) for `param`_{*i*} but allows for partially balanced parentheses when $c = \epsilon$ since a realizable path may not start and end in the same method (lines 12 – 14), and pushes call site i into context c for `ret`_{*i*} (line 15).

Chapter 3

Demand-Driven Analysis on Multi-CPU

In this chapter, we describe the first parallel implementation of demand-driven pointer analysis with CFL-reachability. We explore query level parallelism in the analysis and further accelerate it by reducing redundant graph traversals via a data sharing scheme and a query scheduling scheme. Our solution was previously proposed in [50]. Section 3.1 describes our parallel solution in terms of the data sharing and query scheduling schemes used. Section 3.2 evaluates our solution. Section 3.3 discusses the related work on demand-driven pointer analysis. Section 3.4 summarises this chapter.

3.1 Methodology

CFL-reachability-based pointer analysis is driven by queries issued by application clients. There are two main approaches to dividing work among threads, based on different levels of parallelism available: *intra-query* and *inter-query*.

To exploit intra-query parallelism, we need to partition and distribute the work

performed in computing the points-to set of a single query among different threads. Such parallelism is irregular and hard to achieve with the right granularity. In addition, considerable synchronisation overhead that may be incurred would likely offset the performance benefit achieved.

To exploit inter-query parallelism, we assign different queries to different threads, harnessing modern multicore processors. This makes it possible to obtain parallelism without incurring synchronisation overhead unduly. In addition, some clients may issue queries in batch mode for a program. For example, the points-to information may be requested for all variables in a method, a class, a package or even the entire program. This provides a further optimisation opportunity. The focus of this chapter is on exploiting inter-query parallelism.

3.1.1 A Naive Parallelisation Strategy

A naive approach to exploiting inter-query parallelism is to maintain a lock-protected shared work list for queries and let each thread fetch queries (to process) from the work list until the work list is empty. While achieving some good speedups (over the sequential setting), this naive strategy is inefficient due to a large number of redundant graph traversals made. We propose two schemes to reduce such redundancies. Section 3.1.2 describes our data sharing scheme, while Section 3.1.3 explains our query scheduling scheme.

3.1.2 Data Sharing

Given a program, we are motivated to add edges to its PAG to serve as shortcuts for some paths traversed in a query so that subsequent queries may take the shortcuts instead of re-traversing their associated paths (redundantly). The challenge here is to perform data sharing context- and field-sensitively. We first formulate data

sharing in terms of graph rewriting, and then give an algorithm for realising data sharing in the CFL-reachability framework.

Data Sharing by Graph Rewriting We choose to share paths involving heap accesses, which tend to be long (time-consuming to traverse) and common (repeatedly traversed across the queries). As illustrated in Figure 2.4, we do so by avoiding making redundant alias tests in `REACHABLENODES`(x, c). For its loop at line 18, each iteration starts with a load $x = p.f$ and then examines all the N matching stores $q_1.f = y_1, \dots, q_N.f = y_N$ at line 19. For each $q_k.f = y_k$ accessed in context c_k such that q_k is an alias of p , (y_k, c_k) is inserted into rch , meaning that (x, c) is reachable from (y_k, c_k) (lines 20 - 24). Note that during this process, mutually recursive calls to `POINTSTO`(), `FLOWSTO`() and `REACHABLENODES`() for discovering other aliases are often made.

There are two cases due to the budget constraint. Figure 3.1(a) illustrates the case when an iteration of line 18 is completely analysed in s steps starting from (x, c) within the pre-set budget. A **jmp** edge, $x \xrightarrow[\langle c, c_k \rangle]{\text{jmp}(s)} y_k$, is added for each q_k that is an alias of p . Instead of rediscovering the path from (x, c) to (y_k, c_k) , a subsequent query will take this shortcut.

Figure 3.1(b) explains the other case when an iteration of line 18 is only partially analysed since the analysis runs out of budget after s steps have elapsed from (x, c) . A special **jmp** edge, $x \xrightarrow[\langle c, \epsilon \rangle]{\text{jmp}(s)} \mathcal{O}$, is added to record this situation, where \mathcal{O} is a special node added and ϵ is a “don’t-care” context. A later query will benefit from this special shortcut by making an *early termination (ET)* if its remaining budget is smaller than s .

Therefore, we have formulated data sharing as a graph rewriting problem by adding **jmp** edges to the PAG of a program, in terms of the syntax given in Figure 3.2.

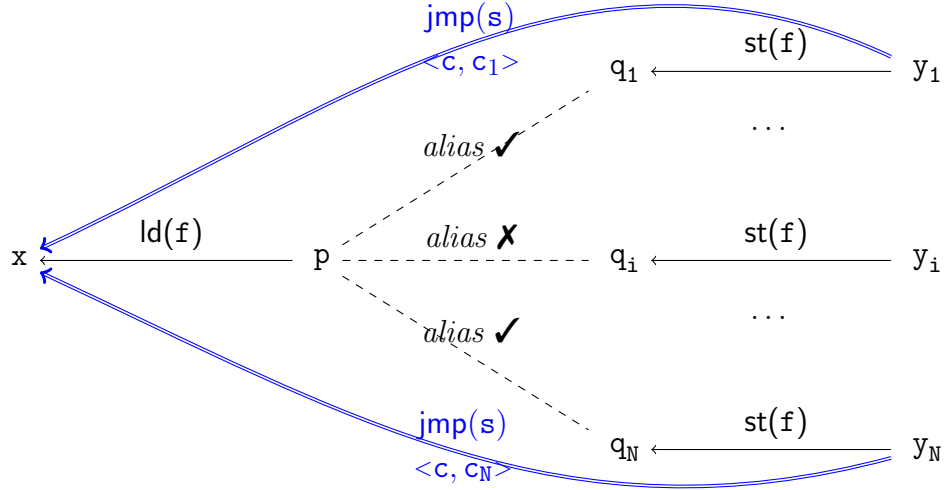
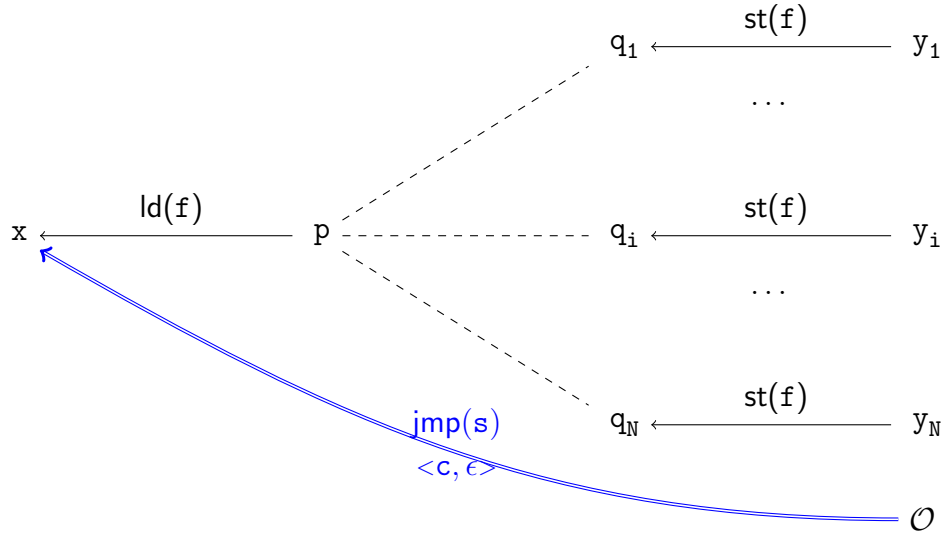
(a) Within budget: all N stores analysed completely in s steps from (x, c) (b) Out of budget: fewer than N stores analysed in s steps from (x, c)

Figure 3.1: Adding **jmp** edges by graph rewriting, for a single iteration of the loop in line 18 of `REACHABLENODES`(x, c). In (a), $x \xleftarrow[\langle c, c_k \rangle]{\text{jmp}(s)} y_k$ is introduced for each (y_k, c_k) added to rch in line 24 of `REACHABLENODES`(x, c) when p and q_k are aliases. In (b), a special $x \xleftarrow[\langle c, \epsilon \rangle]{\text{jmp}(s)} \mathcal{O}$ edge is introduced.

$$\begin{aligned}
l &:= \dots \mid \mathcal{O} && \text{Extended Local Variable} \\
e &:= \dots \\
&\mid l_1 \xrightarrow[\langle c_1, c_2 \rangle]{\text{jmp}(s)} l_2 && \text{Jump (or Shortcut)} \\
\mathcal{O} &\text{ is Unfinished} && c_1, c_2 \in \text{Context}
\end{aligned}$$

Figure 3.2: Syntax of extended PAG.

As described below, **jmp** edges are added on the fly during the analysis. Given a PAG extended with such **jmp** edges, the CFL given earlier in (2.2) is modified to:

$$\begin{aligned}
\text{flowsTo} &\rightarrow \text{new (assign } \mid \text{ jmp}(s) \mid \text{ st}(f) \text{ alias ld}(f))^* \\
\text{alias} &\rightarrow \overline{\text{flowsTo}} \text{ flowsTo} \\
\overline{\text{flowsTo}} &\rightarrow (\overline{\text{assign}} \mid \overline{\text{jmp}(s)} \mid \overline{\text{ld}(f)} \text{ alias } \overline{\text{st}(f)})^* \overline{\text{new}}
\end{aligned} \tag{3.1}$$

By definition of **jmp**, this modified CFL generates the same language as the original CFL if all **jmp** edges of the type illustrated in Figure 3.1(b) (for handling **OUTOFBUDGET**) in the PAG of a program are ignored, since the **jmp** edges of the other type illustrated in Figure 3.1(a) serve as shortcuts only. Two types of **jmp** edges are exploited in our parallel implementation to accelerate its performance as described below.

Algorithm With data sharing, **REACHABLENODES**(x, c) in Algorithm 2 is revised as shown in Algorithm 3. There are three cases, the original one plus the two shown in Figure 3.1:

- In the **if** branch (line 2) for handling the scenario depicted in Figure 3.1(b), the analysis makes an early termination by calling **OUTOFBUDGET**() if its remaining budget at (x, c) , $B - \text{steps}$, is smaller than s . Otherwise, the analysis moves to execute the second **else**.

Algorithm 3 REACHABLENODES with data sharing.

Global E ; **Const** B ; **QueryLocal** $steps, S$;

Procedure REACHABLENODES(x, c)

begin

```

1   $rch \leftarrow \emptyset$ ;
2  if  $\exists x \xrightarrow[\langle c, \epsilon \rangle]{\text{jmp}(s)} \mathcal{O} \in E$  then
3    if  $B - steps < s$  then OUTOFBUDGET( $s$ );
4  else if  $\exists x \xrightarrow[\langle c, c' \rangle]{\text{jmp}(s)} y \in E$  then
5     $steps \leftarrow steps + s$ ;
6    foreach  $x \xrightarrow[\langle c, c' \rangle]{\text{jmp}(s)} y \in E$  do
7       $rch \leftarrow rch \cup \{\langle y, c' \rangle\}$ ;
8    return  $rch$ ;
9  else
10    $s' = steps$ ;
11    $S \leftarrow S \cup \{\langle x, c, s' \rangle\}$ ;
12   foreach  $x \xrightarrow{\text{ld}(f)} p \in E$  do
13     foreach  $q \xrightarrow{\text{st}(f)} y \in E$  do
14        $alias \leftarrow \emptyset$ ;
15       foreach  $\langle o, c' \rangle \in \text{POINTSTO}(p, c)$  do
16          $alias \leftarrow alias \cup \text{FLOWSTO}(o, c')$ ;
17       foreach  $\langle q', c'' \rangle \in alias$  do
18         if  $q' = q$  then
19            $rch \leftarrow rch \cup \{\langle y, c'' \rangle\}$ ;
20            $E \leftarrow E \cup \{x \xrightarrow[\langle c, c'' \rangle]{\text{jmp}(steps-s')} y\}$ ;
21    $S \leftarrow S \setminus \{\langle x, c, s' \rangle\}$ ;
22   return  $rch$ ;

```

Procedure OUTOFBUDGET(BDG)

begin

```

23  foreach  $\langle x, c, s \rangle \in S$  do
24     $E \leftarrow E \cup \{x \xrightarrow[\langle c, \epsilon \rangle]{\text{jmp}(\min(B, BDG + steps - s))} \mathcal{O}\}$ ;
25  exit();

```

- In the first **else** branch for handling the scenario in Figure 3.1(a), the analysis takes the shortcuts identified by the **jmp**(s) edges instead of re-traversing its associated paths. The same precision is maintained even if we do not check the remaining budget $B - steps$ against s , since the source node of a **jmp** edge is a variable (not an object). When this variable is explored later, the remaining budget will be checked in line 6 of Algorithm 2 or line 3 of Algorithm 3.
- In the second **else** branch, we proceed as in **REACHABLENODES**(x, c) given in Algorithm 2 except that we will need to add the **jmp** edge(s) as illustrated in either Figure 3.1(a) (line 20) or Figure 3.1(b) (line 24).

OUTOFBUDGET(BDG) is called from line 6 (by passing 0) in Algorithm 2 or line 3 in Algorithm 3 (by passing s). In both cases, let n be the node visited before the call. With a remaining budget no larger than BDG on encountering n , the analysis will surely run out of budget eventually. For each $(x, c, s) \in S$, the analysis first reaches (x, c) and then n in $steps - s$ steps. Thus, $x \xrightarrow[\langle c, \epsilon \rangle]{\text{jmp}(\min(B, BDG + steps - s))} \mathcal{O}$ is added.

3.1.3 Query Scheduling

The order in which queries are processed affects the number of early terminations made, due to $B - steps < s$ tested in line 3 of Algorithm 3, where s appears in a **jmp**(s) edge that was added in an earlier query and $steps$ is the number of steps already consumed by the current query. In general, if we handle a variable y before those variables x such that x is reachable from y , then more early terminations may result.

To increase early terminations, we organise queries (available in batch mode)

in groups and assign a group of queries rather than a single query to a thread at a time to reduce synchronisation overhead on the shared work list for queries. We discuss below how the queries in a group and the groups themselves are scheduled. We first describe how to group queries. Then we discuss how to order queries. Lastly, we give an illustrating example.

Grouping Queries A group contains all possible variables such that every member is connected with at least another member in the PAG of the program via the following relation:

$$direct \rightarrow (\text{assign}^l \mid \text{assign}^g \mid \text{param}_i \mid \text{ret}_i)^* \quad (3.2)$$

Both $l_1 \xleftarrow{\text{ld}(f)} l_2$ and $l_1 \xleftarrow{\text{st}(f)} l_2$ edges are not included since there is no reachability between l_1 and l_2 .

Ordering Queries For the variables in the same group, we use their so-called *connection distances* (*CDs*) to determine their issuing order. The *CD* of a variable in a group is defined as the length of the longest path that contains the variable in the group (modulo recursion). For the variables in a group, the shorter their *CDs* are, the earlier they are processed.

For different groups, we use their so-called *dependence depths* (*DDs*) to determine their scheduling order. For example, computing $\text{POINTSTO}(x, c)$ for x in Algorithm 2 depends on the points-to set of the base variable p in load $x = p.f$ (line 21). Preferably, p should be processed earlier than x .

To quantify the *DD* of a group, we estimate the dependences between variables based on their (static) types. We define the *level* of a type t (with respect to its

containment hierarchy) as:

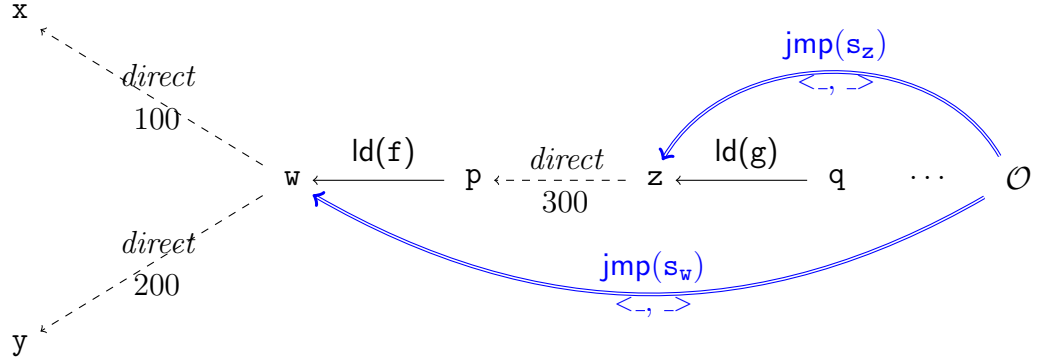
$$L(t) = \begin{cases} \max_{t_i \in F_T(t)} L(t_i) + 1 & \text{isRef}(t) \\ 0 & \text{otherwise} \end{cases}$$

where $F_T(t)$ enumerates the types of all instance fields of t (modulo recursion) and $\text{isRef}(t)$ is true if t is a reference type. The DD of a variable of type t is defined to be $1/L(t)$. Note that the DD of a static variable is also approximated heuristically this way. The DD of a group of variables is defined as the smallest of the DD s of all variables in the group.

During the analysis, groups are issued (sorted) in increasing values of their DD s. Let M be the average size of these groups. To ensure load balance, groups larger than M are split and groups smaller than M are merged with their adjacent groups, so that each resulting group has roughly M variables.

An Example In Figure 3.3, we focus on its three variables x , y , and z , which are assumed to all run out of budget B . According to (3.2), as shown in Figure 3.3(a), x and y (together with w) are in one group and z (together with p) is in another group. The CD s of x , y and z are 100, 200 and 300 steps, respectively. As both x and y depend on z , the latter group will be scheduled before the former group. As a result, our query scheduling scheme will likely cause x , y and z to be processed sequentially according to O_3 (in some thread interleaving) among the three orders, O_1 , O_2 and O_3 , listed in Figure 3.3(b).

For O_1 , y is processed first, which takes B steps (i.e., the maximum budget allowed), with the two **jmp** edges added as shown, where $s_z = B - 500$ and $s_w = B - 200$. When x is processed next, neither shortcut will be taken, since x still has more budget remaining: $B - 400 > B - 500$ at z and $B - 100 > B - 200$ at w . Similarly, the two shortcuts do not benefit z either. Thus, no early termination

(a) PAG with the *direct* relation

Order	Traversed #Steps			jmp(s)		#ETs
	x	y	z	s_z	s_w	
$O_1 : y, x, z$	B	B	B	$B - 500$	$B - 200$	0
$O_2 : x, y, z$	B	200	B	$B - 400$	$B - 100$	1
$O_3 : z, x, y$	400	200	B	B	B	2

(b) Three scheduling orders

Figure 3.3: An example of query scheduling, where x has a smaller CD than y and $\{x, y\}$ has a higher DD than $\{z\}$.

occurs.

For O_2 , x is issued first, resulting in also the same two **jmp** edges added, except that $s_z = B - 400$ and $s_w = B - 100$. So when y is handled next, an early termination is made at w , since its budget remaining at w is $B - 200$ ($< s_w = B - 100$).

According to O_3 , the order that is mostly likely induced by our query scheduling scheme, z is processed first. Only the **jmp**(s_z) edge at z is added, where $s_z = B$. When x is analysed next, z is reached in 400 steps. Taking **jmp**(s_z) (since $B - 400 < s_z = B$), an early termination is made. Meanwhile, the **jmp**(s_w) edge at w is added, where $s_w = B$. Finally, y is issued, causing w to be visited in 200 steps. Taking **jmp**(s_w) (since $B - 200 < s_w = B$), another early termination is made. Of the three orders illustrated in Figure 3.3(b), O_3 is likely to cause more early terminations,

resulting in fewer traversal steps.

Discussion The optimal order for scheduling the queries is hard to compute, if possible, before the points-to information for all variables is gathered. Without points-to information, the time complexity of computing the optimal order is also difficult to estimate. Therefore, it is a simple yet effective solution to employ heuristics for query ordering so that some redundant graph traversals can be eliminated.

3.2 Evaluation

We demonstrate that our parallel implementation of CFL-reachability-based pointer analysis achieves significant speedups than a state-of-the-art sequential implementation.

3.2.1 Implementations

The sequential one is coded in Java based on the publicly available source-code implementation of the CFL-reachability-based pointer analysis [45] in Soot 2.5.0 [58], with its non-refinement (general-purpose) configuration used. Note that the refinement-based configuration is not well-suited to certain clients such as null-pointer detection. Our parallel implementation given in Algorithms 2 and 3 are also coded in Java. In both cases, the per-query budget B is set as 75,000 steps, recursion cycles of the call graph are collapsed, and points-to cycles are eliminated as described as in [45].

In our parallel implementation, we use a `ConcurrentHashMap` to manage `jmp` edges efficiently. We apply a simple optimisation to further reduce synchronisation incurred and thus achieve better speedups.

If we create **jmp** edges exhaustively for all the paths discovered in Algorithm 3, the overhead incurred by such operations as search, insertion and synchronisation on the map may outweigh the performance benefit obtained. As an optimisation, we will introduce the **jmp**(s) edges in Figure 3.1(a) only when $s \geq \tau_F$ and the special **jmp**(s) edge in Figure 3.1(b) only when $s \geq \tau_U$, where τ_F and τ_U are tunable parameters. In our experiments, we set $\tau_F = 100$ and $\tau_U = 10000$. Their performance impacts are evaluated in Section 3.2.4.

For the case in Figure 3.1(a), the set of all **jmp** edges is associated with the key (x, c) when inserted into the map. So no two threads reaching (x, c) simultaneously will insert this set of **jmp** edges twice into the map. For the case in Figure 3.1(b), if one thread tries to insert $\langle (x, c), x \xleftarrow[\langle c, \epsilon \rangle]{\text{jmp}(s_1)} \mathcal{O} \rangle$ and another tries to insert $\langle (x, c), x \xleftarrow[\langle c, \epsilon \rangle]{\text{jmp}(s_2)} \mathcal{O} \rangle$ into the map, only one of the two will succeed. An attempt that selects the one with the large s value (to increase early terminations) can be cost-ineffective due to the extra work performed.

3.2.2 Experimental Settings

The multi-core system used in our experiments is equipped with two Intel Xeon E5-2650 CPUs with 62GB of RAM. Each CPU has 8 cores, which share a unified 20MB L3 cache. Each CPU core has a frequency of 2.00GHz, with its own L1 cache of 64KB and L2 cache of 256KB. The Java Virtual Machine used is the Java HotSpot 64-Bit Server VM (version 1.7.0_40), running on a 64-bit Ubuntu 12.04 operating system.

3.2.3 Methodology

We evaluate the performance advantages of our parallel implementation over the sequential one by comparing the query-processing times taken in both cases. SEQCFL

denotes the sequential implementation. In order to assess the effectiveness of our parallel implementation, we consider a number of variations. PARCFL_m^t represents a particular parallel implementation, where t stands for the number of threads used. Here, m indicates one of the three parallelisation strategies used: (1) the naive solution described in Section 3.1.1 when $m = \text{naive}$, (2) our parallel solution with the data sharing scheme described in Section 3.1.2 enabled when $m = D$, and (3) the parallel solution (2) with the query scheduling scheme described in Section 3.1.3 also enabled when $m = DQ$.

Benchmark	#Classes	#Methods	#Nodes	#Edges	#Queries
_200_check	5,758	54,514	225,797	429,551	1,101
_201_compress	5,761	54,549	225,765	429,808	1,328
_202_jess	5,901	55,200	232,242	440,890	7,573
_205_raytrace	5,774	54,681	227,514	432,110	3,240
_209_db	5,753	54,549	225,994	430,569	1,339
_213_javac	5,921	55,685	240,406	473,680	14,689
_222_mpegaudio	5,801	54,826	230,349	435,391	6,389
_227_mtrt	5,774	54,681	227,514	432,110	3,241
_228_jack	5,806	54,830	229,482	435,159	6,591
_999_checkit	5,757	54,548	226,292	431,435	1,473
avroa	3,521	29,542	108,210	189,081	24,455
batik	7,546	65,899	252,590	477,113	64,467
fop	8,965	79,776	266,514	636,776	71,542
h2	3,381	32,691	115,249	204,516	44,901
lindex	3,160	28,791	108,827	191,126	22,415
lsearch	3,120	28,223	109,439	193,012	17,520
pmd	3,786	33,432	110,388	195,834	56,833
sunflow	6,066	56,673	233,459	447,002	21,339
tomcat	8,458	83,092	265,015	574,236	185,810
xalan	3,716	33,248	109,317	192,441	56,229
Average	5,486	50,972	198,518	383,592	30,624

Table 3.1: Benchmark information.

Table 3.1 lists a set of 20 Java benchmarks used, consisting of all the 10 SPEC JVM98 benchmarks and 10 additional benchmarks from the DaCapo 2009 benchmark suite. For each benchmark, the analysed code includes both the application code and the related library code, with their class and method counts given in Columns 2 and 3, respectively. The node and edge counts in the original PAG of a benchmark are given in Columns 4 and 5, respectively. For each benchmark, the queries that request points-to information are issued for all the local variables in its application code, collected from Soot 2.5.0 as in prior work [43, 64]. Note that more queries are generated in some DaCapo benchmarks than some JVM98 benchmarks even though the DaCapo benchmarks have smaller PAGs. This is because the JVM98 benchmarks involve more library code.

3.2.4 Performance Results

We examine the performance benefits of our parallel pointer analysis and the causes for the speedups obtained.

Speedups Figure 3.4 shows the speedups of our parallel implementation over SEQCFL (as the baseline), where the analysis times of SEQCFL for all the benchmarks are given in Column 7 of Table 3.1. Note that SEQCFL is 49% faster than the open-source sequential implementation of [45] in Soot 2.5.0, since we have simplified some of its heuristics and employed different data structures. When the naive parallelisation strategy is used, PARCFL_{naive}^1 (with one single thread) is as efficient as SEQCFL, since the locking overhead incurred for retrieving the queries from the shared work list is negligible. With 16 threads, $\text{PARCFL}_{naive}^{16}$ attains an average speedup of 7.3X. When our data sharing scheme is used, PARCFL_D^{16} runs a lot faster, with the average speedup being pushed up further to 13.4X. When our

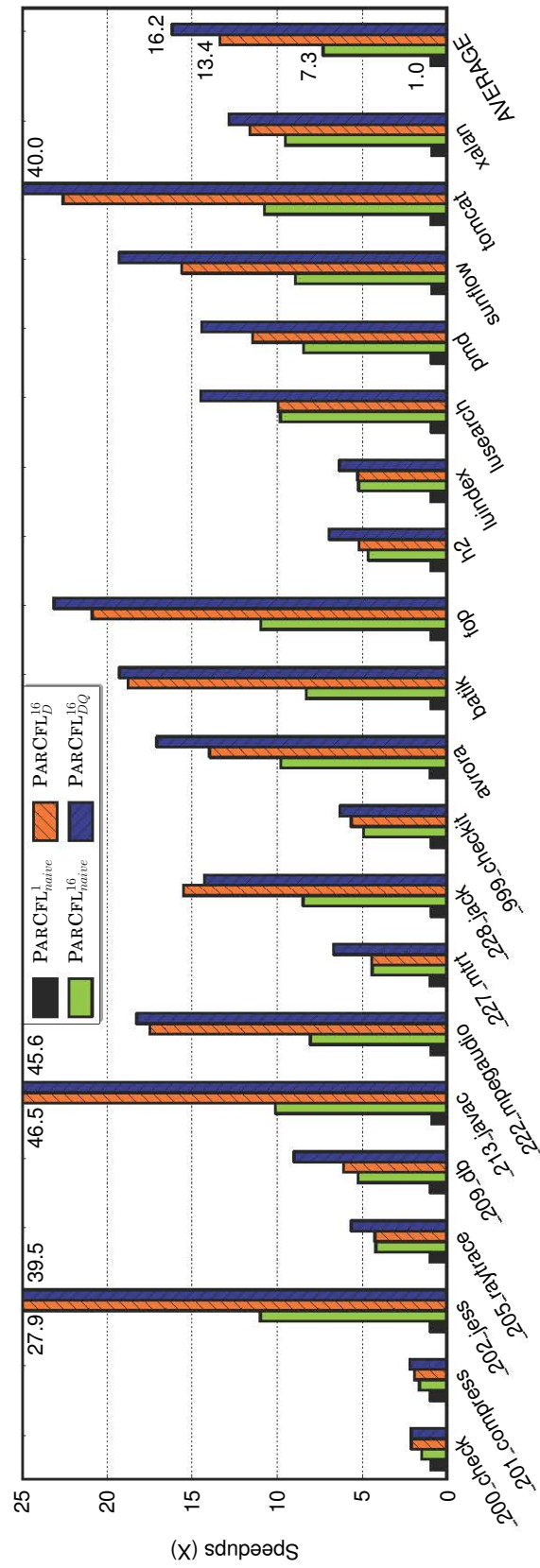


Figure 3.4: Speedups of our parallel implementation (in various configurations) normalised with respect to SeqCFL.

query scheduling scheme is also enabled, PARCFL_{DQ}^{16} , which traverses significantly fewer steps than SEQCFL , has finally reached an average speedup of 16.2X. The superlinear speedups are achieved in some benchmarks due to the avoidance of redundant traversals (a form of caching) in all concurrent query-processing threads as analysed below.

Effectiveness of Data Sharing Our data sharing scheme, which enables the traversal information obtained in a query to be shared by subsequent queries via graph rewriting, has succeeded in accelerating the analysis on top of the naive parallelisation strategy (PARCFL_{naive}^t) for all benchmarks. Our data sharing scheme, which enables the traversal information obtained in a query to be shared by subsequent queries via graph rewriting, has succeeded in accelerating the analysis on top of the naive parallelisation strategy (PARCFL_{naive}^t) for all benchmarks.

To understand its effectiveness, some statistics are given in Table 3.2. For a benchmark, $\#Jumps$ denotes the number of **jmp** edges added to its PAG due to data sharing, $\#S$ represents the total number of steps traversed by SEQCFL (without data sharing) for all the queries issued from the benchmark, and R_S is the ratio of the number of steps saved by the **jmp** edges for the benchmark over the number of steps traversed across the original edges (when data sharing is enabled). For the 20 benchmarks used, 22,023 **jmp** edges have been added on average per benchmark. The number of steps saved by these **jmp** edges is much larger than that of the original ones, by a factor of 28.6X on average. This implies that a large number of redundant traversals ($\#S \times \frac{R_S}{R_S+1}$ for a benchmark) have been eliminated. Thus, PARCFL_D^{16} exhibits substantial improvements over $\text{PARCFL}_{naive}^{16}$, with the superlinear speedups observed in `_202_jess`, `_213_javac`, `_222_mpegaudio`, `batik`, `fop` and `tomcat`.

The optimisation described in Section 3.2.1 for adding **jmp** edges selectively

Benchmark	T_{Seq} (secs)	#Jumps	$\#S (\times 10^6)$	R_S
_200_check	2.88	428	4.14	25.76
_201_compress	3.72	1,210	4.21	8.42
_202_jess	121.11	4,755	193.77	42.68
_205_raytrace	9.39	2,325	62.02	92.84
_209_db	16.98	4,202	10.06	10.02
_213_javac	258.34	5,309	467.28	64.60
_222_mpegaudio	46.52	2,306	86.17	53.33
_227_mtrt	10.38	2,358	62.17	115.70
_228_jack	39.54	25,030	79.48	40.03
_999_checkit	12.61	2,180	10.14	7.94
avroa	51.16	32,046	47.46	6.18
batik	72.72	14,876	114.57	11.95
fop	118.22	25,418	169.92	19.03
h2	25.50	22,094	91.38	12.39
luindex	23.28	62,457	60.93	8.72
lusearch	57.78	77,153	66.26	7.90
pmd	61.05	77,313	69.10	7.93
sunflow	55.56	20,946	49.04	5.57
tomcat	202.89	24,601	243.90	23.14
xalan	54.11	33,459	60.35	7.90
Average	62.19	22,023	97.62	28.6

Table 3.2: Statistics for data sharing.

to reduce synchronisation overhead is also useful for improving the performance. Figure 3.5 reveals the histograms of added **jmp** edges with and without this optimisation. In the absence of such optimisation, many **jmp** edges representing relatively short paths are also added, causing PARCFL_{DQ}^{16} to drop from 16.2X to 12.4X on average.

Effectiveness of Query Scheduling When query scheduling is also enabled, queries are grouped and reordered to increase early terminations made. PARCFL_{DQ}^{16}

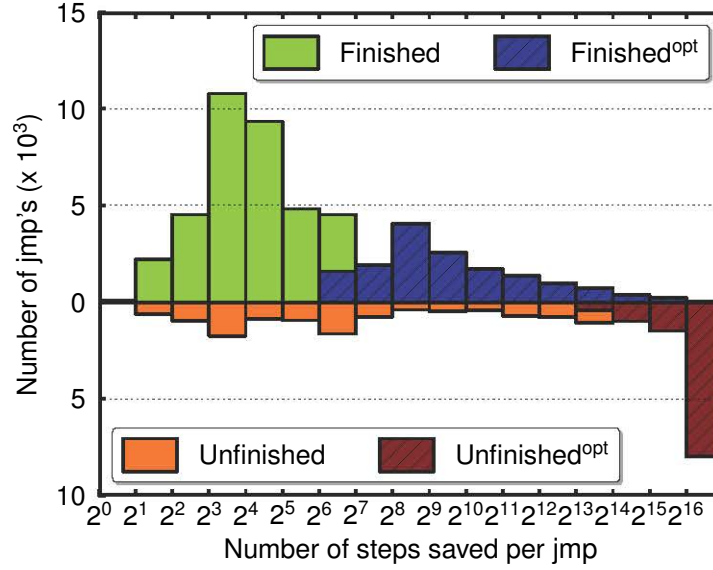


Figure 3.5: Histograms of `jmp` edges (identified by the number of steps saved). `Finished` represents `jmp` edges in Figure 3.1(a) and `Unfinished` `jmp` edges in Figure 3.1(b). `Finishedopt` (`Unfinishedopt`) is the version of `Finished` (`Unfinished`) with the selective optimisation described in Section 3.2.1 being applied.

achieves superlinear speedups in two more benchmarks than PARCFL_D^{16} : `avroa` and `sunflow`. PARCFL_{DQ}^{16} is faster than PARCFL_D^{16} as the average speedup goes up from 13.4X to 16.2X.

To understand its effectiveness, some statistics are given in Table 3.3. For a benchmark, $\overline{S_g}$ gives the average number of queries in a group, $\#ETs$ is the number of early terminations found without query scheduling, and R_{ET} is the ratio of $\#ETs$ obtained with query scheduling over $\#ETs$ obtained without query scheduling. On average, our query scheduling scheme leads to 35% more early terminations, resulting in more redundant traversals being eliminated.

Scalability To see the scalability of our parallel implementation, Figure 3.6 plots its speedups with a few thread counts over the baseline. PARCFL_{DQ}^1 achieves an average speedup of 8.1X, due to data sharing and query scheduling. Our parallel solution scales well to 8 threads for most benchmarks. When moving from 8 to

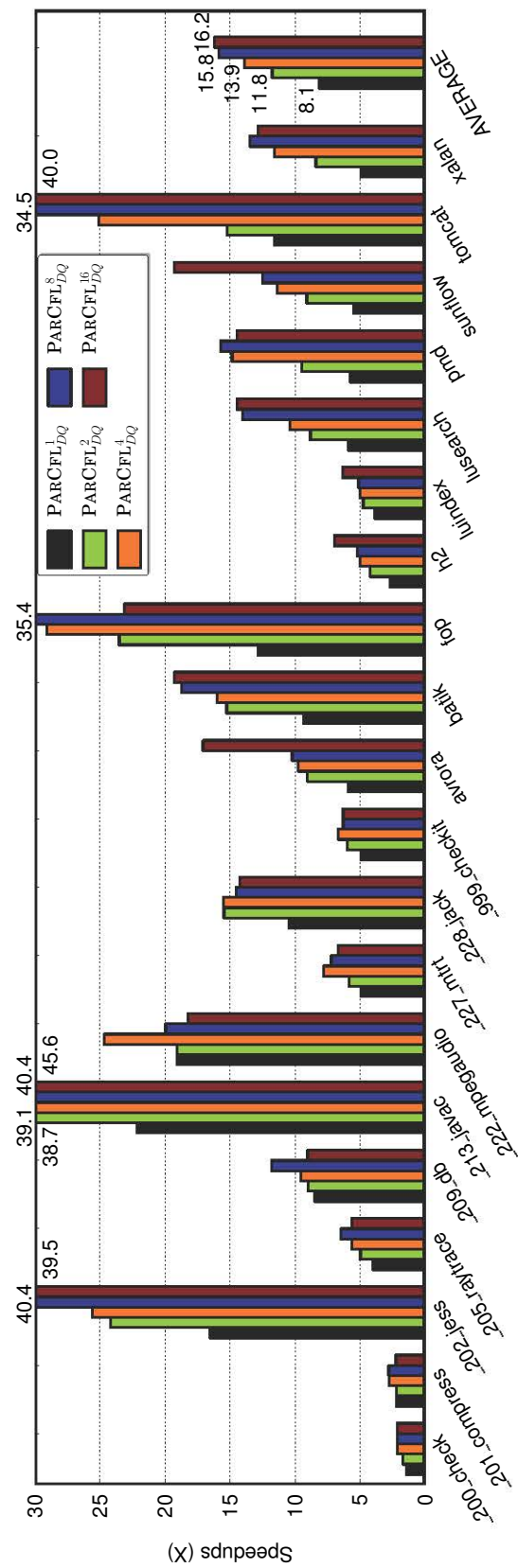


Figure 3.6: Speedups of our parallel modes with different numbers of threads normalised with respect to SEQCFL.

Benchmark	$\overline{S_g}$	$\#ETs$	R_{ET}
_200_check	16.7	0	1
_201_compress	4.6	5	1.00
_202_jess	16.1	617	1.15
_205_raytrace	7.2	8	0.88
_209_db	10.3	18	1.17
_213_javac	9.2	76	0.99
_222_mpegaudio	3.8	53	3.17
_227_mtrt	7.2	7	0.86
_228_jack	14.2	100	1.62
_999_checkit	16.9	23	0.78
avroa	9.4	24	2.83
batik	10.3	38	1.37
fop	18.6	76	1.20
h2	16.0	283	0.66
luindex	8.2	113	0.71
lusearch	9.3	75	1.52
pmd	9.2	84	1.06
sunflow	7.4	24	2.38
tomcat	13.1	574	1.33
xalan	9.4	82	1.43
Average	10.9	114.0	1.35

Table 3.3: Statistics for query scheduling.

16 threads, PARCFL_{DQ}^{16} suffers some performance drops over PARCFL_{DQ}^8 in some benchmarks (with `_209_db` being the worst case at 31%). However PARCFL_{DQ}^{16} is still slightly faster than PARCFL_{DQ}^8 on average.

Memory Usage As garbage collection is enabled, it is difficult to monitor memory usage precisely. By avoiding redundant graph traversals, PARCFL_{DQ}^{16} reduces the memory usage by SEQCFL (the open-source sequential implementation [45]) by 35% (32%) in terms of the peak memory usage, despite the extra memory required

for storing `jmp` edges. In the worst case attained at `tomcat (fop)`, PARCFL_{DQ}^{16} consumes 103% (118%) of the memory consumed by SEQCFL ([45]).

3.3 Related Work

While no parallel solutions to CFL-reachability-based pointer analysis have been proposed before, there is no shortage of optimisations in the sequential setting. To ensure quick response, queries are commonly processed under budget constraints [43, 45, 48, 65, 71]. In addition, refinement-based schemes [45, 48] can be effective for certain clients, e.g., type casting if field-sensitivity is gradually introduced. Summary-based schemes avoid redundant graph traversals by reusing the method-local points-to relations summarised statically [65] or on-demand [43], achieving up to 3X speedups. Must-not-alias information obtained during a pre-analysis can be exploited to yield an average speedup of 3X through reducing unnecessary alias-related computations [64]. Incremental techniques [26, 42], which are tailored for scenarios where code changes are small, take advantage of previously computed CFL-reachable paths to avoid unnecessary reanalysis.

Unlike these efforts on sequential CFL-reachability-based pointer analysis, this chapter introduces the first parallel solution on multicore processors with significantly better speedups.

3.4 Chapter Summary

This chapter presents the first parallel implementation of CFL-reachability-based pointer analysis on multi-core CPUs. Despite the presence of redundant graph traversals, this demand-driven analysis is non-trivial to parallelise due to the dependences introduced by context- and field-sensitivity during graph traversals. We

have succeeded in parallelising it by using (1) a data sharing scheme that enables the concurrent query-processing threads to avoid traversing earlier discovered paths via graph rewriting and (2) a query scheduling scheme that allows more redundancies to be eliminated based on the dependences statically estimated among the queries to be processed. For a set of 20 Java benchmarks evaluated, our parallel implementation significantly boosts the performance of a state-of-the-art sequential implementation with an average speedup of 16.2X on 16 CPU cores.

Chapter 4

Whole-Program Analysis on GPU

This chapter describes an efficient GPU solution to Andersen’s whole-program pointer analysis. This solution leverages the state-of-the-art GPU implementation [28] and also improves it by solving its load imbalance issue, via an imbalance-aware workload partitioning scheme. We also improve its performance further by using an adaptive group propagation scheme to reduce unnecessary graph traversals. Our solution was previously introduced in [51]. Section 4.1 examines the inherent irregularity of graph workloads in Andersen’s analysis on GPUs and motivates our solution. The details of our imbalance-aware workload partitioning and adaptive group propagation schemes will be given in Section 4.2. Section 4.3 evaluates and discusses our solution. In Section 4.4, the related work on Andersen’s analysis is reviewed. Finally, Section 4.5 contains the conclusions of this chapter.

4.1 Challenges and Motivation

In this section, we use `vim`, one of the benchmarks used in our evaluation, as an example to highlight (1) highly irregular graph workloads in Andersen’s analysis and (2) redundant graph traversals that are both present even in a state-of-the-

art GPU implementation [28]. Figure 4.1 depicts the execution times of all its iterations, which are difficult to predict statically due to severe fluctuations.

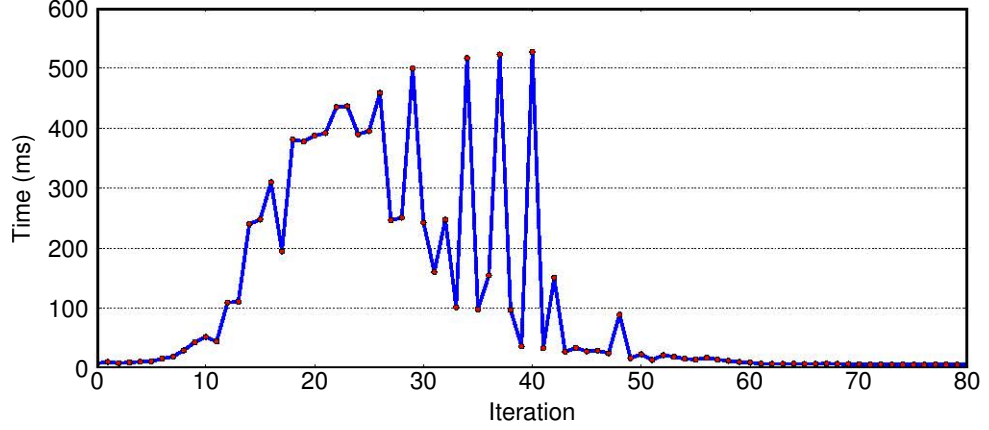


Figure 4.1: Execution times of all iterations in `vim` (with the 0^{th} for the initialisation phase).

4.1.1 Load Imbalance

Figure 4.2 plots the points-to edges for `vim`, with all the \mathcal{P} edges obtained up to iteration 5 depicted in blue and the new $\delta\mathcal{P}$ edges added in iteration 6 depicted in red. Figure 4.3 shows how the outgoing degrees, non-zero $|E_{\mathcal{P}}(x)|$ and $|E_{\mathcal{C}}(x)|$, for the \mathcal{P} and \mathcal{C} edges, respectively, are distributed across the nodes x at the end of the analysis. For example, some nodes can have over 1,000 outgoing edges while others have only a few.

We can observe that the distribution of edges in a graph is highly skewed and also changes dynamically, leading to a highly imbalanced workload distribution across its nodes. As a result, the warps processing low-degree nodes will have to wait for those processing high-degree nodes, causing the computational resources in a GPU to be underutilised. It is difficult but beneficial to address this inter-warp imbalance problem.

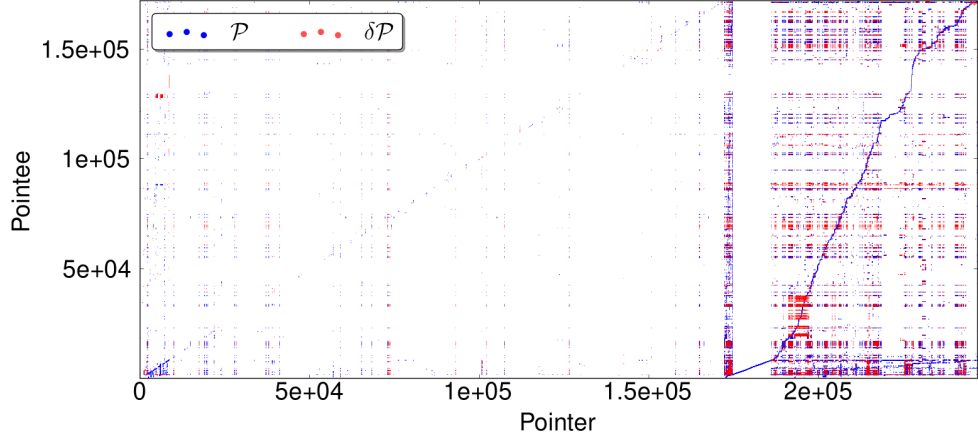


Figure 4.2: The points-to edges for `vim`, with the \mathcal{P} edges obtained up to iteration 5 shown in blue and the new $\delta\mathcal{P}$ edges added in iteration 6 shown in red.

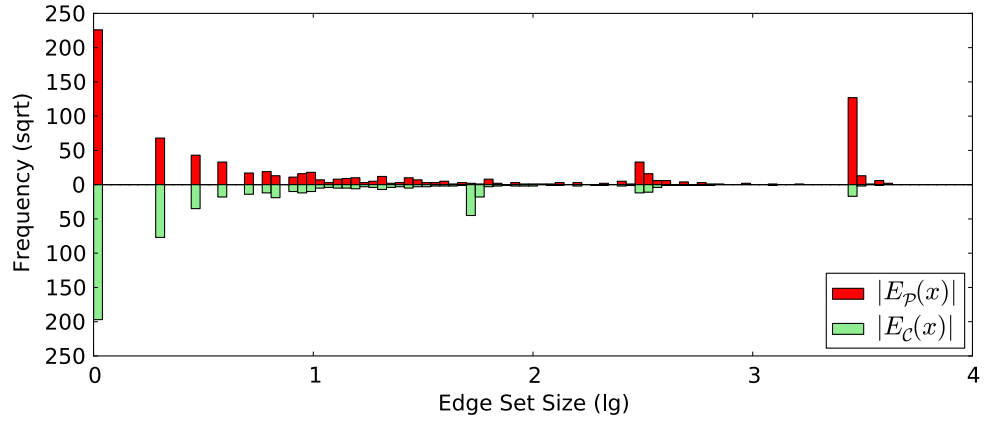


Figure 4.3: Distributions of non-zero outgoing degree $|E_{\mathcal{P}}(x)|$ and $|E_{\mathcal{C}}(x)|$ in `vim`.

4.1.2 Redundant Traversals

When each rule is applied to a node x (lines 12 – 13 or 15 – 17 in Algorithm 1), its sparse bit vector $E_{t3}(x)$ is traversed $|E_{t1}(x)|$ times, one for each $y \in E_{t1}(x)$. As a result, many such redundant traversals are made, especially when $E_{t3}(x)$ is a long list.

It is important to avoid such redundant traversals on long sparse bit vectors, as doing so not only speeds up rule applications but also facilitates load balancing. However, long sparse bit vectors are difficult to identify efficiently before they

are actually traversed, since they grow dynamically and unpredictably during the analysis, as illustrated earlier in Figure 4.2.

4.2 Methodology

We describe an efficient GPU implementation of Andersen’s analysis by both leveraging the state-of-the-art GPU implementation [28] and addressing its two deficiencies described in Section 4.1. In Section 4.2.1, we present a new imbalance-aware workload partitioning scheme to reduce its inter-warp imbalance significantly. In Section 4.2.2, we present an adaptive group propagation scheme to reduce redundant traversals, thereby improving also its inter-warp imbalance further. These two techniques together are expected to provide insights on parallelising other graph algorithms that also make modifications to their input graphs on GPUs.

4.2.1 Imbalance-Aware Workload Partitioning

To maximise utilisation of warp resources with a balanced workload throughout the analysis, we divide a rule application in two stages: (1) a coarse-grain stage when the rule is executed based on the warp-centric model, and (2) a fine-grain stage when the rule is executed based on the task pool model. The first stage aims to exploit coarse-grain parallelism without resorting to any synchronisation. Even though the distribution of degrees (number of edges per node) is highly skewed, the warp-centric model can still be desirable, provided that there are sufficiently many warps to keep all the computational resources busy. As soon as warp resources become underutilised due to workload imbalance, the analysis will start decomposing the remaining workload into smaller tasks and depositing these smaller tasks in a central task pool. The fine-grain stage then comes into play with these smaller

tasks being mapped to warps in a more balanced way (than otherwise). Therefore, this second stage aims to expose fine-grain parallelism at the expense of some synchronisation overhead.

Algorithm 4 Imbalance-Aware Workload Partitioning. W is a worklist formed from V .

Procedure APPLY($t1, t2, t3, V$)
begin

1 $W \leftarrow V;$ 2 $\Omega \leftarrow \emptyset;$ 3 foreach warp w do $waitingWarpCount_{blk(w)} \leftarrow 0;$ 4 synchronise all warps in $blk(w);$ 5 foreach warp w do 6 while $W \neq \emptyset$ do 7 $x \leftarrow \text{GETFROM}(W);$ 8 $\text{PROP}(w, x, t1, t2, t3, \Omega);$ 9 $waitingWarpCount_{blk(w)}++;$ <i>// atomic increment</i> 10 synchronise all warps; 11 foreach warp w do 12 while $\Omega \neq \emptyset$ do 13 $\langle x, y, t2, t3 \rangle \leftarrow \text{GETFROM}(\Omega);$ 14 $E_{t3}(x) \leftarrow E_{t3}(x) \uplus E_{t2}(y);$	<div style="display: inline-block; vertical-align: middle;"> } the coarse-grain stage </div> <div style="display: inline-block; vertical-align: middle; margin-top: 40px;"> } the fine-grain stage </div>
---	---

To implement this two-staged approach, we have modified APPLY in Algorithm 1 as shown in Algorithm 4. However, APPLYOFFSET in Algorithm 1 remains the same, for the following reason. Recall that there are five types of pointer-manipulating statements considered in Table 2.1. Without loss of generality, we have adopted an Intermediate Representation (IR) from [28] that is similar to the LLVM IR so that all top-level variables are in Static Single Assignment (SSA) form. For an offset statement, $x = y + o$, both x and y must be top-level variables, because all address-taken variables, such as z in a points-to statement $\dots = \&z$, can only be accessed indirectly via a load $\dots = *p$ or a store $*p = \dots$. As a result,

each top-level variable x can only be defined once in an offset statement, $x = y + o$, meaning that the sparse bit vector for x contains only y for this particular offset, o . In addition, no graph-rewriting rule listed in Table 2.2 generates any new offset edge during the analysis. Therefore, R_{offset} does not cause workload imbalance as the other four rules do.

We first describe the warp-centric model modified for achieving coarse-grain parallelism in the coarse-grain stage. We then describe our task-pool-based technique for achieving fine-grain parallelism in the fine-grain stage.

The Coarse-Grain Stage In this stage, each rule application, which is realised in lines 5 – 8 of Algorithm 4 (by ignoring the initialisation code in its lines 1 – 4), together with lines 1, and 6 – 7 in Algorithm 5, proceeds exactly the same as before in a warp-centric manner. Every warp, w , is dedicated to handling one node, x . This warp executes each rule of the form of $x \xrightarrow{t1} y \wedge y \xrightarrow{t2} z \Rightarrow x \xrightarrow{t3} z$ for all elements y in the sparse bit vector $E_{t1}(x)$. For a given y , the rule is executed in an SIMD manner, but different elements in $E_{t1}(x)$ are processed sequentially.

Algorithm 5 Imbalance-Aware Propagation. W_x^w is a worklist formed from $E_{t1}(x)$.

Procedure PROP($w, x, t1, t2, t3, \Omega$)

begin

```

1   $W_x^w \leftarrow E_{t1}(x);$ 
2  while  $W_x^w \neq \emptyset$  do
3      if  $\text{waitingWarpCount}_{\text{blk}(w)} > \tau_{\text{imba}}$  then
4           $\Omega \leftarrow \Omega \cup \{ \langle x, y, t2, t3 \rangle \mid y \in W_x^w \};$ 
5          return;
6       $y \leftarrow \text{GETFROM}(W_x^w);$ 
7       $E_{t3}(x) \leftarrow E_{t3}(x) \cup E_{t2}(y);$ 
```

In addition, warp w for node x is also responsible for detecting a workload imbalance for its own thread block (lines 3 – 4 in Algorithm 5). An alert is raised when

$waitingWarpCount_{blk(w)} > \tau_{imba}$, where $waitingWarpCount_{blk(w)}$ is the number of finished warps in the thread block $blk(w)$ that contains warp w and τ_{imba} is a global threshold set for all the thread blocks. We do not detect imbalanced workloads globally since that would require expensive global synchronisation across all the thread blocks.

On detecting a workload imbalance, warp w stops its warp-centric processing at node x . At this point, w may have only partially processed the sparse bit vector $E_{t1}(x)$, with W_x^w indicating the remaining part of the list that has not been processed. In line 4 of Algorithm 5, warp w will decompose the remaining workload in W_x^w into smaller tasks, with one task $\langle x, y, t2, t3 \rangle$ corresponding to every unprocessed element $y \in W_x^w$, insert these tasks into the central task pool Ω , and finally, suspend its execution. Note that once one warp has suspended its execution this way, all the other non-finished warps in the same block will also do so due to line 3 in Algorithm 5.

The warp w dedicated to node x will reach line 8 in Algorithm 4 in one of the two ways: (1) it has finished processing all the elements in $E_{t1}(x)$ or (2) it has done so only partially but has suspended its execution due to workload imbalance. As soon as all the warps in the program have reached this synchronisation point, the fine-grain stage, as discussed below, will begin. Some imbalance across the thread blocks may occur as its detection requires global synchronisation, which is avoided here as discussed above.

The Fine-Grain Stage In this stage, as realised in lines 11 – 14 of Algorithm 4, all the warps will work together executing all the tasks in the central task pool Ω (protected by a lock). Unlike the coarse-grain stage, in which a node x is always updated by the same warp w in line 7 of Algorithm 5, the fine-grain stage allows different warps to update the same node x at the same time. This can happen when

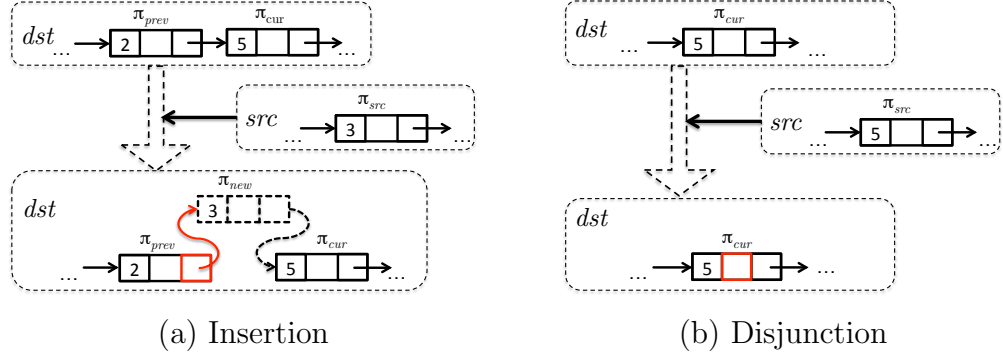


Figure 4.4: Two lock-free atomic operations for merging an element pointed to by π_{src} from a source sparse bit vector src with a destination sparse bit vector dst .

one warp w fetches $\langle x, y, t2, t3 \rangle$ from Ω and another warp w' fetches $\langle x, y', t2, t3 \rangle$ from Ω in line 13 of Algorithm 4. As a result, the same sparse bit vector $E_{t3}(x)$ is updated by both warps, w and w' , in line 14 of Algorithm 5, where \uplus stands for a race-free union operation. Below our primary focus is on how to perform \uplus efficiently (in a race-free manner).

We describe how a warp executes line 14 in Algorithm 4 abstracted as $dst \leftarrow dst \uplus src$, where dst and src are sparse bit vectors, by assuming that dst may also be concurrently updated by another warp executing $dst \leftarrow dst \uplus src'$. This set union operation will be implemented efficiently by using lock-free atomic operations. Note that src and src' remain unchanged. In addition, dst , src and src' may contain elements $e1$, $e2$ and $e3$, respectively, with the same *base*, so that $e2$ and $e3$ will both be combined with $e1$ concurrently.

Let us discuss how to merge a given element pointed to by π_{src} from the source list src with the destination list dst . There are two cases as illustrated in Figure 4.4:

Insertion In this case, π_{src} points to the element with $\pi_{src} \rightarrow base = 3$ such that no element in dst has the same *base*, 3. Let $*\pi_{new}$ be a copy of $*\pi_{src}$ (by ignoring $\pi_{src} \rightarrow next$). Then $*\pi_{new}$ will be inserted between the two elements pointed to by π_{prev} and π_{cur} , respectively, in dst . Recall that each 32-word

element is processed by the 32 threads in a warp. This insertion will be made so that $\pi_{prev \rightarrow next}$ is updated atomically by one thread. After this insertion, dst remains sorted.

Disjunction In this case, the element pointed to by π_{src} is merged with the one pointed to by π_{cur} , where $\pi_{src \rightarrow base} = \pi_{cur \rightarrow base} = 5$. There are 30 words represented by $\pi_{cur \rightarrow bits}$. Each of these words will be updated atomically by one thread. For correctness, it suffices to update each of the 960 bits in $\pi_{cur \rightarrow bits}$ atomically since the variables represented by these bits are independent. Thus, updating all the 30 words atomically (as a whole) is unnecessary and also less efficient.

There are generally more insertions than disjunctions, with their ratio being 1.6:1 on average for the 14 benchmarks evaluated in Section 4.3. There are two main reasons behind. First, $\delta\mathcal{P}$ and $\delta\mathcal{C}$ are reset to \emptyset at the beginning of each iteration and grow with new edges being inserted with subsequent rule applications. Second, sparse bit vectors tend to be very sparse. So src and dst tend to contain elements with different bases.

Algorithm 6 describes how a warp executes $dst \leftarrow dst \uplus src$ by calling (1) INSERTION in Algorithm 7 to perform the basic insertion operation illustrated in Figure 4.4(a) and (2) DISJUNCTION in Algorithm 8 to perform the basic disjunction operation illustrated in Figure 4.4(b). To avoid dealing with special cases in the two operations, we assume that every sparse bit vector has a pseudo head with its $base$ being $-\infty$ and a pseudo tail with its $base$ being $+\infty$. In our implementation, neither pseudo heads nor pseudo tails are actually used. In both INSERTION and DISJUNCTION, a word-wise atomic operation is implemented by *atomicCAS*¹, which is an atomic compare-and-swap operation provided by CUDA.

¹*atomicCAS(addr, old, new)* updates the value *val* stored in *addr*, with *new* if *val* is tested

Algorithm 6 Performing $dst \leftarrow dst \uplus src$ by a warp.

Procedure UNION(dst, src)

begin

```

1   $\pi_{prev} \leftarrow$  address of  $dst$ 's pseudo head;
2   $\pi_{cur} \leftarrow \pi_{prev} \rightarrow next$ ;
3   $\pi_{src} \leftarrow$  address of  $src$ 's pseudo head;
4   $\pi_{src} \leftarrow \pi_{src} \rightarrow next$ ;
5  while  $\pi_{src} \rightarrow base \neq +\infty$  do
6      if  $\pi_{cur} \rightarrow base > \pi_{src} \rightarrow base$  then
7           $\text{INSERTION}(\pi_{prev}, \pi_{cur}, \pi_{src})$ ;
8      else if  $\pi_{cur} \rightarrow base = \pi_{src} \rightarrow base$  then
9           $\text{DISJUNCTION}(\pi_{prev}, \pi_{cur}, \pi_{src})$ ;
10     else
11          $\pi_{prev} \leftarrow \pi_{cur}$ ;
12          $\pi_{cur} \leftarrow \pi_{cur} \rightarrow next$ ;

```

Algorithm 7 Atomic insertion of one element from src into dst .

Procedure INSERTION($\pi_{prev}, \pi_{cur}, \pi_{src}$)

begin

```

1   $\pi_{new} \leftarrow$  fetch a memory block for representing a new element from a pre-
    allocated chunk of memory space;
2   $*\pi_{new} \leftarrow *\pi_{src}$ ;
3  for only one thread in warp  $w$  do
4       $\pi_{new} \rightarrow next \leftarrow \pi_{cur}$ ;
5       $val \leftarrow \text{atomicCAS}(\&\pi_{prev} \rightarrow next, \pi_{new} \rightarrow next, \pi_{new})$ ;
6      if  $\pi_{new} \rightarrow next \neq val$  then
7           $\pi_{cur} \leftarrow val$ ;
8      else
9           $\pi_{prev} \leftarrow \pi_{new}$ ;
10          $\pi_{src} \leftarrow \pi_{src} \rightarrow next$ ;

```

} Try to insert π_{new} after π_{prev}

} failure

} success

Algorithm 8 Atomic disjunction of an element from src with another in dst .

Procedure DISJUNCTION($\pi_{prev}, \pi_{cur}, \pi_{src}$)

```

begin
  // The 30-word-wide bits field of an element is accessed by 30 threads in
  // parallel.
1  foreach thread  $t$  in warp  $w$ , where  $t.id \in \{0, 1, \dots, 29\}$  do
2     $old^t \leftarrow \pi_{cur} \rightarrow bits[t.id]$ ;
3     $union^t \leftarrow old^t \cup \pi_{src} \rightarrow bits[t.id]$ ;           // bitwise OR operation
4    if  $union^t \neq old^t$  then
5       $val^t \leftarrow \text{atomicCAS}(\&\pi_{cur} \rightarrow bits[t.id], old^t, union^t)$ ;
6      while  $old^t \neq val^t$  do
7         $old^t \leftarrow val^t$ ;
8         $union^t \leftarrow old^t \cup \pi_{src} \rightarrow bits[t.id]$ ;   // bitwise OR operation
9         $val^t \leftarrow \text{atomicCAS}(\&\pi_{cur} \rightarrow bits[t.id], old^t, union^t)$ ;
10    $\pi_{prev} \leftarrow \pi_{cur}$ ;
11    $\pi_{cur} \leftarrow \pi_{cur} \rightarrow next$ ;
12    $\pi_{src} \leftarrow \pi_{src} \rightarrow next$ ;

```

In INSERTION, new elements to be inserted are created in pre-allocated memory for efficiency considerations. In DISJUNCTION, each of the 30 words in $\pi_{cur} \rightarrow bits$ is updated atomically by one thread individually, as discussed above. The update for all the 30 words in $\pi_{cur} \rightarrow bits$ is finished after all 30 threads have succeeded.

To show that Algorithm 6 is correct, we argue that a warp performs $dst \leftarrow dst \uplus src$ correctly in the presence of concurrent modifications to dst by other warps, from a few facts. First, src never changes during the fine-grain stage. Second, dst does not admit any delete operations (due to the nature of Andersen's analysis). Third, Algorithm 6 calls INSERTION and DISJUNCTION appropriately as expected. Fourth, INSERTION($\pi_{prev}, \pi_{cur}, \pi_{src}$) inserts π_{src} into dst only when π_{prev} and π_{cur} point to a pair of two consecutive elements in dst such that $\pi_{prev} \rightarrow base < \pi_{src} \rightarrow base < \pi_{cur} \rightarrow base$. If the atomic CAS operation in INSERTION succeeds (line 5), we let π_{prev} point to what π_{new} points to (line 9) so that the next the same as old . It returns val .

element pointed to by π_{src} in *src* (line 10) will be merged somewhere in *dst* after the element pointed to by π_{prev} . Otherwise, we let π_{cur} point to the successor of the element pointed to by π_{prev} (line 7), so that the element pointed to by π_{src} will be inserted either just before the element pointed to by π_{cur} or merged somewhere after in *dst*. In either case, if π_{prev} and π_{cur} do not yet happen to point to two consecutive elements in *dst*, they will eventually be made to do so after some unsuccessful atomic CAS operations. Finally, $\text{DISJUNCTION}(\pi_{prev}, \pi_{cur}, \pi_{src})$ performs a disjunction operation as illustrated in Figure 4.4(b). Once it is successful, π_{prev} , π_{cur} and π_{src} are adjusted appropriately in lines 10 – 12.

4.2.2 Adaptive Group Propagation

In this section, we introduce a so-called *adaptive group propagation* scheme to reduce the number of redundant traversals highlighted in Section 4.1 in both stages of our workload partitioning scheme. Let us examine the problem in more detail by considering a scenario when a warp executes a rule (lines 12 – 13 in **APPLY** from Algorithm 1) at a node during the coarse-grain stage. The same problem remains for **APPLYOFFSET** from Algorithm 1 and during the fine-grain stage (once modified as discussed shortly).

For a node x , its dedicated warp will execute $E_{t3}(x) \leftarrow E_{t3}(x) \cup E_{t2}(y)$ in an SIMD manner for every $y \in E_{t1}(x)$, with different y 's being processed sequentially. As a result, each *source* sparse bit vector $E_{t2}(y)$ (for a given y) is propagated into (i.e., combined with) the *destination* sparse bit vector $E_{t3}(x)$ individually, resulting in the destination list being traversed a total of $|E_{t1}(x)|$ times, once for each element in $E_{t1}(x)$. Such redundant traversals can be inefficient, particularly when $E_{t1}(x)$ is a long list.

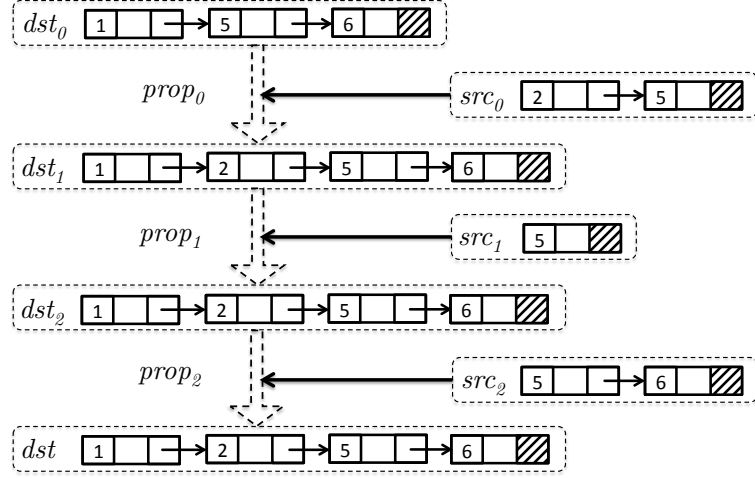
Our basic idea for reducing redundant traversals is simple. Let y_0, \dots, y_{n-1}

be a group of n variables taken from $E_{t1}(x)$. There are n (sorted) source lists $E_{t2}(y_0), \dots, E_{t2}(y_{n-1})$, where each list may contain at most one element of a given *base*. Instead of solving $E_{t3}(x) \leftarrow E_{t3}(x) \cup E_{t2}(y_i)$ for each y_i individually for a total of n times, we combine $E_{t2}(y_0), \dots, E_{t2}(y_{n-1})$ into a so-called *virtual sparse bit vector* $E_{t2}(y) = \cup_{0 \leq i < n} E_{t2}(y_i)$ conceptually (but not actually in our implementation), by merging their elements with a common *base*, where y can be understood to encapsulate all the n variables y_0, \dots, y_{n-1} in the group. We then execute $E_{t3}(x) \leftarrow E_{t3}(x) \cup E_{t2}(y)$ as before but only once. Of course, such a group propagation scheme will be performed adaptively so that the benefit reaped will offset the overhead incurred.

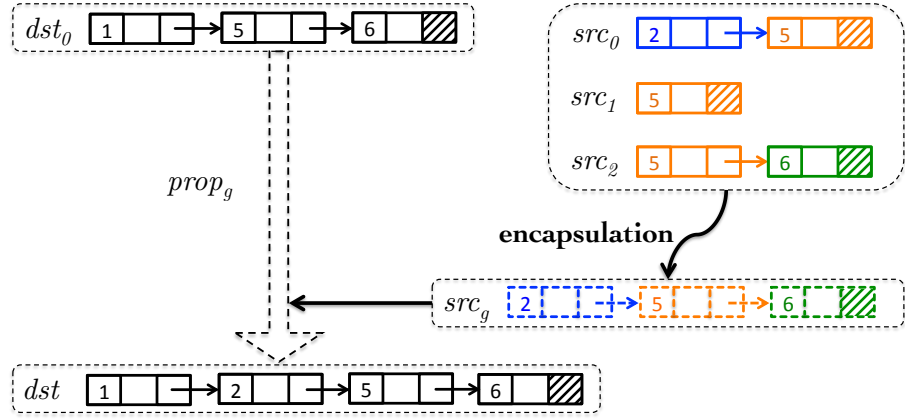
We first give an illustrating example. We then describe how our earlier algorithms are modified to incorporate this group propagation scheme. Lastly we give a cost-benefit analysis in order for the scheme to be used profitably.

An Example In this example shown in Figure 4.5, there are three source lists src_0 , src_1 and src_2 and one destination list dst_0 . Figure 4.5(a) depicts the three individual propagations, $prop_0$, $prop_1$ and $prop_2$, in that order, producing successively, dst_1 , dst_2 and dst . Figure 4.5(b) depicts our group propagation, $prop_g$. The three source lists are conceptually combined into a virtual one src_g , with all elements of the same *base* merged, and then propagated into dst_0 only once. Figure 4.5(c) demonstrates the performance benefit of our scheme in terms of memory accesses reduced, measured in terms of the number of instructions executed for reading from or writing into the elements of all the sparse bit vectors involved (at the warp level). With 20 scored by the former and 15 by the latter, our scheme appears to be more efficient.

The parameters regarding read and write memory accesses listed in Figure 4.5(c) during the execution of $dst \leftarrow dst \cup src_i$, where $i \in \{0, 1, 2, g\}$, are estimated as



(a) Individual propagation



(b) Group propagation

Operation	Individual Propagation				Group Propagation	
	$prop_0$ ($i = j = 0$)	$prop_1$ ($i = j = 1$)	$prop_2$ ($i = j = 2$)	$\Sigma prop_i$	$prop_g$ ($i = g, j = 0$)	Overhead
$R_{dst_j}^{src_i}$	2	3	4	9	3	-
$W_{dst_j}^{src_i}$	3	1	2	6	4	-
R_{src_i}	2	1	2	5	8	3
Total	7	5	8	20	15	3

(c) Memory accesses incurred, where “overhead” indicates the additional number of reads made in src_g (as pure overhead)Figure 4.5: Comparing individual and group propagation schemes for executing $dst \leftarrow dst \cup src_i$ with three source lists, src_0 , src_1 and src_2 , by a warp in terms of their algorithms and the numbers of memory accesses incurred (at the warp level).

follows:

- R_{src_i} ($i \in \{0, 1, 2\}$), which denotes the number of reads (i.e., elements) in src_i , is $|src_i|$, which represents the number of elements in src_i .
- R_{src_g} , the number of reads incurred in src_g , is divided into (1) the number of reads in the three source lists, i.e., $|src_0| + |src_1| + |src_2|$ and (2) the number of additional reads incurred for maintaining the virtual sparse bit vector src_g , i.e., $|src_g|$.
- $R_{dst}^{src_i}$ ($i \in \{0, 1, 2, g\}$), which represents the number of reads in dst , depends on the location in dst where the element with the largest *base* in src_i is merged. Let $b_{\max}^{src_i}$ be the largest *base* in src_i and B_{dst} the set of all *bases* in dst . Then we have:

$$R_{dst}^{src_i} = \begin{cases} |B_{dst}| & \forall b \in B_{dst} : b < b_{\max}^{src_i} \\ |\{b \in B_{dst} \mid b \leq b_{\max}^{src_i}\}| & b_{\max}^{src_i} \in B_{dst} \\ |\{b \in B_{dst} \mid b < b_{\max}^{src_i}\}| + 1 & \text{otherwise} \end{cases}.$$

- $W_{dst}^{src_i}$ ($i \in \{0, 1, 2, g\}$), the number of writes in dst , is the sum of $|src_i|$ and the number of elements in src_i inserted into dst . In computing this estimate, the atomic CAS operations in Algorithm 7 and 8 are assumed to succeed optimistically. Each element in src_i is merged into dst with either a disjunction or insertion operation. In the former case, one write is incurred. In the latter case, two writes are needed, one for cloning the inserted element (line 2 in Algorithm 7) and one for updating a *next* link in dst (line 5 in Algorithm 7).

The Modified Algorithm In order to incorporate our group propagation scheme, we will modify Algorithms 5 – 8 slightly as follows:

Algorithm 5 Instead of a single variable from W_x^w in line 6, a warp will fetch up to $n = \min(|W_x^w|, 32)$ variables from W_x^w to form a group \mathcal{G} , where $|\mathcal{G}| = n$. In line 7, y can be understood as representing the n variables y_0, \dots, y_{n-1} in \mathcal{G} , so that the warp will be essentially executing $E_{t3}(x) \leftarrow E_{t3}(x) \uplus E_{t2}(y)$, where $E_{t2}(y) = \cup_{0 \leq i < n} E_{t2}(y_i)$ symbolises a so-called virtual sparse bit vector illustrated in Figure 4.5.

Algorithm 4 Similarly, $E_{t2}(y)$ in line 14 is also a virtual sparse bit vector.

Algorithm 6 – 8 In all these algorithms, src represents a virtual sparse bit vector, with π_{src} indicating a virtual element in src . Let the n concrete sparse bit vectors encapsulated by src be src_0, \dots, src_{n-1} . Let $\Phi_{\mathcal{G}}$ be an n -element array. Then the initialisation for π_{src} in line 3 of Algorithm 6 is replaced by the following initialisation for $\Phi_{\mathcal{G}}$: (1) $\Phi_{\mathcal{G}}[i].addr$ is set to point to the first non-pseudo element π in src_i and (2) $\Phi_{\mathcal{G}}[i].base = \pi.base$. In addition, each occurrence of $\pi_{src} \leftarrow \pi_{src \rightarrow next}$ is replaced by $\pi_{src} = \text{GETVIRTELEM}(\Phi_{\mathcal{G}})$, where π_{src} , which is stored in the shared memory (for efficiency reasons), is now representing an element instead of a pointer as before. As a result, each occurrence of $*\pi_{src}$, $\pi_{src \rightarrow base}$ and $\pi_{src \rightarrow bits}$ is replaced by π_{src} , $\pi_{src}.base$ and $\pi_{src}.bits$, respectively.

The basic functionality of `GETVIRTELEM` given in Algorithm 9, as illustrated in Figure 4.5, is simple. All the n concrete sparse bit vectors src_0, \dots, src_{n-1} are conceptually combined into a virtual one, where $n = |\mathcal{G}|$. On each invocation, its first element is removed and returned. As $\Phi_{\mathcal{G}}[i].base$ can be accessed multiple times, $\Phi_{\mathcal{G}}$ is stored in the shared memory. However, due to the resource limits, its $bits$ field is not cached. Note that almost all statements are executed in parallel by the threads in the same warp, except for a parallel reduction done in $O(\log_2 |\mathcal{G}|)$ in

line 2.

Algorithm 9 Implementing $\pi_{src} \leftarrow \pi_{src \rightarrow next}$ equivalently as $\pi_{src} = \text{GETVIRTELEM}(\Phi_{\mathcal{G}})$ for a virtual sparse bit vector src that encapsulates all the concrete sparse bit vectors represented by $\Phi_{\mathcal{G}}$.

Procedure GETVIRTELEM($\Phi_{\mathcal{G}}$)

begin

```

1   $\Pi_{\mathcal{G}}.bits \leftarrow 0;$  // in parallel
2   $\Pi_{\mathcal{G}}.base \leftarrow \min_{0 \leq i < |\mathcal{G}|} \Phi_{\mathcal{G}}[i].base;$  // parallel reduction
3  if  $\Pi_{\mathcal{G}}.base = +\infty$  then return  $\Pi_{\mathcal{G}};$ 
4  foreach  $i \in \{0, 1, \dots, |\mathcal{G}| - 1\}$  do
5      if  $\Phi_{\mathcal{G}}[i].base = \Pi_{\mathcal{G}}.base$  then
6          foreach thread  $t$  in warp  $w$  do
7              // coalesced access by 32 threads
8               $((\&\Pi_{\mathcal{G}}))[t.id] \leftarrow ((\&\Pi_{\mathcal{G}}))[t.id] \cup ((\Phi_{\mathcal{G}}[i].addr))[t.id];$ 
9  foreach thread  $t$  in warp  $w$  do
10     if  $0 \leq t.id < |\mathcal{G}| \wedge \Phi_{\mathcal{G}}[t.id].base = \Pi_{\mathcal{G}}.base$  then
11          $addr^t \leftarrow \Phi_{\mathcal{G}}[t.id].addr \rightarrow next;$ 
12          $\Phi_{\mathcal{G}}[t.id].addr \leftarrow addr^t;$ 
13          $\Phi_{\mathcal{G}}[t.id].base \leftarrow addr^t \rightarrow base;$ 
14 return  $\Pi_{\mathcal{G}};$ 

```

A Cost-Benefit Analysis The major source of overhead in our group propagation scheme comes from the extra global memory read accesses incurred in transforming conceptually a number of source sparse bit vectors into a virtual one. Every element in a source list is accessed twice, once for caching its *base* value before it is selected (line 13), and once for accessing its *bits* and *next* fields after it has been selected (implemented in one coalesced global read for both fields in lines 8 and 11). The memory reads in the first case will be more time-consuming as being uncoalesced.

Therefore, it is necessary to conduct a cost-benefit analysis as illustrated in Figure 4.5 to determine if our group propagation scheme is more profitable or not

than the default individual propagation scheme. For this motivating example, the overall memory access overhead for group propagation (as analysed earlier) is:

$$cost_{group} = R_{src_g} + R_{dst_0}^{src_g} + W_{dst_0}^{src_g} = 15 \quad (4.1)$$

The overall memory access overhead for individual propagation, where the number of source sparse bit vectors is $n = 3$, is given by:

$$cost_{indiv} = \sum_{i=0}^{n-1} R_{src_i} + \sum_{i=0}^{n-1} R_{dst_i}^{src_i} + \sum_{i=0}^{n-1} W_{dst_i}^{src_i} = 20 \quad (4.2)$$

Since $cost_{group} < cost_{indiv}$, group propagation is considered to yield better performance.

It is impractical to conduct such sophisticated cost-benefit analysis by using (4.1) and (4.2) during Andersen’s analysis. A simple yet efficient adaptive strategy is to prefer group propagation when n exceeds a threshold:

$$n \geq \tau_{grp} \quad (4.3)$$

4.3 Evaluation

Using a set of 14 C benchmarks, we evaluate our GPU implementation of Andersen’s analysis against a *reference* state-of-the-art GPU solution, which was implemented by the original authors in [28]. We show that our GPU implementation, with balanced workloads and reduced redundancies, achieves significantly better speedups than this reference GPU implementation. We also show that our GPU implementation outperforms a state-of-the-art multi-CPU implementation, which is also discussed in [28], in most of the benchmarks evaluated.

Benchmark	#Variables	#Statements of Five Different Types					
		\mathcal{P}	\mathcal{C}	\mathcal{L}	\mathcal{S}	\mathcal{F}	Total
ex	11,078	1,423	3,881	1,940	611	826	8,681
gcc	120,870	6,224	68,225	25,506	5,625	21,591	127,171
gdb	232,814	25,783	92,386	36,693	8,229	35,842	198,933
gimp	558,867	53,636	347,199	63,245	14,245	87,330	565,655
linux	919,188	83,202	129,218	79,336	38,077	69,993	399,826
mplayer	537,254	39,099	154,100	42,030	9,945	69,707	314,881
nh	97,936	12,283	52,748	14,109	1,345	20,756	101,241
perl	53,361	3,467	27,972	13,157	3,120	8,261	55,977
php	339,538	22,576	135,399	50,689	10,319	50,346	269,329
pine	612,916	33,259	142,039	43,947	10,057	37,135	266,437
python	92,599	10,274	45,020	18,396	3,037	16,100	92,827
svn	107,708	9,564	76,394	17,118	5,134	14,348	122,558
tshark	1,555,835	522,148	867,549	39,762	21,579	71,131	1,522,169
vim	246,944	9,886	42,099	16,208	3,004	18,031	89,228
Average	391,922	59,487	156,016	33,010	9,595	37,243	295,351

Table 4.1: Benchmark statistics: sizes of initial constraint graphs.

4.3.1 Methodology and Benchmarks

We have conducted Andersen’s analysis to a program to reflect the way it is used in practice. In particular, cycle elimination is incorporated to speed up its analysis. We have developed our GPU implementation on top of the reference GPU implementation [28]. In Andersen’s analysis, every Strongly-Connected Cycle (SCC) formed by copy edges can be collapsed since its variables have the same points-to edges. Hybrid Cycle Detection (HCD) [14] has been employed to detect and collapse such SCCs in both the reference and our implementations. HCD comes with

two stages: an offline stage that collapses all the cycles detectable on the initial graph, and an online stage that deals with the cycles formed dynamically during constraint resolution.

We use the same benchmark suite used in the reference GPU implementation [28]. There are 14 C programs with 11K – 1556K variables (i.e., graph nodes) and 9K – 1522K statements (i.e., edges in their initial graphs), which are obtained after the offline stage has been performed. Some of their salient properties are given in Table 4.1.

We compare the reference implementation, denoted *Ref*, with three different configurations of our implementation: (1) *I* with our imbalance-aware workload partitioning scheme used alone, (2) *G* with our adaptive group propagation scheme used alone, and (3) *IG* with both schemes used at the same time.

4.3.2 Experimental Settings

We conducted our experiments on a 0.71GHz NVIDIA Tesla K20c GPU. This Kepler-based GPU has 4.6GB global memory and 13 SMXs, each containing 192 cores. Each SMX has a private 64KB of on-chip memory, which is configured as 48KB of shared memory and 16KB of L1 cache. All SMXs share a 1280KB L2 cache. The CUDA code is compiled under NVCC v5.5, with the flag “-m64 -arch=sm_30” and the optimization level “-O2”.

Each of the five rule applications performed by calling `APPLY` or `APPLYOFFSET` and each of the updates (lines 3 – 10 in Algorithm 1), is executed by a separate GPU kernel (Some rule applications in the reference implementation are combined for more balanced workload, as discussed in Section 2.1.2). For each kernel, 13 thread blocks (for 13 SMXs) and 1024 threads per block (32 warps per block) are used, since it is good to have many threads per block where communication among

threads within the same block is cheaper [28].

The online phase of HCD, which detects and collapses SCCs during online Andersen’s analysis, uses 512 threads per block (16 warps per block). Its thread count is not maximized (1024), as restricted by shared memory (48KB per block), which is heavily used as a cache for the operations on sparse bit vectors [28].

Table 4.2 summarises the number of blocks and the number of threads per block used by the GPU kernels in both the reference and our implementations. According to [28] and our own evaluations, both implementations yield the best performance in this setting. The resource utilization of our GPU implementation is 50% according to the CUDA occupancy calculator. We can, for example, push it up to 70% by using 39 blocks with 480 threads per block, but the implementation is 42% slower on average. Thus, we are better off by using more threads per block for cheaper communication.

Kernel	#Blocks	#Threads per Block
APPLY/APPLYOFFSET	13	1024
Update	13	1024
HCD	13	512

Table 4.2: Kernel configurations.

For imbalance-aware workload partitioning, the threshold τ_{imba} , which is used to decide when to start the fine-grained stage in Algorithm 5, is set to 16, as we consider a load imbalance to occur when fewer than half of the (32) warps in a thread block remain active. For adaptive group propagation, τ_{grp} in (4.3) is set to 8.

Benchmark	Analysis Times (ms)				Speedups (X)		
	t_{Ref}	t_I	t_G	t_{IG}	r_I	r_G	r_{IG}
ex	90	90	90	90	1.00	1.00	1.00
gcc	310	300	300	300	1.03	1.03	1.03
gdb	3,260	3,000	2,999	2,909	1.09	1.09	1.12
gimp	6,240	5,530	5,470	5,450	1.13	1.14	1.14
linux	11,400	10,090	10,320	9,880	1.13	1.10	1.15
mplayer	7,780	6,160	6,760	5,900	1.26	1.15	1.32
nh	210	190	180	180	1.11	1.17	1.17
perl	370	340	340	320	1.09	1.09	1.16
php	6,610	6,160	6,110	5,780	1.07	1.08	1.14
pine	7,070	5,980	6,439	5,850	1.18	1.10	1.21
python	1,250	1,060	1,110	1,040	1.18	1.13	1.20
svn	1,530	1,220	1,290	1,150	1.25	1.19	1.33
tshark	3,130	1,250	1,380	1,230	2.50	2.27	2.54
vim	7,940	2,100	6,490	2,000	3.78	1.22	3.97
Average	4,085	3,105	3,520	3,006	1.41	1.20	1.46

Table 4.3: Analysis times and speedups (with t_x and r_x representing the analysis time and speedup of $x \in \{I, G, IG\}$ over Ref for a program, respectively).

4.3.3 Speedups

Table 4.3 compares the reference implementation *Ref* with the three configurations, I , G and IG , of our implementation in terms of their analysis times. Note that the analysis time required for a large program is not always longer than that for a smaller one (with the size of a program being measured in terms of the size of its initial graph). The most notable examples are **tshark** and **vim**: **tshark** has slightly over 6 times as many variables and 17 times as many edges initially

as `vim` (Table 4.1), but it is only up to 61% as costly as `vim` to analyse for any implementation (Table 4.3).

Our imbalance-aware workload partitioning scheme alone results in a substantial average speedup of 41%, reaching 378% for `vim`. Our adaptive group propagation scheme alone achieves an average speedup of 20%, reaching 227% for `tshark`. The worst-case scenario happens at `ex`, where no speedup is observed in either scheme. For this particular program, its graph is relatively small. Neither scheme has yielded a performance gain that has noticeably exceeded the overhead incurred.

Among the two schemes introduced in this chapter, imbalance-aware workload partitioning is more effective overall, since it is proposed to tackle head-on the unbalanced workloads encountered in Andersen’s analysis. However, our adaptive group propagation scheme is also useful when used on top of our imbalance-aware workload partitioning scheme for many programs evaluated, by pushing the average speedup to 46%.

4.3.4 Effectiveness of Workload Balancing

We evaluate the effectiveness of imbalance-aware workload partitioning by analysing the maximum and average warp execution times, which provide a good indication about the extent of imbalance among the concurrently running warps. For each program, we include only the execution time elapsed on the kernels performing rule applications, since these kernels represent the dominant workload in the program.

Figure 4.6 plots the maximum and average warp execution times consumed by the reference implementation *Ref* and our implementation *IG* for all the 14 benchmarks. Our solution has addressed effectively the load imbalancing problem inherent in Andersen’s analysis. In the case of *IG*, the gap between IG_{\max} and

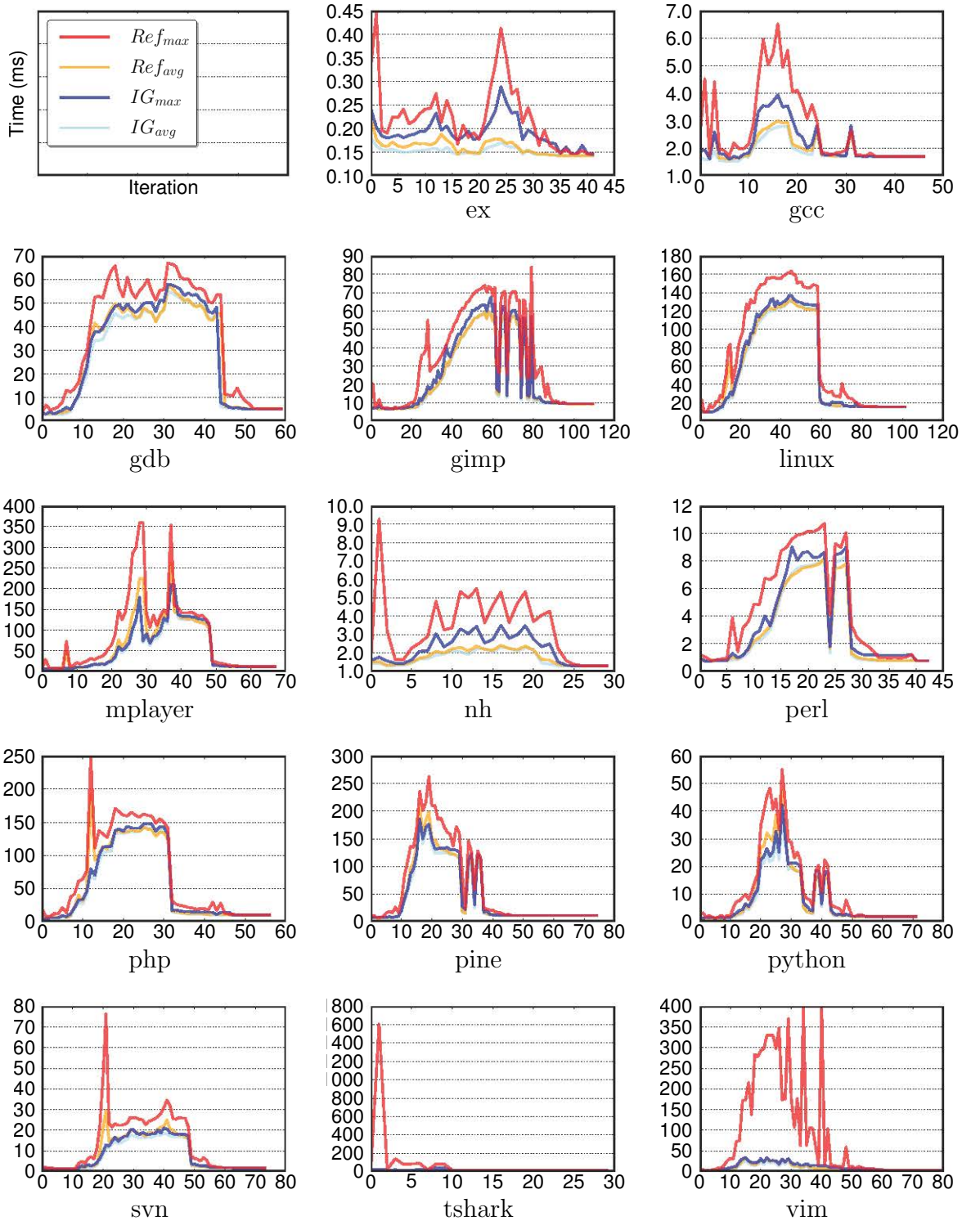


Figure 4.6: Effectiveness of load balancing achieved by the two proposed schemes used together. According to the legend positioned at the top-left corner, the x axis indicates the iterations performed and the y axis the maximum and average execution times (in ms) for both the reference implementation Ref and our implementation IG .

IG_{avg} is insignificant for most benchmarks. In contrast, Ref gives rise to wide gaps between Ref_{max} and Ref_{avg} in many benchmarks such as **ex**, **gcc**, **nh**, **tshark** and **vim**, indicating severely imbalanced workloads exercised by Ref . In particular, our solution IG is notably effective for **tshark** and **vim**. In the case of **ex**, **gcc** and **nh**, the workloads across the analysis iterations under IG are still not as balanced as expected but have been significantly alleviated compared to the workloads under Ref . In general, load balancing for such benchmarks with short analysis times, is hard to improve significantly.

For each program, the gap between Ref_{max} and IG_{max} provides the reason for the speedup achieved by IG over Ref . The largest gaps are found in **tshark** and **vim**, where the best speedups (2.54X and 3.97X, respectively) are achieved. In the case of **ex** and **gcc**, IG has achieved more balanced workloads than Ref in terms of Ref_{max} and IG_{max} but has not improved the analysis times of these two benchmarks by much (Table 4.3). This is because the analysis times for these two benchmarks are so short that the benefits achieved do not outweigh remarkably the overheads incurred.

4.3.5 Effectiveness of Adaptive Group Propagation

We evaluate the effectiveness of our adaptive group propagation scheme by comparing G with Ref in terms of the number of memory accesses made, as discussed in Section 4.2.2 and obtained by instrumenting these two implementations.

Table 4.4 gives the results for the 14 benchmarks used. By achieving an average reduction of 21% (with $\mathcal{R}_{RW} = 0.79$) on the number of memory accesses (both reads and writes) made by Ref , G has achieved an average speedup of 20% over Ref . We can see that a reduction on the number of writes is limited (at 3%) but more pronounced for the number of reads (at 26%). Our adaptive group propa-

gation scheme is the most effective for `tshark` and `vim`, where the best speedups (2.27X and 1.22X, respectively) are observed, with their r_{RW} being 0.57 and 0.64, respectively.

Benchmark	<i>Ref</i> (millions)		<i>G</i> (X)			
	$\#R$	$\#W$	\mathcal{R}_R	\mathcal{R}_W	\mathcal{R}_{RW}	Speedup
ex	0.05	0.02	0.95	1.00	0.96	1.00
gcc	2.53	1.73	0.98	0.98	0.98	1.03
gdb	222.36	64.88	0.73	0.97	0.79	1.09
gimp	175.53	97.53	0.85	0.99	0.90	1.14
linux	491.19	226.84	0.87	0.99	0.91	1.10
mplayer	793.77	107.04	0.58	0.97	0.63	1.15
nh	1.80	1.04	0.89	0.97	0.92	1.17
perl	4.08	2.01	0.90	1.00	0.93	1.09
php	328.95	111.10	0.63	0.99	0.72	1.08
pine	637.05	103.94	0.62	0.98	0.67	1.10
python	151.20	14.57	0.60	0.93	0.63	1.13
svn	69.57	22.97	0.77	0.98	0.82	1.19
tshark	15.57	5.72	0.44	0.94	0.57	2.27
vim	298.66	28.19	0.61	0.95	0.64	1.22
Average			0.74	0.97	0.79	1.20

Table 4.4: Effectiveness of adaptive group propagation. For a benchmark, $\#R$ and $\#W$ denote the numbers of reads and writes (in millions), respectively, made by *Ref* (measured as discussed in Section 4.2.2). \mathcal{R}_R represents the ratio of the number of reads made by our implementation *G* over that of *Ref*. \mathcal{R}_R and \mathcal{R}_{RW} are similarly defined but for writes and reads + writes (i.e., all memory accesses), respectively.

4.3.6 Comparing with a Parallel CPU Implementation

Table 4.5 compares our GPU implementation *IG* with a state-of-the-art multi-CPU implementation in terms of the results reported in [28] on a machine running Ubuntu 10 with four 4-core 2.7 GHz AMD Opteron processors. The baseline, *CPU-1*, is a sequential Andersen’s analysis running on one CPU core. The parallel CPU implementation, *CPU-16*, runs with 16 threads, one per core. For comparison purposes, we have also included the reference GPU implementation *Ref* discussed earlier.

Benchmark	<i>CPU-1</i> (ms)	<i>CPU-16</i>	<i>Ref</i>	<i>IG</i>
ex	400	1.54	4.44	4.44
gcc	1,000	4.63	3.23	3.33
gdb	31,300	6.95	9.60	10.76
gimp	20,500	7.83	3.29	3.76
linux	120,340	7.67	10.56	12.18
mplayer	66,260	6.07	8.52	11.23
nh	1,280	5.54	6.10	7.11
perl	1,990	6.18	5.38	6.22
php	44,670	5.97	6.76	7.73
pine	38,950	4.93	5.51	6.66
python	17,890	3.99	14.31	17.20
svn	14,630	5.70	9.56	12.72
tshark	12,110	3.53	3.87	9.85
vim	10,110	9.39	1.27	5.06
Average		5.71	6.60	8.45

Table 4.5: The speedups of *Ref*, *IG* and *CPU-16* (a 16-thread parallel implementation of Andersen’s analysis introduced in [28]) over the baseline, *CPU-1* (the sequential Andersen’s analysis). The results of *CPU-1* and *CPU-16* are taken directly from [28], where these results are obtained on a machine running Ubuntu 10 with four 4-core 2.7 GHz AMD Opteron processors.

On average, *IG* outperforms *CPU-16* (and *Ref* as discussed earlier). In addition, *IG* is the best performer for all benchmarks except *gcc*, *gimp* and *vim*. In particular, *IG* runs significantly faster than *CPU-16* for *python* but nearly twice as slow as *CPU-16* for *vim*. In the case of *python*, there are many rule applications of R_{offset} , which are highly efficient on sparse bit vectors under *IG*. For *vim*, many operations on the BDDs (Binary Decision Diagrams) that are used for representing points-to information under *CPU-16* are cached, i.e. memorised to speed up the same operations performed later.

4.4 Related Work

4.4.1 Sequential Andersen’s Analysis

Among a number of different styles of pointer analysis [4, 24, 25, 43, 44, 45, 60, 63, 68], Andersen’s analysis provides a good trade-off between precision and efficiency. This analysis plays an important role in many other analyses, including program slicing [47], interprocedural SSA analysis [5], precise pointer analysis [15, 32, 52], and bug detection [53, 66].

There are a number of optimisations on Andersen’s analysis running on a single CPU core [10, 11, 14, 17, 34, 36, 38, 52, 56, 59], demonstrating the importance of Andersen’s analysis as an on-going research topic.

In the difference propagation scheme introduced in [11], $\Delta\mathcal{P}$ was used to avoid propagating some points-to edges unnecessarily. Some later improvements can be found in [36, 46]. This technique has been used in our GPU implementation, for not only $\Delta\mathcal{P}$ ($\delta\mathcal{P}$) but also $\delta\mathcal{C}$.

Cycle detection and elimination was proposed in [10, 14, 36] to detect and collapse the SCCs formed by copy edges, where the variables in an SCC have the

same points-to edges. In our implementation, HCD [14] has been employed both for the offline phase (graph initialisation) and the online phase (constraint resolution).

Some other techniques [34, 38] are introduced to speed up Andersen’s analysis by influencing the order in which the points-to information is propagated during constraint resolution. Unlike these techniques, our two schemes accelerate Andersen’s analysis by taking advantage of the full power of GPUs. In addition, our group propagation scheme can also be used to avoid redundant traversals in the sequential setting.

4.4.2 Parallel Pointer Analysis

In recent years, there have been several attempts on parallelising pointer analysis algorithms on GPUs. Méndez-Lojo *et al.* [28] proposed the first GPU implementation of Andersen’s analysis, which is offset-based, field-sensitive but flow- and context-insensitive, formulated in Section 2.1. This first GPU implementation achieves an average speedup of 7X over a sequential implementation on a 1.15GHz NVIDIA Tesla C2070 GPU with 14 SMs (448 cores). The performance benefits come mainly from the warp-centric workload partitioning scheme used. In particular, their graph representation is based on the warp-size sparse bit vectors (with 128 bytes per element). Furthermore, the points-to information is propagated using a so-called pull-based method, so that a rule can be applied to different graph nodes simultaneously without synchronisation.

However, their implementation suffers from imbalanced workloads. While several built-in heuristics (as described in Section 2.1.2) can alleviate moderately the workload imbalance problem, highly imbalanced workloads are still present in some benchmarks, as highlighted in Section 4.1. In this chapter, we focus primarily on developing effective techniques for overcoming this obstacle, achieving an average

speedup of 46%.

Nasre [32] later introduced another GPU implementation of Andersen’s analysis, by also considering flow-sensitivity. This work explores the use of a bloom filter, trading off between precision of analysis results and performance. The flow-insensitive GPU implementation introduced in [28] is used as a pre-analysis to guarantee that the precision achieved lies always between that of flow-sensitive and flow-insensitive versions. With the bloom filter data structure, this parallel analysis achieves an average speedup of 7.8X over the sequential analysis also on a 1.15GHz NVIDIA Tesla C2070 GPU. The techniques described in this chapter can be used to further speed up such analysis (e.g., its flow-insensitive pre-analysis).

There are also several implementations of pointer analysis on multicore CPU systems, targeting either C or Java programs [8, 29, 31, 40]. By considering field-, flow- and/or context-sensitivity, these parallel implementations have achieved various speedups (2.6X – 4.4X) over the corresponding sequential analysis.

4.4.3 Parallel Graph Algorithms

Nasre *et al.* [33] proposed general techniques for GPU implementations of irregular graph algorithms such as Delaunay mesh refinement, survey propagation, pointer analysis and minimum spanning tree. This chapter aims at reducing load imbalance and redundant computations in a parallel GPU implementation of Andersen’s analysis.

There is a lot of work on parallelising graph algorithms such as breadth-first search (BFS) [1, 12, 13, 16, 19, 20, 30, 35], single-source and all-pairs shortest path [16], SCC detection [3], flow analysis [39] and PageRank [12, 13]. Dynamic workload distribution is employed to divide heavy workloads into fine-grain ones for BFS on GPUs [19]. This is similar in spirit to the use of a fine-grain stage in

our imbalance-aware workload partitioning scheme. However, the graph workloads in BFS are predictable since the underlying graphs do not change. In contrast, the graph workloads in Andersen’s analysis are not as these graphs, but undergo a series of modifications during the analysis.

4.5 Chapter Summary

This chapter describes an efficient GPU implementation of Andersen’s analysis. The presence of dynamic and unpredictable modifications to constraint graphs makes it difficult to balance such graph workloads and avoid redundant traversals during Andersen’s analysis. We address these two challenges by introducing an imbalance-aware workload partitioning scheme and an adaptive group propagation scheme. For a set of 14 C programs evaluated, our GPU implementation outperforms the state-of-the art by achieving an average speedup of 46% on an NVIDIA Tesla K20c GPU.

Chapter 5

Whole-Program Analysis on Heterogeneous CPU-GPU Systems

In this chapter, we describe the first heterogeneous CPU-GPU solution to Andersen’s analysis. We take advantage of a previous formulation of this analysis in terms of graph-rewriting rules [28]. We minimise workload imbalance via a dynamic workload distribution scheme and minimise communication by adopting difference propagation and overlapping communication and computation. Our solution was previously published in [49]. Section 5.1 highlights some architectural differences between CPU and GPU, and motivates our solution. Section 5.2 describes our CPU-GPU solution to Andersen’s analysis. In Section 5.3, we present several optimisations for further improving its performance. Section 5.4 evaluates and analyses our solution. Related work on heterogeneous solutions to graph algorithms is presented in Section 5.5. Section 5.6 concludes this chapter.

5.1 Motivation

We recall the key differences [21] with respect to the CPU-GPU system used in this chapter. The host is equipped with two eight-core Intel Xeon CPUs and the accelerator is an NVIDIA GPU, TESLA K20c, based on the Kepler architecture. The GPU consists of 13 SMXs, each containing 192 cores, giving rise to thousands of GPU cores (two orders of magnitude more than the host). In addition, the GPU has a peak memory bandwidth of 208GB/s, about 10 times of that for the host. This suggests that the GPU is well suited for regular, balanced workloads with abundant data parallelism when its massive number of cores and high memory bandwidth are fully utilised. However, the GPU, which clocks at 0.71 GHz, is less powerful than a CPU, which clocks at 2.0 GHz. In addition, the GPU has memory access latency of 400 – 800 cycles, making it less competitive than CPU if its cores and memory bandwidth are underutilised, which is hard to avoid for irregular, imbalanced workloads.

In the case of imbalanced workloads, the GPU’s computational resources can be underutilised, as discussed in Section 4.1. For CPU, however, usually only dozens of threads can be launched at the same time. The inter-thread imbalance is not as severe, especially when the CPU’s memory access latency, which is lower than the GPU, can be hidden by large caches.

When parallelising Andersen’s analysis on a CPU-GPU system, we exploit the respective architectural advantages of CPU and GPU to accelerate its performance.

5.2 Method

A naive solution would be to dynamically assign portions of the constraint graph of a program to the CPU and GPU and let them apply all graph-rewriting rules applicable to their own portions. As evaluated later, this simplistic solution suffers from poor performance, due to workload imbalance and communication overhead incurred, because the modifications to the underlying graph can be unpredictable.

Algorithm 10 A CPU-GPU solution of Andersen’s analysis.

```

begin
1   $G = (V, E) \leftarrow \text{CREATEGRAPH}();$ 
2  repeat
3    Reset  $\mathcal{W}$ ;
4    CPU side
5    
      FETCHANDAPPLY( $\mathcal{W}$ );
      Transfer  $E_{\Delta_{\text{CPU}}}$  to GPU;
       $E_{\Delta} \leftarrow E_{\Delta_{\text{CPU}}} \cup E_{\Delta_{\text{GPU}}}$ ;
       $E \leftarrow E \cup E_{\Delta}$ ;
    
6    GPU side
7    
      FETCHANDAPPLY( $\mathcal{W}$ );
      Transfer  $E_{\Delta_{\text{GPU}}}$  to CPU;
       $E_{\Delta} \leftarrow E_{\Delta_{\text{GPU}}} \cup E_{\Delta_{\text{CPU}}}$ ;
       $E \leftarrow E \cup E_{\Delta}$ ;
    
    until fixed-point;
```

The basic idea behind our solution is sketched in Algorithm 10. Initially, CREATEGRAPH used in Algorithm 1 is called to initialise the constraint graph identically on both the CPU and GPU. Then Andersen’s analysis is performed iteratively on both the CPU and GPU until a fixed-point is reached. The final points-to information will be available on both the CPU and GPU. The key novelty lies in prioritising, i.e., sorting different types of graph-rewriting rules in a shared worklist \mathcal{W} , so that each side of a CPU-GPU system can always obtain the work from \mathcal{W} that it is the most suitable to process.

At each iteration, both the CPU and GPU calls FETCHANDAPPLY(\mathcal{W}) to fetch the work from \mathcal{W} and then apply appropriate rules to the work obtained until \mathcal{W} is empty. Then both sides exchange only the new points-to and copy edges, $E_{\Delta_{\text{GPU}}}$

and $E_{\Delta_{\text{CPU}}}$, discovered, based on difference propagation. After the constraint graph at each side has been updated, the next iteration begins, if needed.

Section 5.2.1 describes our graph data representation. Section 5.2.2 focuses on CPU-GPU communication. Section 5.2.3 explains how to perform parallel rule applications on CPU and GPU, assisted by our dynamic workload distribution scheme.

5.2.1 Graph Data Representation

Constraint graphs are sparse and their structures change dynamically during Andersen’s analysis. Therefore, selecting an appropriate data structure to store such graphs can have a profound impact on the amount of computations performed on both the CPU and GPU and the amount of data exchanged.

To represent edge sets compactly and support operations on them efficiently, sparse bit vectors and BDDs are popular. BDDs are complex and ill-suited for GPU [28]. Sparse bit vectors are 2X faster than BDDs on CPU [14]. To minimise CPU-GPU communication, we have opted to use sparse bit vectors (as introduced in Section 2.1.2) uniformly on both CPU and GPU.

5.2.2 Managing Communication between CPU and GPU

In heterogeneous CPU-GPU computing, the communication between the two sides can be a major cost. As constraint graphs are sparse and changing during the analysis, it is challenging but important to reduce the communication cost.

In the sequential setting, difference propagation [11, 36, 46] is used to reduce the work of propagating points-to edges in a constraint graph. As shown in Table 2.2, Andersen’s analysis may modify a constraint graph by adding new points-to and copy edges, i.e., new \mathcal{P} and \mathcal{C} edges. We make use of difference propagation (for

the first time) to ensure that at the end of each iteration, the CPU and GPU only need to exchange the new \mathcal{P} and \mathcal{C} edges introduced in that iteration.

Rule	Semantics
$R_{\text{copy}}(x)$	$x \xrightarrow{\mathcal{C}} y \wedge y \xrightarrow{\Delta\mathcal{P}} z \Rightarrow x \xrightarrow{\delta\mathcal{P}} z$
$R_{\text{load}}(x)$	$x \xrightarrow{\mathcal{L}} y \wedge y \xrightarrow{\Delta\mathcal{P}} z \Rightarrow x \xrightarrow{\delta\mathcal{C}} z$
$R_{\text{store}}(x)$	$x \xrightarrow{\Delta\mathcal{P}^{-1}} y \wedge y \xrightarrow{\mathcal{S}} z \Rightarrow x \xrightarrow{\delta\mathcal{C}} z$
$R_{\delta\text{copy}}(x)$	$x \xrightarrow{\Delta\mathcal{C}} y \wedge y \xrightarrow{\mathcal{P}} z \Rightarrow x \xrightarrow{\delta\mathcal{P}} z$
$R_{\text{offset}}(x)$	$x \xrightarrow[\mathcal{O}]{\mathcal{F}} y \wedge y \xrightarrow{\Delta\mathcal{P}} z \Rightarrow x \xrightarrow{\delta\mathcal{P}} z + o$

Table 5.1: Constraint resolution: graph-rewriting rules.

To facilitate concurrent applications of graph-rewriting rules, double buffering is used. Table 5.1 gives the graph-rewriting rules modified from Table 2.2 to support double buffering. In each rule, $\Delta\mathcal{P}$ ($\Delta\mathcal{C}$) in the premise signifies a new points-to (copy) edge produced in the previous iteration and $\delta\mathcal{P}$ ($\delta\mathcal{C}$) in the conclusion signifies a new points-to (copy) edge produced in the current iteration. As before, \mathcal{P} , \mathcal{C} , \mathcal{L} , \mathcal{S} or \mathcal{F} , identifies an edge of that type available at the end of the previous iteration.

In Algorithm 10, $E_{\Delta\text{CPU}}$ ($E_{\Delta\text{GPU}}$) denotes the set of $\Delta\mathcal{P}$ and $\Delta\mathcal{C}$ edges produced in the current iteration on CPU (GPU).

Below we introduce a *reference* CPU-GPU solution of Andersen’s analysis, which has been useful in guiding the development and evaluation of our CPU-GPU solution. Consider the CPU-only and GPU-only implementations of Andersen’s analysis detailed in Section 5.4.1. At this stage, it suffices to know that both proceed essentially by applying the double-buffering-based rules given in Table 5.1 based on exactly the same algorithm. Let w_i be the workload at the i -th iteration. Let t_{CPU}^i and t_{GPU}^i be the analysis times elapsed at the i -th iteration on CPU

and GPU, respectively. By assuming constant work rates for CPU and GPU and zero communication and synchronisation overhead, a reference CPU-GPU solution spends the following analysis time at the i -th iteration:

$$t_{\text{REF}}^i = \frac{w_i}{\frac{w_i}{t_{\text{CPU}}^i} + \frac{w_i}{t_{\text{GPU}}^i}} = \frac{t_{\text{CPU}}^i \times t_{\text{GPU}}^i}{t_{\text{CPU}}^i + t_{\text{GPU}}^i} \quad (5.1)$$

The benefit at the i -th iteration from CPU-GPU computing is:

$$\text{benefit}(i) = \min(t_{\text{CPU}}^i, t_{\text{GPU}}^i) - t_{\text{REF}}^i \quad (5.2)$$

Let us analyse the potential performance gains achieved by performing CPU-GPU communication via difference propagation. Consider a CPU-GPU implementation that always produces the same amount of new points-to and copy edges at both sides at each iteration. Let $d_{\Delta\mathcal{P}\Delta\mathcal{C}}^i$ ($d_{\mathcal{P}\mathcal{C}}^i$) be the set of new (all) points-to and copy edges produced at the i -th iteration, which is half of the same points-to information produced by the CPU- or GPU-only implementation at the i -th iteration. If Host-to-Device and Device-to-Host transfers are concurrent, then the costs, $\text{cost}_{\Delta\mathcal{P}\Delta\mathcal{C}}^i$ and $\text{cost}_{\mathcal{P}\mathcal{C}}^i$, for exchanging $d_{\Delta\mathcal{P}\Delta\mathcal{C}}^i$ and $d_{\mathcal{P}\mathcal{C}}^i$ between the CPU and GPU at the i -th iteration are:

$$\begin{aligned} \text{cost}_{\Delta\mathcal{P}\Delta\mathcal{C}}(i) &= \frac{d_{\Delta\mathcal{P}\Delta\mathcal{C}}^i}{B} + S \\ \text{cost}_{\mathcal{P}\mathcal{C}}(i) &= \frac{d_{\mathcal{P}\mathcal{C}}^i}{B} + S \end{aligned} \quad (5.3)$$

where B and S are the host-to-device bandwidth and the startup cost, respectively, on the CPU-GPU system considered.

Figure 5.1 plots the functions benefit , $\text{cost}_{\Delta\mathcal{P}\Delta\mathcal{C}}$ and $\text{cost}_{\mathcal{P}\mathcal{C}}$ for **svn**, a program in our benchmark suite. In (5.3), $B = 6\text{GB/s}$ and $S = 10\mu\text{s}$ for the CPU-GPU system used in this chapter. The startup cost, taken from [27], is negligible relative to the data transfer times that are between two to three orders of magnitude longer. During the iterations from 20 to 65, the cost of transferring \mathcal{P} and \mathcal{C} edges can be

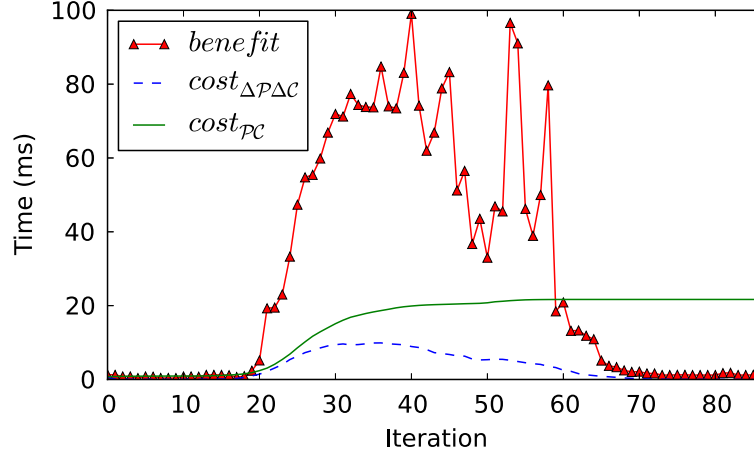


Figure 5.1: A cost-benefit analysis for the `svn` benchmark.

about 1/3 of the benefit while the cost of transferring $\Delta\mathcal{P}$ and $\Delta\mathcal{C}$ edges is moderate. From iteration 65 onwards, the cost of transferring \mathcal{P} and \mathcal{C} edges is overwhelming. Fortunately, the cost of transferring $\Delta\mathcal{P}$ and $\Delta\mathcal{C}$ edges is still no larger than the benefit (even it is small). Therefore, transferring $\Delta\mathcal{P}$ and $\Delta\mathcal{C}$ between the CPU and GPU is important to mitigate the negative impact of communication cost on performance.

5.2.3 Partitioning Computation for CPU and GPU

The objective here is to maximise parallel rule applications on both CPU and GPU at negligible synchronisation overhead. We first describe how to orchestrate the concurrent execution of graph-rewriting rules on CPU and GPU. We then describe how to distribute rule applications dynamically to CPU and GPU to ensure that workload balance is maintained.

Parallel Rule Applications In our sparse representation of a constraint graph, different types of outgoing edges of a node are stored in different sparse bit vectors. Due to double buffering used in the rules given in Table 5.1, different applications of

the same rule can be executed concurrently without synchronisation. In addition, applications of different rules can also be concurrent as long as these rules do not write into the same sparse bit vector storing $\delta\mathcal{P}$ or $\delta\mathcal{C}$.

At the GPU side, every rule application at a node x is executed by a warp as in [28] in a warp-centric manner [19].

At the host side, every rule application is executed sequentially inside a CPU thread. Instead of performing union operations on words, long words are used for efficiency.

The R_{store} rule requires the $\Delta\mathcal{P}^{-1}$ edges, i.e., new pointed-by edges to be stored, which can be space-consuming. We avoid this by adopting the same two-phase strategy as described in Section 2.1.2. As a result, all applications of the R_{store} rule can be executed in parallel without synchronisation.

Dynamic Workload Distribution To accelerate Andersen’s analysis on a CPU-GPU system, it is critically important to minimise workload imbalance between the CPU and GPU. As shown in Algorithm 10, we use a work sharing scheme so that both the CPU and GPU fetch the work to do from a mutex-protected shared worklist, \mathcal{W} , at each iteration.

A simple-minded scheme, referred to as NAIVE, for implementing $\text{FETCHANDAPPLY}(\mathcal{W})$ in Algorithm 10 is given in Algorithm 11. The worklist \mathcal{W} consists of all the N nodes in the constraint graph, as illustrated in Figure 5.2. The CPU and GPU repeatedly fetch a set \mathcal{M} of nodes from the beginning of \mathcal{W} and then apply all the rules to the nodes in \mathcal{M} .

As evaluated later, NAIVE suffers from poor performance due to workload imbalance, because it does not consider the suitability of CPU and GPU for processing different types of rules. As discussed in Section 2.1.2, the GPU is more powerful than the CPU for regular, balanced workloads, but performs more poorly on ir-

regular, imbalanced workloads. In Andersen’s analysis, severe inter-warp workload imbalance can occur when the warps are processing nodes with varying out-degrees of their edges. Figure 5.3 plots the distributions of the sizes of the \mathcal{P} edges and \mathcal{C} edges for *svn*, a program in our benchmark suite, in the *lg-sqrt* format, after Andersen’s analysis is finished. These sizes vary greatly, starting from 0 and approaching 10000. However, the CPU is capable of handling such imbalanced workloads more efficiently.

Algorithm 11 A naive workload distribution scheme.

Procedure FETCHANDAPPLY(\mathcal{W})

begin

```

1   while  $\mathcal{W} \neq \emptyset$  do
2        $\mathcal{M} \leftarrow$  get work from  $\mathcal{W}$ ;
3       APPLY( $\mathcal{C}, \Delta\mathcal{P}, \delta\mathcal{P}, \mathcal{M}$ );
4       APPLY( $\mathcal{L}, \Delta\mathcal{P}, \delta\mathcal{C}, \mathcal{M}$ );
5       APPLY( $\Delta\mathcal{P}^{-1}, \mathcal{S}, \delta\mathcal{C}, \mathcal{M}$ );
6       APPLY( $\mathcal{F}, \Delta\mathcal{P}, \delta\mathcal{P}, \mathcal{M}$ );
7       APPLY( $\Delta\mathcal{C}, \mathcal{P}, \delta\mathcal{P}, \mathcal{M}$ );

```

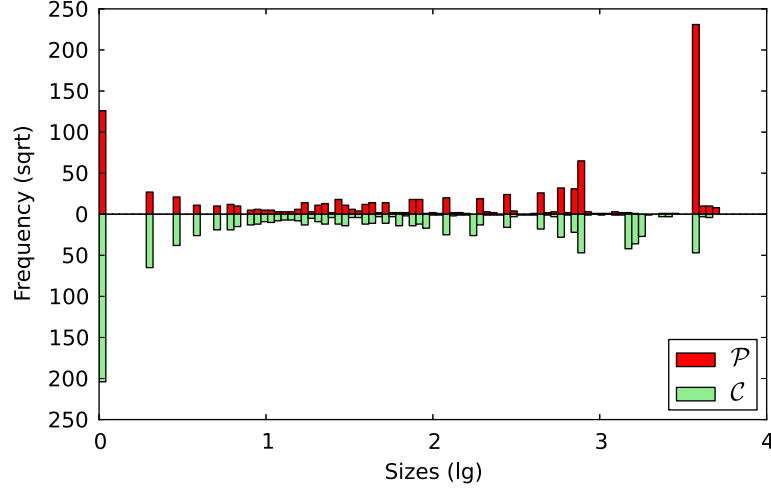
CPU & GPU getFromList



Figure 5.2: The shared worklist \mathcal{W} used in Algorithm 11.

Different applications of the same rule at the same node may induce different workloads at different iterations. However, it is difficult to decide precisely where to execute a rule (on the CPU or GPU) and when, since these workloads are changing in an unpredictable manner during the analysis.

Our key observation is that different types of rules in Andersen’s analysis tend to exhibit different workload characteristics. In general, some rules are more amenable

Figure 5.3: Sizes of points-to and copy edges for `svn`.

to CPU execution while the others fare better on the GPU. It is thus possible to prioritise different types of rules according to the suitability of CPU or GPU for processing them.

Consider the four rules, R_{copy} , R_{load} , R_{offset} and $R_{\delta\text{copy}}$, given in Table 5.1. We will deal with the R_{store} rule differently later. For each of these four rules applied at a node x , three types of outgoing edges, indicated as $t1$, $t2$ and $t3$ in lines 7 – 13 in Algorithm 1, are accessed. In Table 5.1, $t1$ and $t2$ appear in the premise of a rule and $t3$ in its conclusion.

For each rule R applied at node x in lines 7 – 13 in Algorithm 1, the amount of work performed is dictated by $|E_{t1}(x)| \times |E_{t2}(y)|$, where $t1$ and $t2$ indicate the types of edges processed in the premise of the rule. We write DI_R to represent the *degree of imbalance* for the workloads performed by applying rule R at all the possible nodes in the constraint graph, measured by how $|E_{t1}(x)| \times |E_{t2}(y)|$ varies across these nodes (using, for example, its standard derivation).

We propose to use the DI_R of rule R to determine the suitability of the CPU or GPU for applying the rule. The higher DI_R is, the more (less) suitable the CPU

(GPU) is for applying rule R . According to the workload characteristics of the four rules, R_{copy} , R_{load} , R_{offset} and $R_{\delta\text{copy}}$, we have:

$$DI_{R_{\delta\text{copy}}} > DI_{R_{\text{copy}}} > DI_{R_{\text{load}}} > DI_{R_{\text{offset}}} \quad (5.4)$$

We observe that the larger $\max_{v \in V} |E_T(v)|$ is for a particular type of edges in a constraint graph, the greater $|E_T(v)|$ varies across different nodes in V . In general, $|E_{\mathcal{F}}(v)|$ is much smaller than $\min(|E_{\Delta\mathcal{P}}(v)|, |E_{\Delta\mathcal{C}}(v)|)$ and $|E_{\mathcal{C}}(v)|$, $|E_{\mathcal{L}}(v)|$ and $|E_{\mathcal{P}}(v)|$ are approximately an order of magnitude larger than $\max(|E_{\Delta\mathcal{P}}(v)|, |E_{\Delta\mathcal{C}}(v)|)$. Therefore, $DI_{R_{\text{offset}}}$ is the smallest. While being close to $|E_{\mathcal{L}}(v)|$ when the constraint graph is initialised, $|E_{\mathcal{P}}(v)|$ and $|E_{\mathcal{C}}(v)|$ can increase dramatically during the analysis. So $DI_{R_{\text{load}}}$ is the second smallest. In general, $|E_{\mathcal{P}}(v)|$ is much larger than $|E_{\mathcal{C}}(v)|$, as indicated in Figure 5.3. Therefore, we have $DI_{R_{\delta\text{copy}}} > DI_{R_{\text{copy}}}$.

As the $\Delta\mathcal{P}^{-1}$ edges are not stored, we deal with the R_{store} rule differently. As discussed before, a separate worklist, $\mathcal{W}_{R_{\text{store}}}$, is maintained from which the set $V_{R_{\text{store}}}$ of nodes that require the R_{store} rule to be applied are obtained. Let $DI_{R_{\text{store}}} = \max\{|E_{\Delta\mathcal{P}^{-1}}(x)| \mid x \in V_{R_{\text{store}}}\}$ be used to approximate the degree of imbalance for this rule. Of course, the $\Delta\mathcal{P}^{-1}$ edges, which are not stored, are deduced from the $\Delta\mathcal{P}$ edges. At each iteration, we decide dynamically whether the CPU or GPU is more suitable to apply the R_{store} rule to the nodes in $V_{R_{\text{store}}}$. Our simple heuristic is to select the CPU if and only if $DI_{R_{\text{store}}} \geq \tau$, which is set as 20 empirically.

In our CPU-GPU solution of Andersen's analysis, `FETCHANDAPPLY(\mathcal{W})` in Algorithm 10 is given in Algorithm 12. A so-called DI-based dynamic workload distribution scheme, referred to as IDD, is therefore adopted. At an iteration, either the CPU or GPU applies the R_{store} rule to all the nodes in $V_{R_{\text{store}}}$ depending on whether $DI_{R_{\text{store}}} \geq \tau$ holds or not. As for the other four rules, R_{copy} , R_{load} , R_{offset}

and $R_{\delta\text{copy}}$, the shared worklist \mathcal{W} , which is described below, maintains all rule applications to be executed at any iteration. The CPU and GPU repeatedly fetch a set \mathcal{M} of nodes from the worklist \mathcal{W} and apply appropriate rules to the nodes in \mathcal{M} .

Algorithm 12 A DI-based dynamic workload distribution.

Procedure FETCHANDAPPLY(\mathcal{W})

begin

```

1  APPLY( $\Delta\mathcal{P}^{-1}$ ,  $\mathcal{S}$ ,  $\delta\mathcal{C}$ ,  $\mathcal{W}_{R_{\text{store}}}$ ); //  $R_{\text{store}}$ 
2  while  $\mathcal{W} \neq \emptyset$  do
3       $(\mathcal{M}, r) \leftarrow$  get work from  $\mathcal{W}$ ;
4      if  $r = R_{\text{copy}}$  then    APPLY( $\mathcal{C}$ ,  $\Delta\mathcal{P}$ ,  $\delta\mathcal{P}$ ,  $\mathcal{M}$ );
5      if  $r = R_{\text{load}}$  then   APPLY( $\mathcal{L}$ ,  $\Delta\mathcal{P}$ ,  $\delta\mathcal{C}$ ,  $\mathcal{M}$ );
6      if  $r = R_{\text{offset}}$  then APPLY( $\mathcal{F}$ ,  $\Delta\mathcal{P}$ ,  $\delta\mathcal{P}$ ,  $\mathcal{M}$ );
7      if  $r = R_{\delta\text{copy}}$  then APPLY( $\Delta\mathcal{C}$ ,  $\mathcal{P}$ ,  $\delta\mathcal{P}$ ,  $\mathcal{M}$ );

```

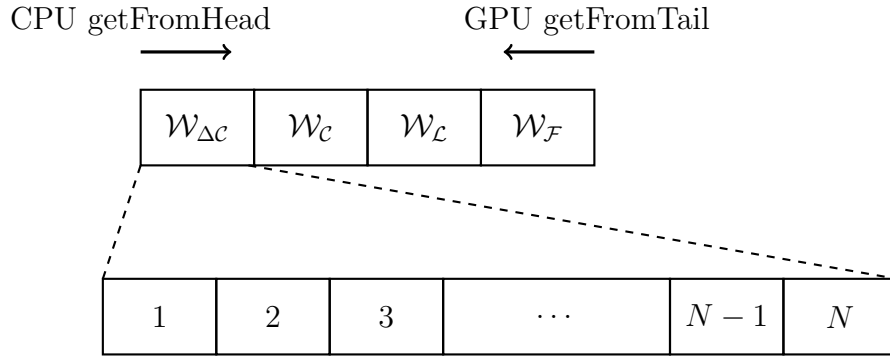


Figure 5.4: The shared worklist \mathcal{W} used in Algorithm 12.

Based on (5.4), our shared worklist, \mathcal{W} , illustrated in Figure 5.4, consists of four sub-worklists for rules $R_{\delta\text{copy}}$, R_{copy} , R_{load} and R_{offset} , sorted in decreasing order of their degrees of imbalance. Each sub-worklist consists of all the nodes in the constraint graph as in NAIVE, shown in Figure 5.2. The CPU and GPU will fetch the work from the two opposite sides. This ensures that each side of the system will always apply rules in decreasing order of its suitability for processing these

rules. As a result, IDD achieves better load balance than NAIVE.

While concurrent applications of the same rule in Table 5.1 can be synchronisation-free, concurrent applications of different rules may require synchronisation when they may have write-write races (e.g., between R_{copy} and $R_{\delta\text{copy}}$). To avoid such synchronisation altogether on CPU or GPU, we have introduced two barriers, one between $\mathcal{W}_{\Delta\mathcal{C}}$ and $\mathcal{W}_{\mathcal{C}}$ and one between $\mathcal{W}_{\mathcal{C}}$ and $\mathcal{W}_{\mathcal{F}}$. For this reason, a call to `FETCHANDAPPLY(\mathcal{W})` in Algorithm 12 always returns a set \mathcal{M} of nodes associated with the same rule type, r .

5.3 Optimisations

We describe several optimisations to further improve the performance of our CPU-GPU solution of Andersen’s analysis.

5.3.1 Optimisation I: On Hiding Communication Overhead

In Algorithm 10, $E_{\Delta\text{CPU}}$ is the union of $E_{\Delta\text{CPU}}^{\Delta\mathcal{P}}$ and $E_{\Delta\text{CPU}}^{\Delta\mathcal{C}}$, which are the sets of $\Delta\mathcal{P}$ and $\Delta\mathcal{C}$ edges produced on CPU, respectively. $E_{\Delta\text{GPU}}$ is similarly decomposed into $E_{\Delta\text{GPU}}^{\Delta\mathcal{P}}$ and $E_{\Delta\text{GPU}}^{\Delta\mathcal{C}}$ on GPU. Concurrent Host-to-Device and Device-to-Host transfers are used to maximise their overlap. To further reduce communication cost, computation-communication overlap is employed at each iteration. As $|E_{\Delta\text{CPU}}^{\Delta\mathcal{P}}| > |E_{\Delta\text{CPU}}^{\Delta\mathcal{C}}|$ and $|E_{\Delta\text{GPU}}^{\Delta\mathcal{P}}| > |E_{\Delta\text{GPU}}^{\Delta\mathcal{C}}|$ usually, the $\Delta\mathcal{C}$ edges are exchanged before the $\Delta\mathcal{P}$ edges between the CPU and GPU. As soon as one side has received the $\Delta\mathcal{C}$ edges from the other, the operations indicated in lines 6 – 7 are performed on $\Delta\mathcal{C}$, by overlapping with the transfer of the $\Delta\mathcal{P}$ edges.

5.3.2 Optimisation II: On ΔP -Equivalence and R_{store}

ΔP -equivalent variables have the same set of outgoing ΔP edges in the current iteration [28]. For example, if $E_{\Delta P}(x) = E_{\Delta P}(y)$, then applying $R_{\text{copy}}(z)$, where $E_C(z) = \{x, y\}$, yields the same result for z if either $E_{\Delta P}(x)$ or $E_{\Delta P}(y)$ is used.

Work on identifying (1) ΔP -equivalent variables [28] and (2) the set $V_{R_{\text{store}}}$ of variables where the R_{store} rule is applied (the first phase of this rule application as discussed in Section 5.2.3) is expensive on CPU, costing over 3X more than on GPU. Therefore, such computations are performed on the GPU, with the results transferred to the CPU.

5.3.3 Optimisation III: On Adaptive Heterogeneity

As illustrated in Figure 5.1, the performance benefit of a CPU-GPU solution of Andersen’s analysis is more than offset by the CPU-GPU communication cost incurred during the first and last few iterations. Therefore, an adaptive scheme is used. When the cost exceeds the benefit, the faster of the two, CPU and GPU, will perform the iteration alone. For the first and last few iterations, the workloads of rule applications are small with negligible imbalance. The GPU is more suitable and thus preferred. Therefore, our CPU-GPU solution begins in the GPU-alone mode, switches to the heterogeneous mode when $t_{\text{tran}} \leq t_{\text{comp}} * \alpha$, and returns to the GPU-alone mode again when $t_{\text{tran}} > t_{\text{comp}} * \alpha$. Empirically, $\alpha = 0.2$ is used.

5.4 Evaluation

We show that our parallel solution of Andersen’s analysis on a CPU-GPU system achieves better average speedups than CPU-only and GPU-only solutions for a set of seven C benchmarks considered. Even if the better of the speedups from

CPU-only and GPU-only solutions is selected for each benchmark, our CPU-GPU solution remains faster on average.

For the CPU-GPU system used in our experiments, the host (running 64-bit Ubuntu 12.04) is equipped with two eight-core 2.00GHz Intel Xeon E5-2650 CPUs with 62GB of RAM. Each core has a 64KB L1 cache and a 256KB L2 cache. Each CPU has a 20MB L3 unified cache shared by its eight cores. The code for the host is written in C++ using POSIX threads and compiled under “GCC -O3”. The GPU used is a 0.71GHz NVIDIA Tesla K20c GPU with 13 SMXs, each containing 192 cores. Each SMX has a 64KB of on-chip memory configured as 48 KB of shared memory and 16 KB of L1 cache. All SMXs share a 1280KB L2 cache. The CUDA code is compiled under “nvcc -arch=sm_30” (v5.0).

Table 5.2 lists the seven C benchmarks used, with the number of variables ranging from 53K to 559K and the number of statements ranging from 55K to 560K. Note that every SCC formed by copy edges is collapsed as its variables have the same points-to edges.

Benchmark	#Variables	#Statements
perl	53,362	55,977
python	92,599	92,827
svn	107,708	122,558
gcc	120,870	127,171
gdb	232,814	198,933
vim	246,944	89,226
gimp	558,867	565,655

Table 5.2: Benchmark suite: sizes of initial constraint graphs.

5.4.1 Implementations

Our CPU-GPU solution is implemented as follows:

- For GPU kernels, we use 13 thread blocks (for 13 SMXs), 864 threads per block for R_{copy} , R_{load} , R_{store} and $R_{\delta\text{copy}}$, and 1024 threads for R_{offset} . This represents the same configuration as in [28], limited by available shared memory (48KB per block) and maximum number of threads per block (1024).
- As for the host, there are 16 compute threads, one per core, and two control threads. The GPU-control thread is responsible for fetching the work from a shared worklist for the GPU to do, launching kernel execution and exchanging $\Delta\mathcal{P}$ and $\Delta\mathcal{C}$ between the CPU and GPU. The CPU-control thread oversees and coordinates the execution of the 16 compute threads.

The 16 CPU compute threads and the GPU-control thread will fetch the work from a mutex-protected shared worklist, illustrated in Figure 5.4. The granularity of each fetch is determined empirically. For a CPU thread, a chunk of 128 nodes appears to be a good choice. As for the GPU, any value in the range 1K – 8K is adequate, even when the GPU happens to get the last chunk from the shared worklist. So 8K is used.

The CPU-only (GPU-only) solution is derived from our CPU-GPU solution, by ignoring the GPU-control (CPU-control) thread, so that the entire analysis is now performed on the CPU (GPU) alone, without online cycle elimination. The GPU-only solution is similar to the state-of-the-art GPU implementation introduced in [28].

The CPU-only solution is faster than the CPU implementation introduced in [29] (both without online cycle elimination), based on the experimental results given in [28], since our CPU-only solution is able to apply graph-rewriting rules in parallel without synchronisation.

The CPU-only implementation is faster than the GPU-only implementation, although the latter is highly optimised. This GPU-only implementation is almost

the same as the implementation from [28] where sophisticated optimisations, e.g., using shared memory, have been performed. The only difference is that online cycle elimination is not performed for the GPU-only and CPU-only implementations, which are the baseline for evaluation here. With less cycles eliminated, some particularly long-running tasks are more likely to occur, and consequently the load imbalance issue for GPU becomes more severe.

5.4.2 Speedups

Figure 5.5 compares the speedups of our CPU-GPU solution of Andersen’s analysis against the CPU-only and GPU-only solutions (normalised to GPU-only). For each benchmark, the left bar is for CPU-only, the middle bar for GPU-only and the right bar for our solution. Each of our speedup bars is shown as a breakdown of five components, contributed by (1) NAIVE (the naive workload distribution given in Algorithm 11), (2) IDD (our dynamic workload distribution scheme given in Algorithm 12), (3) Opt I, (4) Opts I + II, and (5) Opts I + II + III, where the three optimisations are described in Section 5.3.

We observe that the performance ratios of CPU-only over GPU-only vary wildly across these benchmarks. It is therefore not easy to decide which of the two analyses to use for a given program. However, our CPU-GPU solution outperforms (1) CPU-only by 50.6%, (2) GPU-only by 78.5%, and (3) an oracle that behaves as the faster of (1) and (2) for each benchmark by 34.6% on average. In addition, our solution is faster than the oracle for six benchmarks. The only exception is `gcc`, for which our solution is slightly better than CPU-only but a lot worse than GPU-only. There are two main reasons behind. First, `gcc` induces fewer new points-to edges than the other benchmarks during the analysis. The overhead incurred in CPU-GPU communication and workload distribution is relatively high. Second, the

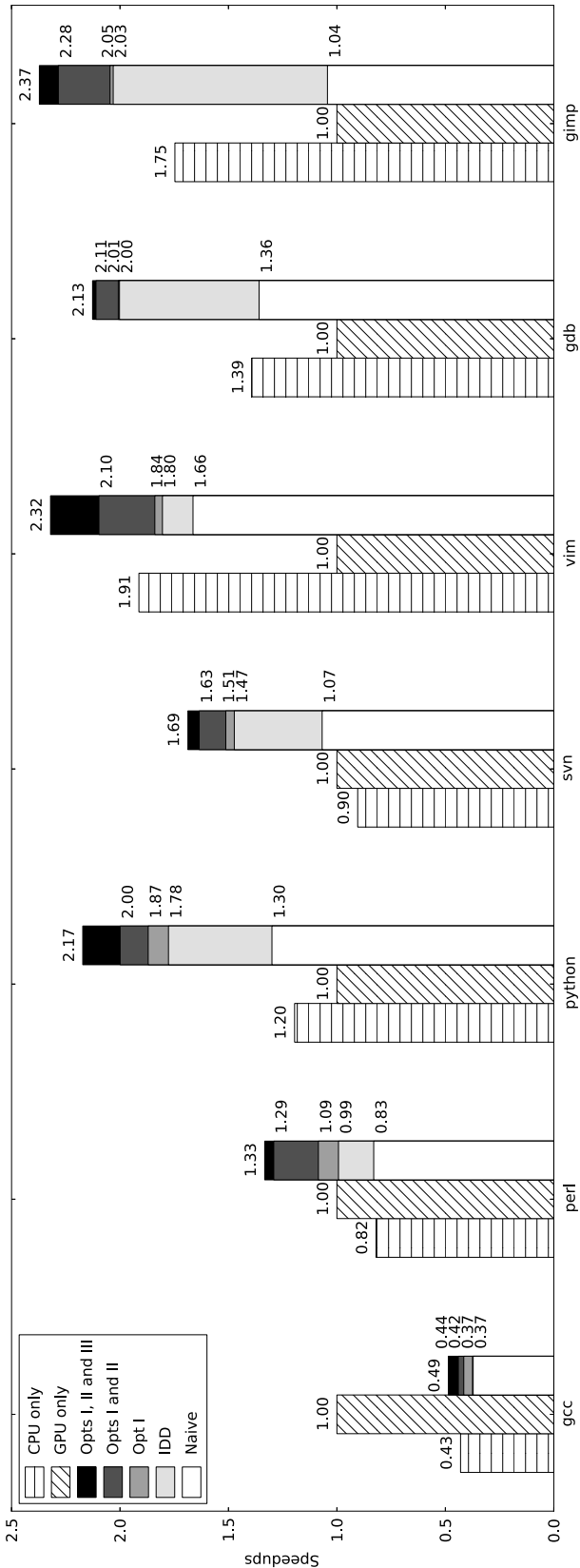


Figure 5.5: Speedups of our parallel Andersen’s analysis and their CPU-only and GPU-only versions (normalised to GPU-only).

performance gap between GPU-only and CPU-only, 2.3X, is the highest among all the benchmarks (with `vim` coming as the second highest). Thus, workload balance is relatively hard to achieve.

5.4.3 Dynamic Workload Balancing

Our dynamic workload distribution scheme, IDD, has succeeded in accelerating Andersen’s analysis further on top of NAIVE for all benchmarks. The performance improvements are substantial in `python`, `svn`, `gdb` and `gimp`.

Benchmark	IDD workload distribution					
	$R_{R_{\text{offset}}}$	$R_{R_{\text{load}}}$	$R_{R_{\text{copy}}}$	$R_{R_{\text{copy}}}$	S_{GPU} (%)	$R_{R_{\text{store}}}$
perl	0.07	0.17	0.53	1.62	31.0	7.18
python	0.04	0.15	0.70	3.63	37.1	4.52
svn	0.04	0.13	0.61	2.41	34.2	7.29
gcc	0.09	0.40	0.57	0.61	81.6	1.21
gdb	0.04	0.28	0.90	3.31	13.7	5.35
vim	0.08	0.50	0.64	5.59	100.0	0.19
gimp	0.07	0.56	0.64	3.58	77.6	0.87

Table 5.3: Analysis of our CPU-GPU solution (including its key method employed).

To understand why IDD is effective, some statistics are given in Table 5.3. R_x , where x is one of the five rules given in Table 5.1, represents the ratio of the time spent by GPU-only over CPU-only on applying Rule x (accumulated in all iterations). $R_{R_{\text{offset}}} < R_{R_{\text{load}}} < R_{R_{\text{copy}}} < R_{R_{\delta\text{copy}}}$ holds across the seven benchmarks. This justifies the priorities assigned to the four rules in (5.4) on CPU and GPU.

Let us analyse `gcc` and `vim` to see why performing Andersen’s analysis on both CPU and GPU is the least beneficial among the seven benchmarks. For `gcc`, $R_{R_{\text{offset}}}$, $R_{R_{\text{load}}}$, $R_{R_{\text{copy}}}$ and $R_{R_{\delta\text{copy}}}$ are smaller than 1, indicating that the GPU is more suitable than the CPU for applying these rules. For `vim`, speedup is limited for

a different reason. The $R_{\delta\text{copy}}$ rule is particularly expensive to apply, consuming 30.3% (83.0%) of the analysis time of CPU-only (GPU-only). As a result, the GPU is rather inefficient for the entire analysis. As discussed in Section 5.4.1, the granularities of workloads fetched from the shared worklist by a CPU thread and a GPU kernel are 128 and 8K nodes, respectively. The GPU stalls for 8.5% of its analysis time, waiting for the CPU to finish. The ratio can be lowered if the granularities are reduced. However, the overall performance even worsens due to less efficient GPU kernel execution and more synchronisation overhead incurred.

Let us look at the effectiveness of the heuristic $DI_{R_{\text{store}}} \geq \tau$ used in determining where to execute the R_{store} rule, on CPU or GPU, in line 1 of Algorithm 12. In Table 5.3, S_{GPU} stands for the percentage of iterations that the R_{store} rule is executed on the GPU. The larger $R_{R_{\text{store}}}$ is (i.e., the slower the GPU is than the CPU in applying the rule), the smaller S_{GPU} is (the fewer iterations that the GPU will be asked to execute the rule). For `vim`, where $S_{GPU} = 100\%$, the R_{store} rule is always applied on the GPU, which is much more efficient than the CPU (with $R_{R_{\text{store}}} = 0.19$). As the number of stores in a program is small, the benefit of adaptively determining where to execute the R_{store} rule is small. Nevertheless, setting τ to 20 still delivers a speedup of 3.3% (2.9%) compared to when the rule is applied on CPU (GPU) exclusively.

5.4.4 Optimisations

For the three optimisations described in Section 5.3, their effects on performance are shown in Figure 5.5. We analyse them using the statistics given in Table 5.4.

Opt I, which overlaps communication with computation, is the most beneficial for `gcc` and `perl` but the least for `gdb` and `gimp`. Its effectiveness depends on the degree of overlap between (1) the process of exchanging their respective $\Delta\mathcal{P}$ sets

Benchmark	Opt I		Opt II	Opt III	S_{REF}
	$O_{\Delta\mathcal{P}}$ (%)	$O_{\Delta\mathcal{C}}$ (%)	O_H (%)	O_I (%)	
perl	8.1	10.6	9.3	21.1	1.31
python	13.2	9.4	3.9	36.2	1.40
svn	10.5	7.4	5.5	13.6	1.20
gcc	15.2	16.5	14.6	21.8	0.48
gdb	12.3	5.0	5.2	15.0	1.10
vim	15.5	9.7	6.7	34.9	1.18
gimp	11.3	5.3	8.8	27.6	1.03

Table 5.4: Analysis of our CPU-GPU solution (including its key method and optimisations employed).

between the CPU and GPU and (2) the computations performed on the local and remote $\Delta\mathcal{C}$ sets on both the CPU and GPU. In Table 5.4, $O_{\Delta\mathcal{P}}$ and $O_{\Delta\mathcal{C}}$ represent the times elapsed on performing (1) and (2) in percentage, respectively, over the total analysis time for a benchmark. The transfer times for $\Delta\mathcal{P}$ are completely hidden for `gcc` and `perl` since $O_{\Delta\mathcal{P}} < O_{\Delta\mathcal{C}}$ for each benchmark, but only hidden by less than 50% for `gdb` and `gimp` since $O_{\Delta\mathcal{P}} > 2 \times O_{\Delta\mathcal{C}}$ for each benchmark. The (unhidden) communication cost is 8.4% on average, with `gcc` reaching 16.9%, since it induces fewer edges than the other benchmarks during Andersen’s analysis.

Opt II, which relies on the GPU to identify $\Delta\mathcal{P}$ -equivalent variables and the variables for which the R_{store} rule should be applied, is generally more effective than Opt I. In Table 5.4, O_H represents the time spent (in percentage) on these computations over the total analysis time for a benchmark. Opt II is the least effective for `python`, `svn` and `gdb` because the values of O_H for these benchmarks, 3.9%, 5.5% and 5.2%, are small.

Opt III, which decides adaptively whether to perform an iteration of Andersen’s analysis on CPU or GPU or both, is profitable for all the benchmarks except `gdb`. In Table 5.4, O_I represents the percentage of iterations executed on the GPU alone.

For `gdb`, $O_I = 15\%$. By offloading this much of the total analysis to the GPU, the potential performance benefit obtained may not outweigh the cost incurred. A similar problem exists for `svn`, where $O_I = 13.6\%$.

5.4.5 Overall Effectiveness

We discuss the effectiveness of our CPU-GPU solution with respect to the reference CPU-GPU solution with its analysis time (5.1) derivable from those of CPU-only and GPU-only solutions. In Table 5.4, S_{REF} represents the speedup of our solution over this reference. Our solution outperforms the reference in six benchmarks with an average speedup of 1.1X. The exception is `gcc` again, for the reasons discussed above.

These results demonstrate the effectiveness of our solution. By dispatching graph-rewriting rules to the “better side” of a CPU-GPU system to apply, our solution performs better than the reference, for which even zero communication and synchronisation overhead has been assumed.

5.5 Related Work

While there are no heterogeneous CPU-GPU solutions to pointer analysis, a lot of efforts have been made on parallelising graph algorithms on CPU-GPU systems, where the structure of the underlying graph is not modified. In [20], the best from a few implementations of BFS is selected dynamically for each level of the BFS algorithm. Later, workload-aware and fixed-partitioned-space strategies [30] are considered for BFS. In [12], a graph programming model, Totem, is presented and applied to two applications, BFS and PageRank. In [13], workload distribution is studied in the Totem framework, by distinguishing workloads in terms of node

degrees, so that the suitability of CPU or GPU for the workloads can be estimated.

However, these techniques cannot be directly applied to Andersen’s analysis since the graph changes dynamically in an unpredictable manner during the analysis. Our focus in this chapter is to introduce a new dynamic workload distribution scheme for Andersen’s analysis to prioritise workloads to CPU or GPU that is better suited for processing them.

5.6 Chapter Summary

This chapter describes the first parallel implementation of Andersen’s analysis on a CPU-GPU system. The presence of dynamic and unpredictable modifications to a constraint graph makes it difficult to balance workloads between CPU and GPU. The sparsity of a constraint graph posts obstacles in engineering efficient CPU-GPU communication. To overcome these two challenges, we distribute graph-rewriting rules to the CPU or GPU that is better suited for processing the rules and adopt difference propagation of points-to information between the CPU and GPU to reduce the communication cost. On a set of seven C programs evaluated, our CPU-GPU solution outperforms on average the variants of state-of-the-art CPU-only and GPU-only implementations.

Chapter 6

Conclusions and Future Work

This chapter firstly concludes the three solutions that explore different parallel platforms for accelerating pointer analysis, and then discusses future work.

6.1 Conclusions

Making the highly important pointer analysis more efficient and scalable especially for large program has been a challenging task. Over the years, great efforts have been made in the sequential setting on accelerating both the whole-program analysis and demand-driven analysis. However, it is still difficult to perform precise analysis in an efficient manner. In recent years, with the ubiquity of parallel platforms, including multicore CPUs and GPUs, boosting the performance of pointer analysis by exploring parallelisation techniques has become increasingly popular.

There have been a number of attempts on parallelising pointer analysis algorithms for analysing C or Java programs on multi-core CPUs and/or GPUs [8, 28, 29, 31, 32, 40]. As compared in Table 6.1, all these parallel solutions are different forms of Andersen’s whole-program pointer analysis [2] with varying precision considerations in terms of context-, flow- and field-sensitivity.

Analysis	Algorithm	Sensitivity (Precision)			Application	Platform
		Context	Field	Flow		
[29]	Whole-Program	✗	✓	✗	C	CPU
[28]	Whole-Program	✗	✓	✗	C	GPU
[8]	Whole-Program	✗	✗	✓*	Java	CPU
[40]	Whole-Program	✓	✗	✗	C	CPU
[31]	Whole-Program	✗	✓	✓	C	CPU
[32]	Whole-Program	✗	✓	✓	C	GPU
Chapter 3	Demand-Driven	✓	✓	✗	Java	CPU
Chapter 4	Whole-Program	✗	✓	✗	C	GPU
Chapter 5	Whole-Program	✗	✓	✗	C	CPU-GPU

*: Partial flow-sensitivity without performing strong updates

Table 6.1: Comparing different parallel pointer analysis.

In this thesis, we have proposed three solutions to accelerate pointer analysis on three different parallel platforms: multicore CPUs, GPU and heterogeneous CPU-GPU systems.

Chapter 3 has presented the first parallel implementation of pointer analysis with CFL reachability, a basis for supporting demand queries in compiler optimisation and software engineering. Formulated as a graph traversal problem (often with context- and field-sensitivity for desired precision) and driven by queries (issued often in batch mode), this analysis is non-trivial to parallelise. We introduced a parallel solution to the CFL-reachability-based pointer analysis, with context- and field-sensitivity. We exploited its inherent parallelism by avoiding redundant graph traversals with two novel techniques, data sharing and query scheduling. With data sharing, paths discovered in answering a query are recorded as shortcuts so that subsequent queries will take the shortcuts instead of re-traversing its associated paths. With query scheduling, queries are prioritised according to their statically estimated dependences so that more redundant traversals can be further avoided. Evaluated using a set of 20 Java programs, our parallel implementation of CFL-reachability-based pointer analysis achieves an average speedup of 16.2X over a state-of-the-art sequential implementation on 16 CPU cores.

Chapter 4 has proposed an efficient GPU implementation of Andersen’s whole-program inclusion-based pointer analysis, a fundamental analysis on which many others are based, including optimising compilers, bug detection and security analyses. Andersen’s algorithm makes extensive modifications to the graph that represents the pointer-manipulating statements in a program. These modifications are highly irregular, input-dependent and statically unpredictable, making it much more challenging to balance such graph workloads across a multitude of GPU cores than those dealt with by traditional graph algorithms such as DFS and BFS. To

parallelise Andersen’s analysis efficiently on GPUs, we introduced an imbalance-aware workload partitioning scheme that divides its workload dynamically among the concurrent warps, initially in a warp-centric manner (during the coarse-grain stage) but later switches to a task-pool-based model when a workload imbalance is detected (during the fine-grain stage). We have improved further its performance by using an adaptive group propagation scheme to reduce some redundant traversals. For a set of 14 C benchmarks evaluated, our parallel implementation of Andersen’s analysis achieves a significant speedup of 46% on average over the state-of-the art on an NVIDIA Tesla K20c GPU.

Chapter 5 has described the first implementation of Andersen’s inclusion-based pointer analysis for C programs on a heterogeneous CPU-GPU system, where both its CPU and GPU cores are used. Existing parallel solutions to Andersen’s analysis run on either the CPU or GPU but not both, rendering the underlying computational resources underutilised and the ratios of CPU-only over GPU-only speedups for certain programs (i.e., graphs) unpredictable. We observed that a naive parallel solution of Andersen’s analysis on a CPU-GPU system suffers from poor performance due to workload imbalance. We have introduced a solution that is centered around a new dynamic workload distribution scheme. The novelty lies in prioritising the distribution of different types of workloads, i.e., graph-rewriting rules in Andersen’s analysis to CPU or GPU according to the degrees of the processing unit’s suitability for processing them. This scheme is effective when combined with synchronisation-free execution of tasks (i.e., graph-rewriting rules) and difference propagation of points-to information between the CPU and GPU. For a set of seven C benchmarks evaluated, our CPU-GPU solution outperforms (on average) (1) the CPU-only solution by 50.6%, (2) the GPU-only solution by 78.5%, and (3) an oracle solution that behaves as the faster of (1) and (2) on every benchmark by

34.6%. Note that the performance of the GPU-only implementation is worse than that of the CPU-only implementation. This is mainly because online cycle elimination is not adopted (for both the CPU-only and GPU-only implementations), and the load imbalance issue for GPU is more severe, as explained in Section 5.4.1. Chapter 4 mainly targets the load imbalance issue for GPU, and has achieved an average speedup of 46% over the state-of-the-art GPU implementation [28], where online cycle elimination has been performed.

6.2 Future Work

The parallel solution in Chapter 3 explores the abundant inter-query parallelism. It has achieved significant performance speedups over a state-of-the-art sequential implementation for the CFL-reachability-based analysis. Some queries, however, can be extremely time consuming, still resulting in load imbalance among concurrent threads. Exploiting intra-query parallelism can be an effective way to alleviate such an issue. However, it is a challenge to minimise the cost for such fine grained parallelism.

The GPU implementation in Chapter 4 has demonstrated considerable speedups over the state-of-the-art parallel solutions (on both CPUs [29] and GPUs [28]) for Andersen’s inclusion-based analysis. The heterogeneous CPU-GPU implementation in Chapter 5, however, does not adopt online cycle elimination due to obstacles in collapsing cycles on the fly on different memory spaces. It would be promising to perform parallel online cycle elimination. Another challenge of performing parallel pointer analysis on different memory spaces is to minimise communications. With these two problems solved, it would be feasible to accelerate pointer analysis on multi-GPUs and heterogeneous CPU-GPU systems in a scalable manner.

Unsound pointer analysis, which deliberately makes no attempt to consider all possible data flow in a program, is practical for some particular clients. Some constructs in the program are ignored, not because they are unimportant, but simply because they are too hard. However, unsound pointer analysis can be utilised by a range of clients that support IDE services [22]. It is interesting and promising to parallelise unsound pointer analysis, where a trade-off between unsoundness and parallelism could be made.

Our parallel solutions can be applied to many important analyses in software engineering that rely on these two styles of fundamental pointer analyses. For example, program slicing [47], interprocedural SSA analysis [5], more precise pointer analysis [15, 32, 55, 67], bug detection [53, 54, 66], and program behaviour prediction [69] all utilise Andersen’s analysis as their pre-analysis. Their performance would directly benefit from the techniques proposed in this thesis, with our techniques appropriated extended in future.

Bibliography

- [1] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader. Scalable graph exploration on multicore processors. In *SC*, pages 1–11, 2010.
- [2] L. O. Andersen. Program analysis and specialization for the C programming language. *PhD Thesis, DIKU, University of Copenhagen*, 1994.
- [3] J. Barnat, P. Bauch, L. Brim, and M. Ceska. Computing strongly connected components in parallel on CUDA. In *IPDPS*, pages 544–555, 2011.
- [4] M. Berndt, O. Lhoták, F. Qian, L. Hendren, and N. Umanee. Points-to analysis using BDDs. In *PLDI*, pages 103–114, 2003.
- [5] S. Calman and J. Zhu. Increasing the scope and resolution of interprocedural static single assignment. In *SAS*, pages 154–170, 2009.
- [6] W. Chang, B. Streiff, and C. Lin. Efficient and extensible security enforcement using dynamic data flow analysis. In *CCS*, pages 39–50, 2008.
- [7] M. Das, B. Liblit, M. Fähndrich, and J. Rehof. Estimating the impact of scalable pointer analysis on optimization. In *SAS*, pages 260–278, 2001.
- [8] M. Edvinsson, J. Lundberg, and W. Lówe. Parallel points-to analysis for multi-core machines. In *HiPEAC*, pages 45–54, 2011.

-
- [9] D. Evans. Static detection of dynamic memory errors. In *PLDI*, pages 44–53, 1996.
 - [10] M. Fáhndrich, J. S. Foster, Z. Su, and A. Aiken. Partial online cycle elimination in inclusion constraint graphs. In *PLDI*, pages 85–96, 1998.
 - [11] C. Fecht and H. Seidl. Propagating differences: An efficient new fixpoint algorithm for distributive constraint systems. In *ESOP*, pages 90–104, 1998.
 - [12] A. Gharaibeh, L. Beltrão Costa, E. Santos-Neto, and M. Ripeanu. A yoke of oxen and a thousand chickens for heavy lifting graph processing. In *PACT*, pages 345–354, 2012.
 - [13] A. Gharaibeh, L. B. Costa, E. Santos-Neto, and M. Ripeanu. On graphs, GPUs, and blind dating - a workload to processor matchmaking quest. In *IPDPS*, pages 851–862, 2013.
 - [14] B. Hardekopf and C. Lin. The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code. In *PLDI*, pages 290–299, 2007.
 - [15] B. Hardekopf and C. Lin. Flow-sensitive pointer analysis for millions of lines of code. In *CGO*, pages 289–298, 2011.
 - [16] P. Harish and P. J. Narayanan. Accelerating large graph algorithms on the GPU using CUDA. In *HiPC*, pages 197–208, 2007.
 - [17] N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using cla: A million lines of C code in a second. In *CGO*, pages 254–263, 2001.
 - [18] M. Hind. Pointer analysis: haven’t we solved this problem yet? In *PASTE*, pages 54–61, 2001.

-
- [19] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun. Accelerating CUDA graph algorithms at maximum warp. In *PPoPP*, pages 267–276, 2011.
 - [20] S. Hong, T. Oguntebi, and K. Olukotun. Efficient parallel graph exploration on multi-core CPU and GPU. In *PACT*, pages 78–88, 2011.
 - [21] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey. Debunking the 100x GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *ISCA*, pages 451–460, 2010.
 - [22] O. Lhoták, S. Yannis, and M. Sridharan. Pointer analysis. *Report from Dagstuhl Seminar*, 2013.
 - [23] L. Li, C. Cifuentes, and N. Keynes. Boosting the performance of flow-sensitive points-to analysis using value flow. In *FSE*, pages 343–353, 2011.
 - [24] L. Li, C. Cifuentes, and N. Keynes. Precise and scalable context-sensitive pointer analysis via value flow graph. In *ISMM*, pages 85–96, 2013.
 - [25] Y. Li, T. Tan, Y. Sui, and J. Xue. Self-inferencing reflection resolution for Java. In *ECOOP*, pages 27–53, 2014.
 - [26] Y. Lu, L. Shang, X. Xie, and J. Xue. An incremental points-to analysis with CFL-reachability. In *CC*, pages 61–81, 2013.
 - [27] D. Lustig and M. Martonosi. Reducing GPU offload latency via fine-grained CPU-GPU synchronization. In *HPCA*, 2013.
 - [28] M. Méndez-Lojo, M. Burtscher, and K. Pingali. A GPU implementation of inclusion-based points-to analysis. In *PPoPP*, pages 107–116, 2012.

-
- [29] M. Méndez-Lojo, A. Mathew, and K. Pingali. Parallel inclusion-based points-to analysis. In *OOPSLA*, pages 428–443, 2010.
 - [30] L.-M. Munguia, D. A. Bader, and E. Ayguade. Task-based parallel breadth-first search in heterogeneous environments. In *HiPC*, pages 1–10, 2012.
 - [31] V. Nagaraj and R. Govindarajan. Parallel flow-sensitive pointer analysis by graph-rewriting. In *PACT*, pages 19–28, 2013.
 - [32] R. Nasre. Time-and space-efficient flow-sensitive points-to analysis. *TACO*, 10(4):39:1–39:27, 2013.
 - [33] R. Nasre, M. Burtscher, and K. Pingali. Morph algorithms on gpus. In *PPoPP*, pages 147–156, 2013.
 - [34] R. Nasre and R. Govindarajan. Prioritizing constraint evaluation for efficient points-to analysis. In *CGO*, pages 267–276, 2011.
 - [35] R. Niewiadowski, J. N. Amaral, and R. C. Holte. A parallel external-memory frontier breadth-first traversal algorithm for clusters of workstations. In *ICPP*, pages 531–538, 2006.
 - [36] D. J. Pearce, P. H. Kelly, and C. Hankin. Online cycle detection and difference propagation: Applications to pointer analysis. *Software Quality Journal*, 12(4):311–337, 2004.
 - [37] D. J. Pearce, P. H. J. Kelly, and C. Hankin. Efficient field-sensitive pointer analysis of C. *ACM Trans. Program. Lang. Syst.*, 30(1), 2007.
 - [38] F. M. Q. Pereira and D. Berlin. Wave propagation and deep propagation for pointer analysis. In *CGO*, pages 126–135, 2009.

-
- [39] T. Prabhu, S. Ramalingam, M. Might, and M. W. Hall. EigenCFA: Accelerating flow analysis with GPUs. In *POPL*, pages 511–522, 2011.
 - [40] S. Putta and R. Nasre. Parallel replication-based points-to analysis. In *CC*, pages 61–80, 2012.
 - [41] T. Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11):701–726, 1998.
 - [42] L. Shang, Y. Lu, and J. Xue. Fast and precise points-to analysis with incremental CFL-reachability summarisation: preliminary experience. In *ASE*, pages 270–273, 2012.
 - [43] L. Shang, X. Xie, and J. Xue. On-demand dynamic summary-based points-to analysis. In *CGO*, pages 264–274, 2012.
 - [44] Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well: Understanding object-sensitivity. In *POPL*, pages 17–30, 2011.
 - [45] M. Sridharan and R. Bodík. Refinement-based context-sensitive points-to analysis for Java. In *PLDI*, pages 387–400, 2006.
 - [46] M. Sridharan and S. Fink. The complexity of Andersen’s analysis in practice. In *SAS*, volume 5673, pages 205–221, 2009.
 - [47] M. Sridharan, S. J. Fink, and R. Bodík. Thin slicing. In *PLDI*, pages 112–122, 2007.
 - [48] M. Sridharan, D. Gopan, L. Shan, and R. Bodík. Demand-driven points-to analysis for Java. In *OOPSLA*, pages 59–76, 2005.
 - [49] Y. Su, D. Ye, and J. Xue. Accelerating inclusion-based pointer analysis on heterogeneous CPU-GPU systems. In *HiPC*, pages 149–158, 2013.

-
- [50] Y. Su, D. Ye, and J. Xue. Parallel pointer analysis with CFL-reachability. In *ICPP*, pages 451–460, 2014.
- [51] Y. Su, D. Ye, J. Xue, and X. Liao. An efficient GPU implementation of inclusion-based pointer analysis. *IEEE Trans. Parallel Distrib. Syst.*, 2015. To Appear.
- [52] Y. Sui, Y. Li, and J. Xue. Query-directed adaptive heap cloning for optimizing compilers. In *CGO*, pages 1–11, 2013.
- [53] Y. Sui, D. Ye, and J. Xue. Static memory leak detection using full-sparse value-flow analysis. In *ISSTA*, pages 254–264, 2012.
- [54] Y. Sui, D. Ye, and J. Xue. Detecting memory leaks statically with full-sparse value-flow analysis. *IEEE Trans. Software Eng.*, 40(2):107–122, 2014.
- [55] Y. Sui, S. Ye, J. Xue, and P.-C. Yew. SPAS: scalable path-sensitive pointer analysis on full-sparse SSA. In *APLAS*, pages 155–171. 2011.
- [56] Y. Sui, S. Ye, J. Xue, and J. Zhang. Making context-sensitive inclusion-based pointer analysis practical for compilers using parameterised summarisation. *Softw. Pract. Exper.*, 44(12):1485–1510, 2014.
- [57] Q. Sun, J. Zhao, and Y. Chen. Probabilistic points-to analysis for Java. In *CC*, pages 62–81, 2011.
- [58] R. Vallée-Rai, P. Co, E. Gagnon, L. J. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. In *CASCON*, page 13, 1999.
- [59] J. Whaley and M. S. Lam. An efficient inclusion-based points-to analysis for strictly-typed languages. In *SAS*, pages 180–195, 2002.

-
- [60] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI*, pages 131–144, 2004.
- [61] B. Wu, Z. Zhao, E. Z. Zhang, Y. Jiang, and X. Shen. Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on GPU. In *PPoPP*, pages 57–68, 2013.
- [62] X. Xiao and C. Zhang. Geometric encoding: forging the high performance context sensitive points-to analysis for Java. In *ISSTA*, pages 188–198, 2011.
- [63] G. Xu and A. Rountev. Merging equivalent contexts for scalable heap-cloning-based context-sensitive points-to analysis. In *ISSTA*, pages 225–236, 2008.
- [64] G. Xu, A. Rountev, and M. Sridharan. Scaling CFL-reachability-based points-to analysis using context-sensitive must-not-alias analysis. In *ECOOP*, pages 98–122, 2009.
- [65] D. Yan, G. Xu, and A. Rountev. Demand-driven context-sensitive alias analysis for Java. In *ISSTA*, pages 155–165, 2011.
- [66] D. Ye, Y. Sui, and J. Xue. Accelerating dynamic detection of uses of undefined values with static value-flow analysis. In *CGO*, pages 154–164, 2014.
- [67] S. Ye, Y. Sui, and J. Xue. Region-based selective flow-sensitive pointer analysis. In *SAS*, pages 319–336, 2014.
- [68] H. Yu, J. Xue, W. Huo, X. Feng, and Z. Zhang. Level by level: making flow- and context-sensitive pointer analysis scalable for millions of lines of code. In *CGO*, pages 218–229, 2010.

- [69] C. Zhang, H. Xu, S. Zhang, J. Zhao, and Y. Chen. Frequency estimation of virtual call targets for object-oriented programs. In *ECOOP*, pages 510–532, 2011.
- [70] E. Z. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. Shen. On-the-fly elimination of dynamic irregularities for gpu computing. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 369–380. ACM, 2011.
- [71] X. Zheng and R. Rugina. Demand-driven alias analysis for C. In *POPL*, pages 197–208, 2008.