

BLACK-BOX EQUIVALENCE CHECKING ACROSS COMPILER TRANSFORMATIONS

MANJEET DAHIYA



**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY DELHI
OCTOBER 2018**

BLACK-BOX EQUIVALENCE CHECKING ACROSS COMPILER TRANSFORMATIONS

by

MANJEET DAHIYA

Department of Computer Science and Engineering

Submitted

in fulfillment of the requirements of the degree of Doctor of Philosophy

to the



Indian Institute of Technology Delhi

October 2018

Certificate

This is to certify that the thesis titled **Black-box Equivalence Checking across Compiler Transformations** being submitted by **Manjeet Dahiya** for the award of **Doctor of Philosophy** in Computer Science and Engineering is a record of bona fide work carried out by him under my guidance and supervision at the Department of Computer Science and Engineering, Indian Institute of Technology Delhi. The work presented in this thesis has not been submitted elsewhere, either in part or full, for the award of any other degree or diploma.

Sorav Bansal

Associate Professor

Department of Computer Science
and Engineering

Indian Institute of Technology Delhi

New Delhi, India 110 016

Acknowledgments

At the outset, I would like to wholeheartedly thank my advisor Sorav Bansal for guiding me on how to choose the problems to work on, how to write papers, and how to present them. Most importantly, he instilled in me the approach of doing principled research, which I would carry for the rest of my career. I am grateful to my PhD committee members Sanjiva Prasad, Akash Lal and Sandeep Sen for their guidance and evaluation of my work on a regular basis. I am thankful to S. Arun Kumar and Sanjiva Prasad for floating some of the best courses during the period of my PhD, which also helped me in my research.

I would like to thank Deepak and Piyus with whom I had valuable discussions on various topics. I would also like to extend my thanks to the staff of CSE and SIT departments. I also sincerely thank TCS Research for granting me the scholarship during my PhD.

On a personal note, during my PhD, my wife and I were blessed with our cute little children Vaanya and Uday — I am thankful to God for his grace. I am grateful to my parents and family for their unending love and support. I also would like to remember my friends who have encouraged me in my pursuit, and humoured me when needed. Finally, I am grateful to my wife Ruchi because of whom I was able to take the huge step of quitting my job to pursue PhD, and these words could not have been ever written, otherwise. This thesis is dedicated to my loving wife Ruchi.

Manjeet Dahiya

Abstract

Equivalence checking is an important building block for program synthesis and verification. Design of an equivalence checker is dependent on the application; program synthesis tools like superoptimizers demand that the underlying equivalence checker should perform the required equivalence checks in a *black-box* manner, i.e., without requiring the knowledge of the individual constituent transformation passes. This thesis presents techniques for black-box equivalence checking across compiler optimizations and across power environments.

In the first part, we present a technique for black-box equivalence checking across compiler optimizations. Unlike previous work on translation validation, our technique works across multiple composed transformations and does not employ a pass-by-pass approach. Our technique supports undefined behaviour related optimizations, and we are the first to handle undefined behaviour related optimizations in equivalence checking for programs with loops. We test our checker with the optimizations produced by multiple modern compilers, and our results are comparable to that of previous work on translation validation, albeit in a black-box setting.

In the second part, we discuss equivalence checking across power environments. Intermittent programs, which are transiently powered, keep checkpointing the program state to a persistent memory, and on power failures, the programs resume from the last executed checkpoint. An intermittent program is usually automatically generated by instrumenting a given continuous program (continuously powered). The behaviour of the continuous program should be equivalent to that of the intermittent program under all possible power failures. We present a technique to automatically verify the correctness of an intermittent program with respect to its continuous counterpart. We present a model of intermittence to capture all possible scenarios of power failures and an algorithm to automatically find a proof of equivalence between a continuous and an intermittent program.

सारांश

ब्लैक-बॉक्स इक्विवैलेन्स चेकिंग अक्रॉस कम्पाइलर ट्रांसफॉर्मेशन्स

प्रोग्राम सिंथेसिस और वेरिफिकेशन के लिए, समानता की जाँच (इक्विवैलेन्स चेकिंग) एक महत्वपूर्ण निर्माण खंड है। एक समानता परीक्षक (इक्विवैलेन्स चेकेर) का डिजाइन उसके प्रयोग पर निर्भर है; प्रोग्राम सिंथेसिस उपकरण जैसे सुपरऑप्टिमाइज़र कि मांग है कि अंतर्निहित इक्विवैलेन्स चेकेर ब्लैक-बॉक्स जांच करे। ब्लैक-बॉक्स इक्विवैलेन्स चेकेर में आवश्यक है कि इसके जांच के तरीके को ट्रांसफॉर्मेशन पास्सेस के ज्ञान कि जरूरत ना पड़े। यह थीसिस, कम्पाइलर ऑप्टिमाइज़ेशंस और पावर एनवाइरमेंट्स के ब्लैक-बॉक्स समानता जांच के लिये तकनीकें प्रस्तुत करती है

पहले भाग में, हम कम्पाइलर ऑप्टिमाइज़ेशंस कि ब्लैक-बॉक्स इक्विवैलेन्स चेकिंग कि तकनीक पेश करते हैं। ट्रांसलेशन वेलिडेशन पर पिछले काम के विपरीत, हमारी तकनीक कई रचनाकृत परिवर्तनों (मल्टीपल कंपोज्ड ट्रांसफॉर्मेशन्स) पर काम करती है और पास-दर-पास दृष्टिकोण को नियोजित नहीं करती है। हमारी तकनीक अपरिभाषित व्यवहार (अन्डेफाइंड बिहेवियर) से संबंधित ऑप्टिमाइज़ेशंस कि जाँच करती है। इस थीसिस ने अनिर्धारित व्यवहार से संबंधित ऑप्टिमाइज़ेशंस व लूप के साथ प्रोग्राम्स के लिए समानता जांच में पहल करी है। हम कई आधुनिक कंपाइलर्स द्वारा उत्पादित ऑप्टिमाइज़ेशंस के साथ हमारे इक्विवैलेन्स चेकेर का परीक्षण करते हैं, और हमारे परिणाम पिछले काम के तुलनीय हैं, हालांकि एक ब्लैक-बॉक्स सेटिंग में।

दूसरे भाग में, हम पावर एनवाइरमेंट्स कि समानता जांच पर चर्चा करते हैं। इंटरमिटेंट प्रोग्राम्स, जोकी प्रोग्राम स्थिति को एक परसिस्टेंट मेमोरी में चेंकपॉइंट करता है ताकि बिजली विफलताओं (पावर फेल्योर्स) में प्रोग्राम को फिर से शुरू कर सके। इंटरमिटेंट प्रोग्राम आमतौर पर स्वचालित रूप से कंटीन्यूअस प्रोग्राम (कन्टिन्यूसली पॉवर्ड) कि इंस्ट्रुमेंटेशन से उत्पन्न होता है। सभी संभावित बिजली विफलताओं के तहत, कंटीन्यूअस प्रोग्राम का व्यवहार इंटरमिटेंट प्रोग्राम के व्यवहार के समान होना चाहिए। हम स्वचालित रूप से इसके करेक्टनेस को सत्यापित करने कि तकनीक प्रस्तुत करते हैं। हम इंटरमिटेन्स का एक मॉडल और इंटरमिटेंट प्रोग्राम्स व कंटीन्यूअस प्रोग्राम्स के बीच में समानता की जांच का अल्गोरिथम प्रस्तुत करते हैं।

Contents

List of Figures	17
List of Tables	21
List of Algorithms	23
1 Introduction	25
1.1 Contributions and organization	28
2 Equivalence checking across compiler optimizations	31
2.1 Simulation relation as the basis of equivalence	33
2.2 Black-box equivalence checking algorithm	36
2.2.1 Introducing the algorithm through an example	37
2.2.2 Transfer function graph	41
2.2.3 Joint transfer function graph	43
2.2.4 Algorithm for determining the simulation relation	45
2.2.5 Evaluation: guarantees and supported optimizations	50
2.3 Modeling undefined behaviour semantics	53
2.3.1 Motivating example	55
2.3.2 Extended simulation relation (with assumptions)	58
2.3.3 Modeling undefined behaviour assumptions	61
2.3.4 Evaluation	67

2.4	Implementation: optimizations and heuristics	69
2.4.1	Fixing nodes and edges in the optimized TFG	69
2.4.2	Heuristics for prioritizing guesses	71
2.4.3	SMT solver optimizations	72
2.5	Combined evaluation	74
2.5.1	Equivalence checking across compiler optimizations	76
2.5.2	Bugs discovery	83
2.5.3	Superoptimization experiments	88
2.6	Related work	90
3	Equivalence checking across power environments	99
3.1	Example	101
3.2	Program representation	105
3.3	Modeling intermittence	106
3.3.1	Instrumentation model	106
3.3.2	Modeling power failures	108
3.3.3	Resolving indirect branches of restoration logic	109
3.4	Equivalence	110
3.4.1	Correlation	112
3.4.2	Inferring invariants	115
3.4.3	Intermediate observables	116
3.5	Evaluation	119
3.6	Related work	122
4	Conclusion and future directions	127
	Bibliography	131

List of Figures

2.1	Abstracted versions of unoptimized and optimized implementations of programs to compute double sum of first $n - 1$ natural numbers. The first program computes $\sum_{i=1}^{n-1} 2 * i$ whereas, the second program computes $2 * \sum_{j=1}^{n-1} j$	35
2.2	Simulation relation for the programs of Figure 2.1. Table shows the invariants at each correlated location of the two programs. Locations (b0, b0') and (b3, b3') are the entry and the exit rows respectively. <i>Init</i> is the initial condition representing the equivalence of inputs.	35
2.3	An example function computing sum of positive integers of global array <i>g</i>	38
2.4	Abstracted versions of unoptimized and optimized implementations of the program in Figure 2.3. The ternary operator <i>a?b:c</i> represents the <i>cmov</i> assembly instruction.	39
2.5	Simulation relation (JTFG) for the programs of Figure 2.4. Table shows the invariants at each node and graph shows the correlation of edges and nodes of the two programs. <i>Init</i> is the initial conditions representing equivalence of inputs. Operator sl_4 is a shorthand of SMT operator <i>select</i> of size 4. The SMT expression $sl_4(M_A, g_A + i_A)$ is an equivalent representation of $g_A[i_A]$. The edges of the first program between b1 and b4 and b1 and b1 are composite edges made up of the individual edges in between.	41
2.6	Grammar of transfer function graph ($\mathbb{T}$).	42

2.7	TFGs of the unoptimized (top) and optimized programs of Figure 2.4, represented as a table of edges. The ‘condition’ column represents the edge condition, and τ represents the transfer function. Operator sl_4 is a shorthand of <i>select</i> of size 4. Therefore, $sl_4(M, g + i)$ represents $g[i]$. . .	44
2.8	An example function. <code>sum2</code> is allocated by the caller.	56
2.9	Unoptimized and optimized, abstracted versions of the program in Figure 2.8.	56
2.10	Extended simulation relation for the programs in Figure 2.9. Table shows the invariants at each location pair. (b0, b0’) and (b3, b3’) are the entry and exit rows respectively. A_A and $\&sum1_A$ are the base addresses of the globals <code>A</code> and <code>sum1</code> respectively in $Prog_A$. $sl_4(M, addr)$ represents 4 bytes of data read in memory (M) at address $addr$. $=_{\Delta}$ represents equivalent memory states except at Δ ; Δ represents the stack region. <code>Init</code> represents equivalence of inputs.	58
2.11	For every benchmark-compiler option, the first bar shows the success rates when we model all three UB. The remaining three bars show the success rates when a particular type of UB among three (TBSA, SIO, OBVA) is not modeled. Each bar individually shows the contribution to the success rates by cyclic (at least one loop) and acyclic functions.	68
2.12	Performance of benchmarks across different compilers.	75
2.13	Equivalence statistics. Functions with at least one loop are called “cyclic”. The bar corresponding to a compiler (e.g., <code>clang</code>) represents the results across O0/O2 and O0/O3 transformations for that compiler (e.g., for <code>clang2</code> and <code>clang3</code> resp.). The average success rate across 26007 equivalence tests on these benchmarks, is 76% for O2 and 72% for O3 (dashed blue and red lines resp.). The missing bars for <code>ccomp</code> are due to compilation failures for those benchmarks.	77
2.14	Cumulative success rate (pass/fail) vs. ALOC.	78

2.15	Success rate vs. composite edges in TFG_B	78
2.16	Time taken for equivalence test (Y-axis) plotted against number of (a) JTFGs checked, (b) ALOC and (c) number of composite edges in TFG_B . For acyclic programs, the number of JTFGs checked and the number of composite edges in TFG_B will be 2^1 and 2^0 respectively (independent of program ALOC). Both X and Y axes are in log-scale. For some functions (while debugging), we explicitly used a timeout value of > 5 hours, and so a few points appear above the 5 hour limit.	80
3.1	The first assembly program increments a global non-volatile variable <code>nv</code> and returns 0. It also shows the checkpoint locations CP1 and CP2 and respective checkpoint elements (CPelems1 and CPelems2) that need to checkpointed at these locations. The second program is an intermittent program, which is generated by instrumenting the first program at the given checkpoint locations.	103
3.2	Modified grammar of transfer function graph to support volatility and checkpointing of state elements. Bold attributes depict the modifications over the grammar of Figure 2.6.	105
3.3	TFGs of the continuous and the intermittent program of Figure 3.1. . . .	106
3.4	The first figure shows a simplified intermittent TFG, the edges and the nodes have been duplicated for exposition and non-reachable failure paths have been removed. Checkpoint-to-checkpoint paths formed by dashed edges are failure paths and that formed by solid edges are progress paths. The second figure shows the correlation graph; single-edges show correlations of no-moves with failure paths. The third figure shows the invariants at the checkpoint nodes and exit.	114

List of Tables

2.1	Examples of types of C undefined behaviour that can be modeled through syntactic analysis of the program.	62
2.2	Forward dataflow rules to compute <i>lr</i> and <i>dep</i> relations. <i>arg</i> $\in$ <i>function arguments</i> , <i>global</i> $\in$ <i>global variables</i> , <i>heap</i> $\in$ <i>heap variables</i> , and <i>stack</i> $\in$ <i>local variables</i> . $\oplus$ represents the addition or subtraction operators. <i>OP</i> is a function that uses <i>p</i> as an argument.	66
2.3	Benchmarks characteristics. Fun, UN and Loop columns represent the total number of functions, the number of functions containing unsupported opcodes, and the number of functions with at least one loop, resp. SLOC is determined through the sloccount tool. ALOC is based on gcc-O0 compilation. Globals represent the number of global variables in the executable.	76
2.4	Success rates of equivalence checking across <code>compcert-O0</code> and <code>gcc-O2</code>	81

2.5 Examples of programs synthesized with x86 string instructions through a 32-bit superoptimizer. The columns indicate the runtimes for different compiler/optimization pairs. The measurements are speedups relative to unoptimized (O0) code produced by gcc (**higher is better**). The best performing configuration is highlighted in bold. The letter X is used to represent byte (b), word (w), or long (l) variants of the instructions/operations. The speedup column represents the ratio of the performance of the superoptimized sequence over the best configuration among the compilers. At input, the number of iterations (n) is in `ecx`, the array pointers are in `esi` and `edi`, and the value being stored/compared in `eax`; the operand written in the last instruction is the return value. 89

3.1 For each benchmark, the second column gives the number of checkpoint nodes, the third and the fourth column give the average checkpoint size (bytes) determined by DINO and synthesis loop respectively, the fifth column gives improvement by synthesis loop over DINO, and the sixth and the last column give the total time taken by the synthesis loop and the average runtime of the verifier respectively. 121

List of Algorithms

1	Determining correlation. μ is the unroll factor.	46
---	--	----