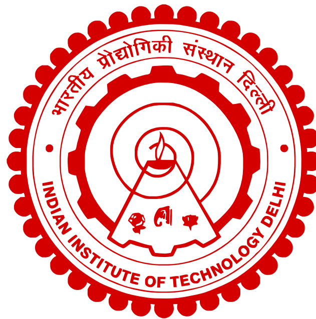
MODELING DYNAMIC ALLOCATIONS AND DEALLOCATIONS OF LOCAL MEMORY FOR TRANSLATION VALIDATION

ABHISHEK ROSE



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY DELHI

FEBRUARY 2025

©Indian Institute of Technology Delhi (IITD), New Delhi - 2025

MODELING DYNAMIC ALLOCATIONS AND DEALLOCATIONS OF LOCAL MEMORY FOR TRANSLATION VALIDATION

by

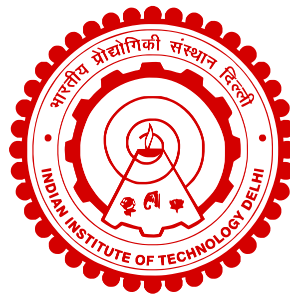
Abhishek Rose

Department of Computer Science and Engineering

Submitted

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

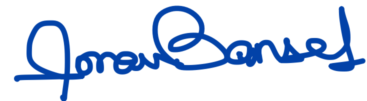
to the



Indian Institute of Technology Delhi
FEBRUARY 2025

Certificate

This is to certify that the thesis titled **Modeling Dynamic Allocations and Deallocations of Local Memory for Translation Validation** being submitted by **Mr. Abhishek Rose** 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.



Prof. Sorav Bansal
Microsoft Chair Professor
Department of Computer Science and Engineering
Indian Institute of Technology Delhi
New Delhi 110016

Acknowledgements

I would like to thank all who contributed to this dissertation.

My sincere gratitude goes to my supervisor, Prof. Prof. Sorav Bansal, for expert guidance, insightful feedback, and unwavering support.

I also appreciate the valuable input of my committee members, Prof. Sanjiva Prasad, Prof. Rahul Purandare, and Prof. Subodh Sharma, whose critiques greatly improved this work.

I am grateful to my fellow students, Shubhani, Indrajit, Vaibhav, and Pratik, for their collaboration and stimulating discussions.

I also acknowledge the staff of the Department of Computer Science and Engineering for their administrative and technical support.

Finally, I thank the reviewers for their thorough evaluations and constructive comments, which helped refine the final dissertation.

Abhishek Rose

Abstract

End-to-End Translation Validation is the problem of verifying the executable code generated by a compiler against the corresponding input source code for a single compilation. This becomes particularly hard in the presence of dynamically-allocated local memory where addresses of local memory may be observed by the program. In the context of validating the translation of a C procedure to executable code, a validator needs to tackle constant-length local arrays, address-taken local variables, address-taken formal parameters, variable-length local arrays, procedure-call arguments (including variadic arguments), and the `alloca()` operator.

We make the following contributions in our work:

1. A formalization of the execution semantics for an unoptimized intermediate representation (IR) of a C program and its compiled 32-bit x86 assembly in the presence of dynamically (de)allocated local memory. This includes modeling of the various dynamic allocation constructs in C, such as address-taken local variables, constant- and variable-length local arrays, address-taken formal parameters, procedure-call arguments (including variadic arguments), and the `alloca()` operator.

2. A notion of correct translation from the IR to the assembly through a refinement definition. The definition incorporates the concept of undefined behavior (UB) within the IR program, originally translated from C, where refinement is permitted to hold

trivially.

3. An algorithm that converts the correct translation check to first-order logic queries over bitvectors, arrays, and uninterpreted functions that can be discharged using off-the-shelf SMT solvers. The algorithm is capable of operating in both blackbox and whitebox modes, with the blackbox mode enabling its usage with third-party compilers that may not employ a specific allocation strategy, such as preallocation. In particular, we are perhaps the first to enable support for dynamic stack allocation strategy for procedure-call arguments used by almost all production compilers (e.g., GCC, Clang/LLVM).

4. A prototype implementation of the algorithm and its comprehensive evaluation on a set of diverse benchmarks, including both micro-benchmarks and a real-world `bzip2` program. Our prototype performs blackbox translation validation of C procedures with up to 100+ SLOC against their corresponding assembly implementations with up to 140+ instructions generated by an optimizing production compilers (such as GCC, Clang/LLVM, ICC) with complex loop and vectorizing transformations.

सार

एंड–टू–एंड ट्रांसलेशन वैलिडेशन का उद्देश्य किसी कम्पाइलर द्वारा जनरेट किए गए एक्ज़ीक्यूटेबल कोड को उसके इनपुट सोर्स कोड के विरुद्ध सत्यापित करना है। यह कार्य विशेष रूप से कठिन हो जाता है जब डायनामिकली–अलोकेटेड लोकल मेमोरी मौजूद होती है, जहाँ लोकल मेमोरी के ऐड्रेसेस को प्रोग्राम द्वारा ऑब्ज़र्व किया जा सकता है।

सी प्रोसीजर के एक्ज़ीक्यूटेबल कोड में ट्रांसलेशन की वैलिडेशन के संदर्भ में, एक वैलिडेटर को निम्नलिखित समस्याओं से निपटना पड़ता है: कन्स्टेंट–लेंथ लोकल एरेज़, ऐड्रेस–टेकन लोकल वेरिएबल्स, ऐड्रेस–टेकन फॉर्मल पैरामीटर्स, वैरिएबल–लेंथ लोकल एरेज़, प्रोसीजर–कॉल आर्ग्युमेंट्स (जिसमें वैरिऐडिक आर्ग्युमेंट्स भी शामिल हैं), और alloca() ऑपरेटर।

हम अपने कार्य में निम्नलिखित योगदान प्रस्तुत करते हैं:

1. सी प्रोग्राम के अन–ऑप्टिमाइज़्ड इंटरमीडिएट रिप्रेजेंटेशन (आई–आर) और 32–बिट एक्स–86 असेंबली के एक्सीक्यूशन सीमांटिक्स का फॉर्मलाइजेशन, जो डायनामिक (डी)अलोकेटेड लोकल मेमोरी की उपस्थिति में वैध रहता है। इसमें सी की विभिन्न डायनामिक अलोकेशन कंस्ट्रक्ट्स जैसे ऐड्रेस–टेकन लोकल वेरिएबल्स, कन्स्टेंट– और वैरिएबल–लेंथ लोकल एरेज़, ऐड्रेस–टेकन फॉर्मल पैरामीटर्स, प्रोसीजर–कॉल आर्ग्युमेंट्स (जिसमें वैरिऐडिक आर्ग्युमेंट्स भी शामिल हैं) और alloca() ऑपरेटर को मॉडल करना शामिल है।

2. आई–आर से असेंबली तक सही ट्रांसलेशन की एक परिभाषा, जो रिफाइनमेंट की अवधारणा पर आधारित है। यह परिभाषा आई–आर प्रोग्राम (जो मूल रूप से सी से ट्रांसलेट किया गया है) में अनडिफाइंड बिहेवियर (यू–बी) की उपस्थिति को सम्मिलित करती है, जहाँ रिफाइनमेंट स्वतःसिद्ध रूप से वैध होता है।

3. एक एल्गोरिदम जो सही ट्रांसलेशन वैलिडेशन को फर्स्ट–ऑर्डर लॉजिक क्वेरीज में कन्वर्ट करता है, जो बिटवेक्टर्स, एरेज़, और अनइंटरप्रिटेड फंक्शन्स के उपयोग से व्यक्त किए जाते हैं और एस–एम–टी सॉल्वर्स द्वारा हल किए जा सकते हैं। यह एल्गोरिदम ब्लैकबॉक्स और व्हाइटबॉक्स दोनों मोड में कार्य कर सकता है। ब्लैकबॉक्स मोड का प्रयोग थर्ड–पार्टी कम्पाइलर्स के साथ किया जा सकता है, जो एक निश्चित अलोकेशन स्ट्रेटजी (जैसे

प्रीअलोकेशन) का पालन नहीं करते हैं। विशेष रूप से, हम पहले शोधकर्ताओं में से हैं जिन्होंने डायनामिक स्टैक अलोकेशन स्ट्रेटजी को प्रोसीजर–कॉल आर्ग्युमेंट्स के लिए सक्षम किया है, जिसका उपयोग लगभग सभी प्रोडक्शन कम्पाइलर्स (जैसे जी–सी–सी, क्लैंग/एल–एल–वी–एम) द्वारा किया जाता है।

4. एल्गोरिदम के एक प्रोटोटाइप इम्प्लीमेंटेशन और इसकी व्यापक मूल्यांकन। हमने इसे विभिन्न बेंचमार्क्स (जिसमें माइक्रो–बेंचमार्क्स और एक व्यावहारिक बी–जिप–2 प्रोग्राम शामिल है) पर परीक्षण किया। हमारा प्रोटोटाइप, ब्लैकबॉक्स ट्रांसलेशन वैलिडेशन को सी प्रोसीजर्स पर लागू करता है जिनकी एस–एल–ओ–सी 100+ तक हो सकती है और उनके संबंधित असेंबली इम्प्लीमेंटेशन जिनमें 140+ इंस्ट्रक्शन्स तक हो सकती हैं, जो जी–सी–सी, क्लैंग/एल–एल–वी–एम, आई–सी–सी जैसे ऑप्टिमाइज़िंग प्रोडक्शन कम्पाइलर्स द्वारा कॉम्प्लेक्स लूप और वेक्टोराइज़िंग ट्रांसफॉर्मेशन्स के साथ जनरेट की गई हैं।

Contents

Acknowledgements **iii**

Abstract **v**

List of Figures **xx**

1 Introduction **1**

1.1 Problem Statement and Motivating Example 4

1.1.1 An address-taken local example 5

1.1.2 Establishing Correct Translation 7

1.1.3 Subtleties 9

1.1.4 A sketch of proposed solution 12

1.2 Prior Work 13

1.2.1 IR-to-IR Translation Validation 13

1.2.2 IR-to-Assembly and Assembly-to-Assembly Translation Validation 14

1.2.3 Verified Compilation 15

1.3 Contributions 15

1.4 Outline 17

2 Execution Semantics and Notion of Correct Translation 19

2.1 Intermediate Source and Assembly Representations 19

2.1.1 Unoptimized IR 21

2.1.2 Assembly 25

2.1.3 Allocation and Deallocation 25

2.2 Transition Graph Representation 26

2.2.1 Address Set 27

2.2.2 Memory Regions 27

2.2.3 Ghost Variables 29

2.2.4 Error Codes 30

2.2.5 Outside world and observable trace 30

2.2.6 Expressions 31

2.2.7 Graph Instructions 31

2.3 Translations of C and A to their Graph Representations 33

2.3.1 Translation of C 34

2.3.2 Translation of A 40

2.4 Observable traces and Refinement Definition 44

2.5 Refinement Definition in the presence of local variables and procedure calls when all local variables are allocated on the stack in A 47

2.5.1 (De)Allocation indicating $\texttt{alloc}_s$ and $\texttt{dealloc}_s$ instructions . . 48

2.5.2 Annotated procedure-call instruction 50

2.5.3 Refinement Definition with only stack-allocated locals and procedure calls 51

2.5.4 Capabilities and Limitations of C $\dot{\sqsupseteq}$ A 53

2.6 Refinement in the presence of potentially register-allocated or eliminated local variables in A 57

2.6.1 Virtual (de)allocations through $\texttt{alloc}_v$ and $\texttt{dealloc}_v$ instructions 57

2.6.2 Revised semantics for assembly procedure instructions 60

2.6.3 Refinement Definition with both stack-allocated and register-allocated or eliminated locals 62

2.7 Towards A More General Refinement Definition and Execution Semantics 66

2.7.1 Comparison with C $\ddot{\sqsupseteq}$ A 69

3 Witnessing Refinement through a Determinized Cross-Product 75

3.1 Program Paths 76

3.2 Determinized Product Graph as a Transition Graph 77

3.3 Analysis of the determinized product graph 78

3.3.1 X requirements 79
3.3.2 Soundness of X requirements 82
3.3.3 Global Invariants in C, Ä, and X 88
3.4 Callers' Virtual Smallest Semantics 90
3.4.1 Soundness of Callers' Virtual Smallest semantics 91
3.5 Safety-Relaxed Semantics 93
3.5.1 Soundness of Safety-Relaxed Semantics 94

4 Automatic Construction of a Product-Program 101

4.1 The DYNAMO algorithm 101
4.1.1 Enumerating A paths 105
4.1.2 Correlating C paths 107
4.1.3 Identifying A annotation 114
4.1.4 Validating structure of identified paths 118
4.1.5 Incremental construction of (Ä, X) 120
4.1.6 Checking requirements on partial X 122
4.1.7 Correlating paths to error nodes due to annotated instructions . 125
4.1.8 Soundness of DYNAMO algorithm 125
4.1.9 Counterexample Guided Best-First Search 125
4.2 Invariant Inference 126

4.2.1 Global Invariants 128

4.3 Running Example of the Algorithm 130

5 SMT Encoding 141

5.1 Preliminary Steps 141

5.2 Representing address sets using allocation state array 143

5.2.1 Encoding of address set updating instructions 144

5.2.2 Full-array encoding 145

5.3 Interval Encoding 146

5.3.1 Interval encoding for $r \in G \cup F \cup Y \cup Z_l \cup \{stk\}$ 146

5.3.2 Interval encoding for $r \in \{hp, cl, cs\}$ 146

5.3.3 Soundness of Interval Encoding 148

5.4 Semantics with Simpler SMT Encoding for stk Region of Ä 157

6 Evaluation 161

6.1 Implementation of DYNAMO 161

6.1.1 System components 161

6.1.2 Discharging Proof Obligations 163

6.1.3 Pseudo-register allocation in $\texttt{LLVM}_d$ 164

6.1.4 Instrumentation of Clang/LLVM for generating annotation hints 164

6.2 Experiments . . . 164

6.2.1 Evaluating efficacy of Dynamo . . . 165

6.2.2 Evaluating modeling cost of local allocations . . . 168

6.2.3 Evaluating Dynamo on a real-world program . . . 170

6.2.4 Analysis of Failures . . . 172

6.3 Other Applications . . . 178

7 Conclusion 181

7.1 Summary . . . 181

7.2 Limitations and Directions for Future Work . . . 182

Appendices 185

A Soundness and Completeness Implications of `isPush()` Choice 187

A.1 $\mathbb{K}$ needs to be at least 2^{d-1} in the presence of VLAs . . . 188

A.2 $\mathbb{K} = 2^{d-1}$ can still lead to completeness problems . . . 189

A.3 $\mathbb{K} = 2^{d-1}$ can also lead to soundness problems . . . 189

A.4 Solution . . . 190

B More details of the experiments 191

B.1 Command-line used for compiling benchmarks in experiments . . . 191

B.2 Full results for the `bzip2` experiment . . . 195

C Full source code of the benchmarks 199

Bibliography 212

List of Publications 213

Biography 215

List of Figures

1.1 Two ways of obtaining a certified executable from an input source code. The certified executable is guaranteed to have identical semantics as the input source code. 3

1.2 C program with an address-taken local and its IR and 32-bit x86 assembly lowerings. `@STR` denotes the address of the format string `"%d"`. $\texttt{mem}_4\texttt{[}a\texttt{]}$ denotes a 4-byte memory access to address a. 5

1.3 C program with variable-length array (VLA) and its assembly lowerings. $\texttt{mem}_4\texttt{[}a\texttt{]}$ denotes a 4-byte memory access to address a. An execution where $\texttt{m} \geq \texttt{n}$ triggers undefined behavior in the C program and C semantics do not put restriction on behavior of the translated assembly program in such an execution. 10

1.4 Conceptual representation of the flow of information from high-level source program and the low-level assembly program to the compiler/-translation validator. 11

1.5 High-level components and flow of our translation validation approach. 12

2.1 C program with variable-length array (VLA) and its lowerings to unoptimized IR and assembly. Subscript '$_s$' denotes signed comparison. Red font (parts of) instructions in assembly are added by our algorithm. . 20

2.2 Pseudo-code for translation of a C procedure-call expression to $\texttt{LLVM}_d$ instructions. `alloc`, `dealloc`, `call`, `va_start_ptr`, etc. are $\texttt{LLVM}_d$ instructions. 23

2.3 Translation of C's variadic macros to $\texttt{LLVM}_d$ instructions. $\texttt{roundup}_4(a)$ returns the closest multiple of 4 greater than or equal to a. 24

2.4 Translation rules for converting $\texttt{LLVM}_d$ instructions to graph instructions. `op` represents an arithmetic, logical, or relational operator. c represents a constant. 36

2.5 Translation rules for converting $\texttt{LLVM}_d$ instructions to graph instructions. 38

2.6 Translation rules for converting pseudo-assembly instructions to graph instructions. `op` represents an arithmetic, logical, or relational operator. 41

2.7 Translation rules for converting pseudo-assembly instructions to graph instructions. 43

2.8 Additional translation rules for converting pseudo-assembly instructions to graph instructions for procedures with only stack-allocated locals. . 49

2.9 Example of transformation where relative order of (de)allocations and procedure calls is not preserved. The refinement definition will not admit the hypothetical assembly but will admit the compiler generated one. . 54

2.10 Translation rules for converting the $\texttt{alloc}_v$ and $\texttt{dealloc}_v$ instructions instructions to graph instructions. 58

2.11 Revised translation rules for converting pseudo-assembly instructions to graph instructions. The IF{$z \in Z_l$}{...}ELSE{...} construct selects one of the translation depending on the result of syntactic predicate $z \in Z_l$. 60

2.12 Translation rules for the converting pseudo-assembly instructions to graph instructions for Ä. (AllocV') is derived from (AllocV) in fig. 2.10. 67

4.1 C source fragment and its abbreviated control-flow graph. 113

4.2 Predicate grammar for constructing candidate invariants. v represents a bitvector variable (registers, stack slots, and ghost variables), c represents a bitvector constant. $\odot \in \{\leq_{s,u}, <_{s,u}, >_{s,u}, \geq_{s,u}\}$. v^* represents a bitvector value drawn from a restricted grammar (explained in text). 127

4.3 Global invariants that hold at each non-entry, error-free node $n_X \in \mathcal{N}_X^{\mathcal{UW}}$. 128

4.4 Reproduced C program and its unoptimized IR and assembly from fig. 2.1. 131

4.5 Abbreviated Transition Graphs for the unoptimized IR and assembly of the `fib` procedure from fig. 4.4. 132

5.1 Revised translation rules for the new stk^+-based semantics for Ä. 158

6.1 High-level components of Dynamo implementation. Trusted Code Base (TCB) blocks have double border and a red background. 162

6.2 Comparison of running times of procedures in table 6.1 with full-interval (filled bars), partial-interval (thick black lines), and full-array (empty bars) encoding. Y-axis is logarithmically scaled. 167

6.3 Summary of refinement check results for the programs in table 6.1. . . 168

6.4 Procedure `s122` from '`globals`' version of (modified) TSVC suite. . . . 169

6.5 Comparison of running times of TSVC benchmarks with exactly same code modulo allocation strategy. Y-axis is logarithmically scaled. 170

6.6 Scatter plot of refinement time (in minutes) vs assembly lines of code (ALOC). Both axes are logarithmically scaled. 172

6.7 `vsl1` procedure from table 6.1 (appears as `vsl`N) and the control-flow graph (CFG) of its GCC compiled assembly. 174

C.1 Benchmarks with VLAs. 200

C.2 Benchmarks with use of `alloca` 201

C.3 Benchmark `rod` with mixed use of VLA and address-taken variable. . . 202

C.4 Benchmarks `mp` and `ms` with variable argument list. `mp` is adapted from `minprintf` of K&R[24] 203

C.5 Structure of `bzip2`'s functions 204