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.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Equivalence checking is a fundamental problem in Computer science. The problem is undecidable and is as old as computing itself. Besides its theoretical significance, equivalence checking has important applications in the areas of compiler verification and program synthesis.

In the setting of compiler verification, equivalence checking is employed to verify the correctness of the transformations performed by the compilers, also called *translation validation* [39, 52]. Compilers perform a fixed set of transformations, which are usually correct-by-construction. However, bugs in compiler implementation could generate incorrect transformation instances and translation validation can catch the same. While the application of equivalence checking for translation validation is optional for compilers, it is an essential and an important building block of a class of synthesis tools that are given programs as the specification, e.g., superoptimizer [8, 2]. A synthesis tool like superoptimizer *generates* many different transformations, and it needs some mechanism to know whether a proposed synthesized transformation is correct with respect to the original program, and it relies on an equivalence checker to know the same.

For an input program, a synthesis tool searches for the output program with respect to some given optimization criteria like a lesser runtime or a smaller code size, etc. The synthesis tool generates (e.g., by enumeration or other heuristic-based search algorithms)

many possible proposed programs, based on the required criteria, and discharges the correctness check to the equivalence checker. If a proposed program is equivalent to the input program, the synthesis tool considers it as a potential optimized output program, otherwise, it looks for other better solutions. By the design of synthesis techniques like superoptimization, the proposed programs are usually not equivalent to the input program like in the correct-by-construction approach of compilers. This necessitates the use of an equivalence checker within a synthesis tool. Thus, an equivalence checker is indispensable to a synthesis tool.

Further, the capability and the performance of a synthesis tool are dependent on that of the underlying equivalence checker. Because the problem of equivalence checking is undecidable in general, incorrect failures are inevitable, i.e., the equivalence checker cannot be both sound and complete. An incorrect equivalence failure (i.e., a false negative due to incompleteness) results in a potentially missed optimization by the synthesis tool, but an incorrect equivalence success (i.e., a false positive due to unsoundness) by the equivalence checker results in an incorrect translation by the synthesis tool. For the application to program synthesis, false positives by an equivalence checker are unacceptable, and we would like to minimize false negatives for better results. Evidently, the capability and the correctness of a synthesis tool are dependent on the capability and the correctness of the underlying equivalence checker. Moreover, the performance of a synthesis tool is also dependent on that of the equivalence checker. This signifies the importance of an equivalence checker in a synthesis tool.

Previous work on program equivalence checking has been done in the context of translation validation, with the goal to verify the correctness of a translation. This prior work has largely employed a pass-by-pass based approach [39, 24], where each pass is verified separately by the equivalence checker, and/or worked with a set of handpicked transformations [39, 24, 52]. A pass-by-pass approach simplifies the verification process by dividing a big step into smaller and simpler steps, and the result is obtained by composing the results of the individual steps. While this meets the objective of translation validation and

is an efficient approach for the same, however, these are unsuitable simplifications for a synthesis tool, where the nature and sequence of transformations are unknown.

In contrast with a fixed set of transformations in compilers, the synthesis tools are search based and the transformations are synthesized. As a result, the nature of a transformation generated by the synthesis tool is not available to the equivalence checker. That is, a generated transformation is a *black-box* to the equivalence checker — this is the basic reason why the synthesis techniques can discover unconventional optimizations. In other words, a synthesis tool does not have a fixed ordered set of transformation passes, and a transformation generated by a synthesis tool may consist of multiple composed passes and in no particular order. This results in a restriction on the underlying equivalence checker as the equivalence checker cannot make the simplifications like in the case of translation validation. The equivalence checker for synthesis has to work without the knowledge of the transformations performed, and it has to verify multiple composed transformations in one go. Clearly, the setting for equivalence checking for synthesis is more challenging than that for translation validation. Stated differently, an equivalence checker suitable for a synthesis tool can be used for translation validation, while, the converse is not true.

Synthesis tools have not matured like compilers; we do not have substantially built synthesis tools to use our equivalence checker with. We resort to using compilers as a proxy for synthesis tools to allow us to bootstrap our equivalence checker independent of a synthesis tool. However, we simulate the setting of synthesis, i.e., by computing equivalence across the black-box composed transformations, without using the knowledge of the nature and sequences of the transformations performed. If we can come up with an equivalence checker that can verify the black-box transformations produced by modern compilers, then a future synthesis tool will be capable of producing modern compiler transformations. This forms the first part of this thesis: we present the study, design, and implementation of an equivalence checker that works across compiler optimizations in a *black-box* setting.

Synthesis techniques are not limited to just the conventional code optimizations. A

synthesis approach can be used to optimize program instrumentations too. For example, a synthesis technique can be employed to automatically generate an optimized *intermittent program* based on a desired criteria, from an input *continuous program*. An intermittent program is one which can work with an intermittent power supply and tolerate power failures due to the transient nature of the power supply. On the other hand, a continuous program requires continuous power supply for correct operation. The instrumentation is added such that the intermittent program keeps checkpointing the intermediate program state at the checkpoint locations, and in case of a power failure, the intermittent program resumes from the last checkpointed state. This allows the intermittent program to work with transiently powered, energy harvesting devices that harvest energy from their surroundings, such as sunlight or RF radio signals. For such a transformation to work, the behaviour of the intermittent program under all possible power failures should be equivalent to that of the continuous program.

The performance of an intermittent program is dependent on the location and the size of checkpoints. A synthesis tool in such a setting would search for an optimal placement of checkpoints and the minimum program state that needs to be checkpointed. Such a synthesis tool would require an equivalence checker that can tell if the intermittent program with the given checkpoints, under all possible power failures, is equivalent to its continuous counterpart. As shown in this thesis, this problem is also undecidable and the capability of such a synthesis tool is dependent on the robustness of the underlying equivalence checker. We present a technique to establish equivalence between a continuous and an intermittent program, and this forms the second part of this thesis.

1.1 Contributions and organization

This thesis makes the following high level contributions:

- A sound and robust algorithm for black-box equivalence checking across modern compiler optimizations. It can compute equivalence across almost all the optimiza-

tions of modern compilers, in a black-box manner. Notably, it supports optimizations related to the undefined behaviour semantics of high-level languages. We are the first to handle undefined behaviour related optimizations in equivalence checking for programs containing loops. Further, we have evaluated our technique exhaustively across multiple optimization levels over multiple compilers.

Chapter 2 presents our black-box equivalence checking algorithm, our technique to handle the optimizations related to undefined behaviour, the evaluation of our equivalence checker across compiler optimizations, details on the bugs found, and a preliminary evaluation of our equivalence checker in the setting of superoptimization.

- A sound technique to check equivalence across power environments, i.e., between a continuous and an intermittent program, which are supplied with continuous and transient power respectively. The technique establishes that the continuous program is equivalent to the intermittent program under all possible power failures, for the given set of checkpoints of the intermittent program. Further, the technique is robust with respect to the permitted set of checkpoints, in the sense that it can establish equivalence even when minimal program state is checkpointed.

  Chapter 3 presents our technique, and the experiments in the setting of synthesis of checkpoints, to demonstrate its capability.

Chapter 2

Equivalence checking across compiler optimizations

Modern compilers are overwhelmingly complex, e.g., GCC has over 14.5 million lines of code [41] and hundreds of optimization passes. We find that the transformations produced by these compilers are much varied and compositions of these optimization passes result in strongly differing control flow graphs. Furthermore, the presence of language level *undefined behaviour* allows them to produce even more aggressive optimizations. Previous work on optimization-unstable code detection [52] reported that 40% of the 8575 C/C++ Debian Wheezy packages they tested, contain unstable code that may get discarded during the optimization because of the presence of undefined behaviour.

Our goal is to design an equivalence checker that can compute equivalence across the optimizations produced by modern compilers. In contrast to previous work on translation validation [39, 24, 52], which assumed a certain set of compiler optimizations, we consider the compiler as a black-box, i.e., we do not assume the knowledge of the exact set or order of the transformations produced by the compiler. Further, we perform equivalence check across multiple composed optimizations, unlike the pass-by-pass approach taken by previous work [39, 24].

We present the design and an implementation of an equivalence checker that computes

equivalence across modern compiler optimizations. Our algorithm meets the requirements of the synthesis tools and can verify the transformations produced by multiple modern compilers. Our contributions towards this goal are:

- A new algorithm to determine the proof of equivalence across programs. The algorithm is robust with respect to modern compiler transformations and in a black-box manner, can handle almost all composed transformations performed by modern compilers.

- New insights in equivalence checking, the most important being handling of language level undefined behaviour based optimizations. Previous work had disabled these optimizations, yet we find that these optimizations are very commonly used in compilers. For example, our equivalence checking success rates increase by 15%-52%, through modeling some important classes of undefined behaviour.

- Comprehensive experiments: we evaluate our implementation across black-box optimizations produced by four modern compilers, namely GCC, LLVM (`clang`), ICC (Intel's C Compiler), and CompCert (`ccomp`). Our tool can automatically generate proofs of equivalence, across O2/O3 compiler transformations, for 74% of the functions in C programs belonging to the SPEC benchmark suite across all four compilers. These results are comparable (and, in some cases, better) to the success rates achieved by previous translation validation tools which operated in simplified settings (not black-box). This is a first of its kind experimental setup for evaluating an equivalence checker. We also test our equivalence checker for synthesis, by using it within a preliminary superoptimizer supporting loops, and present its initial results.

Section 2.1 discusses the notion of a simulation relation in the context of computing equivalence between unoptimized and optimized implementations of a C program. Section 2.2 presents our equivalence checking algorithm. Section 2.3 discusses the details of modeling undefined behaviour semantics, and Section 2.5 presents the results.

2.1 Simulation relation as the basis of equivalence

Two programs are equivalent if for all equal and legal inputs, the two programs have identical observables. We compute equivalence for C programs at function granularity. The inputs in case of C functions are the formal arguments and the memory (minus stack, i.e., the memory without considering the stack) at the function entry, and the observables are the return values and the memory (minus stack) at the exit. Two functions are equivalent if for the same input arguments and memory (minus stack), the functions return identical return values and memory state (minus stack). Equivalence is defined only for *legal* inputs, i.e., all inputs for which the program behaviour is well defined.

A simulation relation is a structure to establish equivalence between two programs. It has been used extensively in previous work on translation validation [39, 59, 48, 24, 42]. A simulation relation is a proof (or witness) of the equivalence between two programs, and is represented as a table with two columns: *Correlation* and *Invariants*. Given two programs $Prog_A$ and $Prog_B$, the correlation is a pair (L_A, L_B) such that L_A and L_B are PCs (program counter or location) in the two programs, and the invariants (I) are predicates in terms of the variables (i.e., states) at these respective PCs. A row $((L_A, L_B), I)$ of a simulation relation encodes that the invariants I hold whenever the two programs are at L_A and L_B respectively. A simulation relation is valid if it is inductively provable. For a valid simulation relation, the invariants at each correlated location should be provable from the invariants at the predecessor correlated locations (inductive case). Further, the invariants at the entry location (pair of entry points of the two programs) must be provable using the input equivalence condition (base case). If the equivalence of the required observables is provable at the exit location (pair of exits of two programs) using the invariants of the simulation relation, we can conclude that the programs are equivalent.

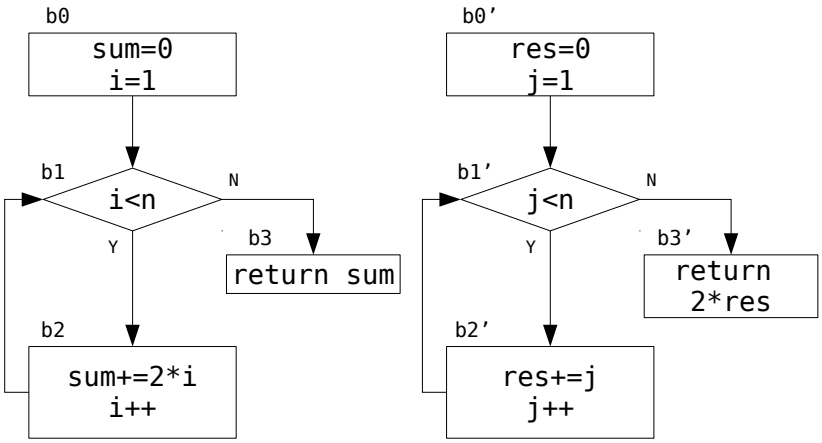
Formally, a simulation relation is valid if:

$$Init \Leftrightarrow invariants_{(Entry_A, Entry_B)}^{(S_{Entry_A}, S_{Entry_B})}$$

$$\underset{(L'_A,L'_B)\to(L_A,L_B)}{\forall} invariants_{(L'_A,L'_B)}^{(S_{L'_A},S_{L'_B})} \Rightarrow_{(L'_A,L'_B)\to(L_A,L_B)} invariants_{(L_A,L_B)}^{(S_{L_A},S_{L_B})}$$

Here $invariants_{(L_A,L_B)}^{(S_{L_A},S_{L_B})}$ represents the conjunction of invariants in the simulation relation in terms of the states S_{L_A} and S_{L_B} of the two programs at the respective locations L_A and L_B, $Init$ is the input equivalence condition at the entry of the two programs, and L'_A (L'_B) is a predecessor of L_A (L_B) in the program $Prog_A$ ($Prog_B$). Note that $\Rightarrow_{(L'_A,L'_B)\to(L_A,L_B)}$ is a special implication representing implication over the paths $L'_A \to L_A$ and $L'_B \to L_B$ in programs $Prog_A$ and $Prog_B$ respectively. It encodes the transfer functions $\tau_{L'_A\to L_A}$ and $\tau_{L'_B\to L_B}$ of the edges $L'_A \to L_A$ and $L'_B \to L_B$ respectively. Implication over an edge $(X \to Y)$ is defined as: $I_X^{S_X} \Rightarrow_{X\to Y} I_Y^{S_Y} \equiv (s_Y = \tau_{X\to Y}(s_X)) \wedge I_X^{S_X} \Rightarrow I_Y^{S_Y}$, or simply $I_X^{S_X} \Rightarrow I_Y^{\tau_{X\to Y}(S_X)}$. Here I_X represents the invariants at the location X.

Consider the example of Figure 2.1 to demonstrate a simulation relation in action. The figure shows two functions that compute the sum of first $n-1$ natural numbers, each multiplied by two. The second program is an optimized version, which avoids multiplication in the loop body. Note that the programs are equivalent, and one can prove the same by proposing a simulation relation between the two. Figure 2.2 shows a simulation relation between the two programs. It has three rows, one each for the entry (b0, b0'), the exit (b3, b3') and the loop-node (b1, b1'). The invariants at the entry and the exit represent the equivalence of the inputs (formal arguments and memory) and the outputs (return values and memory) respectively. We do not show the invariants relating the memory states, as the memory states of both the programs remain unmodified. The invariants at the loop-node are required for the inductive proof of correctness of the simulation relation. The invariants represent the relations which hold across the two programs, e.g., the invariant $i = j$ at the loop-node represents that the variable i of the first program at b1 is equivalent to the variable j of the second program at b1'. `Init` represents the input equivalence condition, which in this case is $n_A = n_B$. The only observable of this function is the return value, which can be proven from the simulation relation. The given simulation relation is valid and the invariants at the exit row can prove the equivalence of observables, i.e.,



(a) Unoptimized **(b)** Optimized

Figure 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 $2 * \sum_{j=1}^{n-1} j$.

Correlation	Invariants (I)
(b0,b0')	$n_A = n_B$
(b1,b1')	$i = j, sum = 2 * res, n_A = n_B$
(b3,b3')	$sum = 2 * res$

Init: $n_A = n_B$

Figure 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. `Init` is the initial condition representing the equivalence of inputs.

$sum = 2 * res$.

This simulation relation based technique can only prove equivalence across bi-similar transformations, e.g., it cannot prove equivalence across the loop tiling transformation. Fortunately, most compiler transformations preserve bi-similarity. Further, our notion of equivalence does not model constructs like exceptions, interrupts and concurrency. Essentially, we model the semantics of the sequential part of the C programming language, while comparing well-defined sequential C programs with its compiled output.

As per the C standard, non-termination in C language is undefined if a non-terminating loop does not produce any observables in the loop body, otherwise, it is well defined [5].

In the first case, compiler is allowed to produce a code of its choice, and compilers usually remove the non-terminating loops altogether. A simulation relation based technique would not work in such a case, unless, we model this undefined behaviour. In the second case, compilers cannot perform the loop removal, and, a simulation relation based technique capability is dependent on the capability of the inference algorithm. If the required invariants and correlation can be inferred, the algorithm will establish the equivalence.

In contrast with checking the correctness of a simulation relation, constructing a simulation relation is harder and is in fact undecidable in the generality of unlimited memory. The goal of our equivalence checking algorithm is to try and construct a valid simulation relation that can prove the equivalence. The next section presents the details of our algorithm, which tries to infer a simulation relation that can establish equivalence.

2.2 Black-box equivalence checking algorithm

Significant literature exists on sound equivalence checking of programs in the space of translation validation and verification [24, 8, 39, 48, 13, 25, 26, 51, 42, 30, 31, 52, 36, 55]. Most of these techniques are based on inferring a *simulation relation* (or bisimulation relation in some papers) between the two programs.

Previous work on equivalence checking has proposed different algorithms, to infer the correlation and invariants, that work in different settings and with different goals. We propose an algorithm to determine a simulation relation that works across *black-box* compiler transformations, which distinguishes our work from all previous work. There are three broad improvements we make over previous work:

1. A robust algorithm for finding the correlation of program points. The algorithm incrementally constructs the simulation relation (correlating one edge in each step), simultaneously inferring the invariants (using a guess-and-check procedure). At each step, there may be multiple correlation choices and our algorithm backtracks if it cannot find a valid edge correlation at any step or if the equivalence cannot be

established with the current correlation. Our algorithm does not make assumptions about the nature of transformations, and it is the first to be demonstrated to work across black-box compiler transformations.

2. We present a systematic guess-and-check based inference of invariants without making assumptions on the transformations performed. Our careful engineering of the guessing heuristics to balance efficiency and robustness is a novel contribution, and we evaluate it through experiments. Previous translation validation approaches made more assumptions on the nature of transformations, and hence would not apply to our setting. The robustness of our guessing strategy is significant to achieving good results in our setting.

3. We model language level undefined behaviour semantics. Our prototype can handle C programs and some common types of undefined behaviour in C language. Previous work on translation validation disabled transformations that exploit undefined behaviour (discussed in Section 2.3).

2.2.1 Introducing the algorithm through an example

We describe our algorithm with the help of an example program of Figure 2.3. Figure 2.4a and Figure 2.4b show the abstracted, unoptimized (A) and optimized (B) versions of the program in Figure 2.3. The optimized program has been compiled by `gcc` using -O3 flag. While the programs are in x86 assembly, we have abstracted them into C like syntax and flow charts for readability and exposition. The program has undergone multiple optimizations like 1) loop inversion, 2) condition inequality ($i < n$) to condition disequality ($i \neq n$) conversion, and 3) usage of conditional move instruction (`cmov`) to get rid of a branch. To our knowledge, no previous work can handle this transformation. All of the previous work fail in proving this transformation correct, either due to one or multiple optimizations (from the above three).

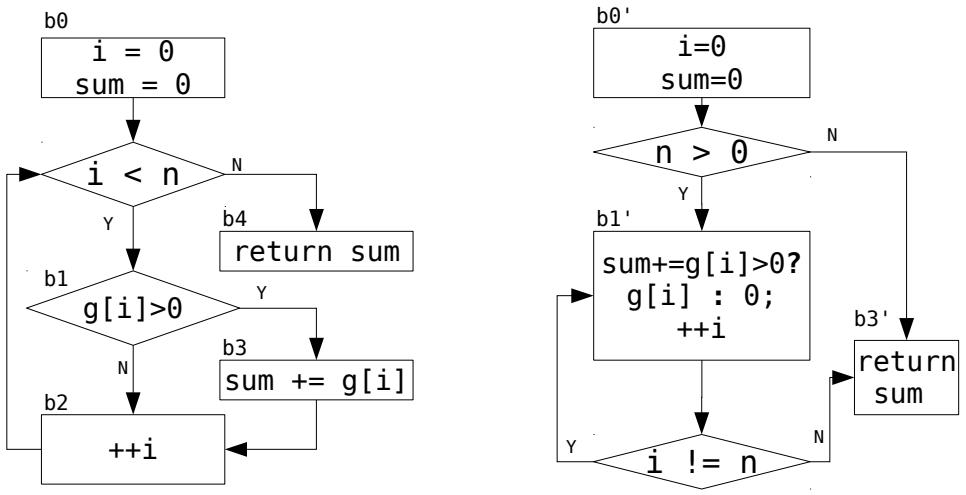
We now discuss how our algorithm computes a valid simulation relation; a simulation

```
int g[144];
void sum_positive(int n)
{
  for(int i = 0; i < n; i++)
  {
    if (g[i] > 0)
      sum = sum + g[i];
  }
  return sum;
}
```

Figure 2.3: An example function computing sum of positive integers of global array `g`.

relation is represented using a *joint transfer function graph* (JTFG), which is constructed incrementally at each step. A JTFG is a correlation between two *transfer function graph*s (TFGs). We represent programs as TFGs; a TFG consists of nodes and edges, where a node represents program counter (PC) and an edge models the control flow transfer from one PC to another. A JTFG represents a correlation across nodes and edges of the two programs (TFGs). A JTFG node represents two PC values, one belonging to the first program and the other to the second program. Similarly, a JTFG edge represents one control flow edge in the first program and its correlated edge in the second program. Further, we require that for two edges to be correlated in a JTFG, they should have equivalent *edge condition*, i.e., if one program makes a certain control transfer (follows an edge), the other program will make a corresponding control transfer along the respective correlated edge in the JTFG, and vice-versa. The individual edges of an edge of a JTFG could be composite: a composite edge (Section 2.2.3) between two nodes is formed by composing a sequence of edges (into a *path*), or by combining a disjunction of multiple paths (an example of a composite edge involving a disjunction of multiple paths is available in the following discussion). Please refer to Section 2.2.2 and Section 2.2.3 for the definitions of TFG and JTFG respectively.

Determining the correlation across program points and control transfers is one of the most involved problems during the construction of a simulation relation. Our algorithm proceeds as follows. We first fix the program points (PCs) and composite edges in one



(a) Unoptimized **(b)** Optimized

Figure 2.4: Abstracted versions of unoptimized and optimized implementations of the program in Figure 2.3. The ternary operator a?b:c represents the `cmov` assembly instruction.

program (say $Prog_B$) and try to find the respective correlated program points and composite edges in the other program ($Prog_A$). For sound reasoning of loops, we ensure that a correlation exists for at least one node in a loop, for all the loops. We pick the entry, the exit and the loop heads in $Prog_B$, as the *anchor* PCs (details in Section 2.4.1) that need to be correlated with the PCs in $Prog_A$. In our example, we pick (b0', b1', b3') in $Prog_B$. Thus, $Prog_B$ can be represented through the three anchor nodes, and a set of four composite edges, $edges_B$ =(b0'-b1', b1'-b1', b1'-b3', b0'-b3'). We now try and find the correlated composite edges in $Prog_A$ for each composite edge in $Prog_B$. Finally, when all the composite edges of $Prog_B$ get correlated, we obtain a candidate correlation between the two programs.

Running our algorithm on the example, we initialize the JTFG with its entry node (b0, b0'). We pick an $edge_B$ from $edges_B$ (sorted in DFS order) and find the list of composite edges in $Prog_A$ (up to some fixed length) that can be correlated with $edge_B$. For edge (b0'-b1') of $Prog_B$ we get (b0-b1, b0-b4, b0-b1-b2, b0-b1-b3, b0-b1-b3-b2, b0-b1-b2||b0-b1-b3-b2)[1] as the list of potential composite edges in $Prog_A$, up to unroll

[1] a-b-c is sequential composition of edges a-b and b-c. a-b-c||a-d-c is parallel composition of edges a-b-c and a-d-c. Please refer to Section 2.2.3 for details on composite edges.

factor 1 (unrolling the loop once). The last edge involves a disjunction of two paths. The conditions of these edges are $(0 < n_A, 0 \geq n_A, 0 < n_A \wedge g[i_A] \leq 0, 0 < n_A \wedge g[i_A] > 0, 0 < n_A \wedge g[i_A] > 0, 0 < n_A)$ respectively. The condition of the current $edge_B$ is $0 < n_B$. However, the edge conditions of the two programs cannot be compared because there is no relation between n_A and n_B. Before comparing the conditions across these two programs, we need to find invariants which relate the variables of the two programs at (b0, b0'). In this example, we require the invariant $n_A = n_B$ at (b0, b0') (we later discuss how to obtain such invariants). Invariant $n_A = n_B$ proves that the conditions of (b0-b1) and (b0'-b1') are equal, implying that the edge (b0'-b1') can be correlated with the edge (b0-b1). This pair of correlated edges is potentially a valid edge in the current JTFG, from the node (b0, b0') to the node (b1, b1'), and it gets added as a single edge to the current JTFG. We then try to correlate the next composite edge of $Prog_B$ until all the composite edges are correlated, and a simulation relation (JTFG) is found which can prove the required equivalence. At each step, it is possible for multiple composite edges in $Prog_A$ to have the required edge condition for correlation, while only one (or a few of the choices) may yield a provable simulation relation. To handle this, our algorithm backtracks to explore the remaining choices for correlated edges (discussed later). The final JTFG and invariants for the example programs are shown in Figure 2.5.

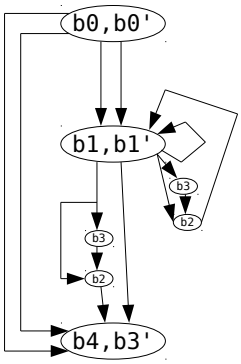
At each step of the algorithm, a partial JTFG gets constructed. For future correlation, we need to infer the invariants at the nodes of the currently constructed partial JTFG. We use a guess-and-check strategy to infer these invariants. This is similar to previous work on invariant inference (e.g., Houdini [15]), except that we are inferring these invariants on the JTFG, while previous work used this strategy for inferring invariants of an individual program. This guess-and-check procedure is formalized in Section 2.2.4.

The constructed JTFG may be incorrect on several counts. For example, it is possible that the invariants inferred at intermediate steps are stronger than what is eventually provable, and hence we infer an incorrect correlation. An incorrect correlation would mean that we will fail to successfully correlate in future steps of the algorithm, or will finish with

Correlation	Invariants (I)
(b0,b0')	$n_A = n_B, g_A = g_B, sum_A = sum_B,$ $M_A = M_B$
(b1,b1')	$sum_A = sum_B, n_A = n_B, i_A = i_B,$ $sl_4(M_A, g_A + i_A) = sl_4(M_B, g_B + g_B),$ $g_A = g_B, M_A = M_B,$ $i_B + 1 \leq n_B$
(b4,b3')	$sum_A = sum_B, M_A =_\Delta M_B$

Init: $n_A = n_B, g_A = g_B, sum_A = sum_B, M_A = M_B$

(a) Simulation relation



(b) JTFG

Figure 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. `Init` is the initial conditions representing equivalence of inputs. Operator sl_4 is a shorthand of SMT operator $select$ 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.

a simulation relation that cannot prove observable equivalence. To handle either of these cases of incorrect correlation, our algorithm backtracks to try other potential composite edges for correlation, unwinding the decisions at each step. In theory, the algorithm is exponential in the number of edges to be correlated, but in practice, backtracking is rare, especially if the candidate edges for correlation are heuristically prioritized. Our simple heuristic to minimize backtracking is to explore the composite edges in the order of increasing depth, up to a given maximum limit controlled by unroll factor. This heuristic is based on the assumption that a majority of the compiler transformations do not perform unrolling, and can thus be proven at smaller depths. If the algorithm succeeds in finding a JTFG that proves observable equivalence at the exit node, we have successfully computed a provable simulation relation, and hence completed the equivalence proof.

2.2.2 Transfer function graph

We need an abstract program representation as a logical framework for reasoning about semantics and equivalence. This abstraction is called the transfer function graph (TFG). A TFG is a graph with nodes and edges. Nodes represent locations in the program, e.g.,

$$
\begin{array}{lll}
\mathbb{T} & ::= & ([\varepsilon], [\varepsilon], [\varepsilon], \mathbb{G}([node], [edge])) \\
node & ::= & pc(int) \mid exit(int) \\
edge & ::= & (node, node, edgecond, \tau) \\
edgecond & ::= & state \rightarrow \varepsilon \\
\tau & ::= & state \rightarrow state \\
state & ::= & [(string, \varepsilon)] \\
\varepsilon & ::= & var(string) \mid nry\_op([\varepsilon]) \mid select(\varepsilon, \varepsilon, int) \mid \\
 & & store(\varepsilon, \varepsilon, int, \varepsilon) \mid uif([\varepsilon])
\end{array}
$$

Figure 2.6: Grammar of transfer function graph ($\mathbb{T}$).

program counter (PC). Edges encode the effect of the instructions and the conditions under which the edges are taken. In our setting of C program implementations, the state of the program consists of bitvectors and a byte-addressable array, representing registers and memory respectively.

A simplified TFG grammar is presented in Figure 2.6. The TFG $\mathbb{T}$ consists of preconditions, inputs, outputs and a graph $\mathbb{G}$ with nodes and edges. A node is named either by its PC location ($pc(int)$), or by an exit location ($exit(int)$); a TFG could have multiple exits. An edge is a four-tuple with from-node and to-node (first two fields), its edge condition $edgecond$ (third field) represented as a function from state to expression, and its transfer function τ (fourth field). The transfer function τ is a function from input program state to output program state along that edge. In other words, it is a functional representation of the effect of taking that edge on program state. An expression ε could be a boolean, bitvector, byte-addressable array, or an uninterpreted function; $var(string)$ represents a named constant, $nry\_op([\varepsilon])$ represents nry operation on n expressions, $select$ and $store$ are read and write operations on arrays, and $uif([\varepsilon])$ models the function calls as uninterpreted functions. Expressions are similar to standard SMT expressions, with a few modifications for better analysis and optimization, e.g., unlike SMT, `select` and `store` operators have an additional third integer argument representing the number of bytes being read/written. An edge is taken when its $edgecond$ holds. An edge's transfer function represents the effect of taking that edge on the program state, as a function of the state at the from-node. A state is represented as a set of (string, ε) tuples, where the string names

the state element (e.g., register name) and ε represents the value expression corresponding the state element. Apart from registers and memory, the state also includes an "IO" element indicating I/O activity, that in our setting, could occur only due to a function call (inside the callee)[2]. A procedure's TFG will have an entry node, and a single return (exit) node.

The C function calls in programs are modeled as uninterpreted functions (`uif`) in the TFGs. For every function, a summary representing the inputs and the outputs of the function is computed. The inputs of a function are memory (heap and globals), IO element, and its formal arguments, and the outputs are memory (heap and globals), IO element, and the return register (i.e, `eax`). This function summary is computed using an alias analysis (Section 2.3.3) and various ELF headers (Section 2.5), and represents the global variables that are read or written by this function. The summary also indicates whether the heap is read or written by the function. Finally, at each call-site, the effect of a function call is captured by assigning every "output variable" of the callee through uninterpreted function calls (`uif`); the inputs of the `uif` are based on the function call arguments and the global variables/heap read by the callee (as determined by the callee summary). The output variables of a callee capture the return value of the function call and any side effects, i.e., global variables/heap written-to by the function call. The callee's summary is conservative, and hence this model is sound but not complete.

Figure 2.7 shows the TFGs of unoptimized and optimized versions of the programs of Figure 2.4.

2.2.3 Joint transfer function graph

A joint transfer function graph (JTFG) is a subgraph of the cartesian product of the two TFGs. Additionally, each JTFG node has predicates (second column of simulation relation) representing the invariants across the two programs. Intuitively, a JTFG represents

[2]In the programs we consider, the only method to perform I/O is through function calls (that may internally invoke system calls).

Edge	condition	$\tau(sum, i, n, M) =$
b0-b1	$n > 0$	$(0, 0, n, M)$
b0-b4	$n \leq 0$	$(0, 0, n, M)$
b1-b2	$sl_4(M, g+i) \leq 0$	(sum, i, n, M)
b1-b3	$sl_4(M, g+i) > 0$	(sum, i, n, M)
b3-b2	$true$	$(sum + sl_4(M, g + i), i, n, M)$
b2-b1	$i + 1 < n$	$(sum, i + 1, n, M)$
b2-b4	$i + 1 \geq n$	$(sum, i + 1, n, M)$

(a) TFG representation of the program of Figure 2.4a.

Edge	condition	$\tau(sum, i, n, M) =$
b0'-b1'	$n > 0$	$(0, 0, n, M)$
b1'-b1'	$i + 1 \neq n$	$let\ u = sl_4(M, g + i)$ $(sum + (u > 0?u : 0), i + 1, n, M)$
b1'-b3'	$i + 1 = n$	$let\ u = sl_4(M, g + i)$ $(sum + (u > 0?u : 0), i + 1, n, M)$
b0'-b3'	$n \leq 0$	$(0, 0, n, M)$

(b) TFG representation of the program of Figure 2.4b.

Figure 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 $select$ of size 4. Therefore, $sl_4(M, g + i)$ represents $g[i]$.

a correlation between two programs: it correlates the move (edge) taken by one program with the move taken by the other program, and vice-versa. Formally, a JTFG (J_{AB}) between $TFG_A = (N_A, E_A)$ and $TFG_B = (N_B, E_B)$ is defined as:

$$J_{AB} = (N_{AB}, E_{AB})$$

$$N_{AB} = \{n_{AB} | n_{AB} \in (N_A \times N_B) \wedge (\bigvee_{e \in outedges_{n_{AB}}} edgecond_e)\}$$

$$E_{AB} = \{(e_{u_A \to v_A}, e_{u_B \to v_B}) \,|(\{(u_A, u_B), (v_A, v_B)\} \in N_{AB) \wedge}$$
$$(edgecond_{e_{u_A \to v_A}} = edgecond_{e_{u_B \to v_B}})\}$$

Here N_A and E_A represent the nodes and edges of TFG_A respectively and $e_{u_A \to v_A}$ is an edge in TFG_A from node u to node v. The condition on n_{AB} (in N_{AB}'s definition) stipulates that the disjunction of all the outgoing edges of a JTFG node should be `true`.

The two individual edges ($e_{u_A \to v_A}$ and $e_{u_B \to v_B}$) in an edge of JTFG should have equivalent edge conditions. The individual edge (e.g., $e_{u_A \to v_A}$) within a JTFG edge could be a composite edge. Recall that a composite edge between two nodes may be formed by composing multiple edges between these two nodes into one. The transfer function of the composite edge is determined by composing the transfer functions of the constituent edges, predicated with their respective edge conditions. We use the `ite` (if-then-else) operator to implement predication. Two edges can be composed into one by either sequential composition or parallel composition. Formally, we define sequential and parallel composition as follows:

Sequential composition

$e1 = (u \to v, edgecond_1, \tau_1)$

$e2 = (v \to w, edgecond_2, \tau_2)$

$sequential(e1, e2) = (u \to w, edgecond_1 \wedge edgecond_2, \tau_2 \circ \tau_1)$

Parallel composition

$e1 = (u \to v, edgecond_1, \tau_1)$

$e2 = (u \to v, edgecond_2, \tau_2)$

$parallel(e1, e2) = (u \to v, edgecond_1 \vee edgecond_2, ite(edgecond_1, \tau_1, \tau_2))$

Sequential composition is applied to two edges in sequence, i.e, one edge goes from u to v and the other edge goes from v to w. Parallel composition is applied to two edges in parallel, i.e., both edges go from u to v. Figure 2.5 shows a JTFG for the programs in Figure 2.7.

2.2.4 Algorithm for determining the simulation relation

Our correlation algorithm works across black-box compiler transformations, which is the primary difference between our work and previous work. Algorithm 1 presents the pseudo

```
Function Correlate(TFG_A, TFG_B)
    jtfg ← InitializeJTFG(EntryPC_A, EntryPC_B);
    edges_B ← DfsGetEdges(TFG_B);
    proofSuccess = CorrelateEdges(jtfg, edges_B, μ);

Function CorrelateEdges(jtfg, edges_B, μ)
    if edges_B is empty then
        return ExitAndIOConditionsProvable(jtfg)
    end
    edge_B ← RemoveFirst(edges_B);
    fromPC_B ← GetFromPC(edge_B) ;
    fromPC_A ← FindCorrelatedFirstPC(jtfg, fromPC_B);
    cedges_A ← GetCEdgesTillUnroll(TFG_A,fromPC_A,μ);
    foreach cedge_A in cedges_A do
        AddEdge(jtfg, cedge_A, edge_B);
        PredicatesGuessAndCheck(jtfg);
        if IsEquivalentEdgeConditions(jtfg) ∧ CorrelateEdges(jtfg, edges_B, μ)
         then
            return true;
        else
            RemoveEdge(jtfg, cedge_A, edge_B);
        end
    end
    return false;

Function IsEquivalentEdgeConditions(jtfg)
    foreach e in edges(jtfg) do
        rel ← GetSimRelationInvariants(jtfg, GetFromPC(e));
        if ¬ (rel ⟹ (e_FirstEdgeCond ⟺ e_SecondEdgeCond)) then
            return false;
        end
    end
    return true;
```

Algorithm 1: Determining correlation. μ is the unroll factor.

code of our algorithm. Function `Correlate()` is the top-level function which takes the TFGs of the two programs, and returns either a provable JTFG or a proof failure. The JTFG (`jtfg`) is initialized with its entry node, which is the pair of entry nodes of the two TFGs. Then, we get the the edges ($edges_B$) of TFG_B in depth-first-search order by calling `DfsGetEdges()`. And finally, the initialized `jtfg`, $edges_B$, and μ are passed as inputs to the `CorrelateEdges()` function, which attempts to correlate each edge in $Prog_B$ with a composite edge in $Prog_A$.

`CorrelateEdges()` consumes one edge from $edges_B$ at a time, and then recursively calls itself on the remaining $edges_B$. In every call, it first checks whether all $edges_B$ have been correlated (i.e., the `jtfg` is complete and correct). If so, it tries proving the equivalence of the observables, through `ExitAndIOConditionsProvable()`, and returns the status of this call. However, if there are still some edges left for correlation (i.e., `jtfg` is not complete), we pick an edge ($edge_B$) from $edges_B$ and try to find its respective candidate composite edge for correlation in TFG_A. Because we are correlating the edges in DFS order, the from-node of $edge_B$ (say $\texttt{fromPC}_B \leftarrow$ `GetFromPC(edge`$_B$`)`) would have already been correlated with a node in TFG_A (say $\texttt{fromPC}_A \leftarrow$ `FindCorrelatedFirstPC(jtfg, fromPC`$_B$`)`). We next compute the composite edges originating at $\texttt{fromPC}_A$ in TFG_A, to identify candidates for correlation with $edge_B$. The function `GetCEdgesTillUnroll()` returns the list of potential composite edges ($cedges_A$) that start at $\texttt{fromPC}_A$, which can be correlated with $edge_B$.

We check every $cedge_A$ in $cedges_A$ for potential correlation in the `foreach` loop in `CorrelateEdges()`. This is done by adding the edge ($cedge_A$, $edge_B$) to the JTFG and checking whether their edge conditions are equivalent; before computing this equivalence however, we need to infer the predicates on this partial JTFG through `PredicatesGuessAndCheck()` (discussed next). These inferred predicates are required to relate the variables at already correlated program points across the two programs. If the edge conditions are proven equivalent (`IsEquivalentEdgeConditions()`), we proceed to correlate (recursive call to `CorrelateEdges()`) the remaining edges

in $edges_B$. If the conditions are not equal or the recursive call returns false (no future correlation found), the added edge is removed (`RemoveEdge()`) from `jtfg` and another $cedge_A$ is tried. If none of the composite edges can be correlated, the algorithm backtracks, i.e., the current call to `CorrelateEdges()` returns false. We now describe the subroutines of this algorithm, and the algorithm to infer the invariants is discussed afterwards.

`DfsGetEdges(tfg)`: First performs the depth-first search over `tfg`, and finally, returns the edges in the order of exploration.

`RemoveFirst(edges)`: Removes the first element from the list `edges` and returns the removed element.

`GetFromPC(edge)`: Returns the PC of the from node of the input $edge$. In other words, the function returns `fromPC` such that $edge = (\texttt{fromPC} \rightarrow \texttt{toPC})$.

`FindCorrelatedFirstPC(jtfg, fromPC`$_B$`)`: Iterates over the nodes of `jtfg` and returns `fromPC`$_A$ such that (`fromPC`$_A$`, fromPC`$_B$) is a node in `jtfg`.

`GetCEdgesTillUnroll(tfg, pc,` μ`)`: It returns the list of all composite edges in `tfg` that start at `pc` with a maximum unrolling of loops bounded by μ (unroll factor). The unroll factor μ allows our algorithm to capture transformations involving loop unrolling and software pipelining.

`AddEdge(jtfg, edge`$_A$`, edge`$_B$`)`: Adds the edge (`edge`$_A$, `edge`$_B$) to `jtfg`.

`RemoveEdge(jtfg, edge`$_A$`, edge`$_B$`)`: Removes the edge (`edge`$_A$, `edge`$_B$) from `jtfg`.

`GetSimRelationInvariants(jtfg, pc)`: Returns the conjunction of invariants in the simulation relation (`jtfg`) at `pc`.

`IsEquivalentEdgeConditions(jtfg)`: For each joint edge in `jtfg`, it checks if the invariants at the from node of the joint edge can prove that the two constituent correlated edges have equivalent $edgeconds$.

Predicates guess-and-check

`PredicatesGuessAndCheck()` is an important building block of our algorithm, and it is one of the elements that lend robustness to our algorithm. Previous work has relied on inferring a relatively small set of syntactically generated invariants (e.g., [39, 48]) which are usually weaker, and do not suffice for black-box compiler transformations. Like Houdini [15], we guess several candidate invariants generated through a grammar, and run a fixpoint procedure to retain only the provable invariants. The guessing grammar needs to be general enough to capture the required invariants, but cannot be too large, for tractability.

Guess: At every node of the JTFG, we generate candidate invariants from the set $\mathbb{G} = \{ \star_A \oplus \star_B, M_A =_{\star_A \cup \star_B} M_B \}$, where operator $\oplus \in \{<, >, =, \leq, \geq\}$. $\star_A$ and $\star_B$ represent the program values (represented as symbolic expressions) appearing in TFG_A and TFG_B respectively (including preconditions) and $M_A =_X M_B$ represents equal memory states except the region X. The guesses are formed through a cartesian product of values in TFG_A and TFG_B using the patterns in $\mathbb{G}$. For example, for the program of Figure 2.1, $E_A \oplus E_B$ is a subset of the candidate invariants that we will generate at the loop-node, where $E_L \in \{sum, sum + 2 * i, 2 * i, i\}$ and $E_R \in \{res, 2 * res, res + j, j\}$.

This grammar for guessing candidate invariants has been designed to work well with the transformations produced by modern compilers, while keeping the enumeration and proof obligation discharge times tractable.

Check: Our checking procedure is a fixpoint computation which eliminates the unprovable candidate invariants at each step, until only the provable candidate invariants remain. At each step, we try and prove the candidate invariants across a JTFG edge, i.e., prove the candidate invariants at the head of the edge, using the current provable candidate invariants at the tail of the edge, and the edge's condition and transfer function. The candidate invariants at the entry node of the JTFG are checked using the initial conditions across the two programs, represented by `Init`. `Init` consists of predicates representing

input equivalence (C function arguments and input memory state (minus stack)). At each step, the following condition is checked for every edge:

$$\underset{(X \to Y) \in edges}{\forall} edgecond_{X \to Y} \Rightarrow (c\_invariants_X \Rightarrow_{X \to Y} c\_invariant_Y)$$

Here $c\_invariants_X$ represents the conjunction of all the (current) candidate invariants at node X and $c\_invariant_Y$ represents a candidate invariant at node Y. $edgecond_{X \to Y}$ is the edge condition for the edge $X \to Y$. $\Rightarrow_{X \to Y}$ is the implication over the edge $X \to Y$ as defined in Section 2.1. For brevity, we have omitted the state superscripts that are shown in the equations of Section 2.1. If this check fails for some guessed candidate invariant $c\_invariant_Y$ at some node Y, then we remove that candidate, and repeat until a fixpoint is reached.

2.2.5 Evaluation: guarantees and supported optimizations

Our equivalence checking algorithm is sound by design, i.e., whenever our algorithm returns $true$, the programs are guaranteed to be equivalent. Moreover, it returns a machine checkable proof of equivalence (witness), which can be checked by a third party verifier. However, the algorithm is incomplete, i.e., there are possible cases of equivalent programs for which the algorithm cannot infer a proof of equivalence.

We prove the following two properties of our algorithm. The first one is that the invariants inferred by our incremental approach of JTFG construction is equivalent to performing a guess-and-check on the final JTFG. The second property describes the correlations that our algorithm explores.

Theorem 2.2.1 *The invariants found by the incremental construction of a JTFG (guess-and-check in each step) is same as found by running the same guess-and-check on the final JTFG.*

Proof 2.2.2 *Let J_i be the intermediate JTFG at the i^{th} step after adding i^{th} edge to J_{i-1},*

and J_n represents the final JTFG after all the n edges have been added. In the each step of the incremental construction of JTFG: an edge is added, candidate invariants are generated at the new node and fixpoint is computed (check is run). Say, the sequence of edges for computing the fixpoint at i^{th} step is S_i (an edge may be repeated in a sequence multiple times). That is, S_i leads to the fixpoint by proving on the sequence of edges S_i. This computation is represented as $fx(J_i, S_i)$ denoting the elimination performed by running the check step on J_i in the sequence of edges S_i. We prove that:

$fx(J_1, S_1)$, $fx(J_2, S_2)$, ..., $fx(J_n, S_n) \equiv fx(J_n, S_1 : S_2 : ... : S_n)$

The LHS represents the elimination performed in the incremental approach and it is equivalent to that in the guess-and-check procedure on J_n (non-incremental approach) in the sequence $S_C = S_1 : S_2 : ... : S_n$ (RHS). Here ':' represents concatenation operator. We use the following properties to prove the same.

Property1: The order/sequence of elimination of invariants, or otherwise, the sequence of picking the edges for elimination of invariants, does not impact the final set of invariants (i.e., the fixpoint) in the check phase of Houdini algorithm. Stated differently, any sequence of edges resulting in a fixpoint, would result in the same outcome [16].

Property2: $fx(J_i, S_i) \equiv fx(J_{i+1}, S_i)$

The elimination on J_{i+1} happens only on the edges that are in S_i. Since, the last edge added in J_{i+1} is not in S_i, therefore, the computation on both the sides is equivalent.

Property3: $fx(J_i, S_i)$, $fx(J_{i+1}, S_{i+1}) \equiv fx(J_{i+1}, S_i : S_{i+1})$

$\equiv fx(J_{i+1}, S_i)$, $fx(J_{i+1}, S_{i+1})$ *(using Property2)*

$\equiv fx(J_{i+1}, S_i : S_{i+1})$ *(equivalent representation)*

Repeatedly, applying Property2 and Property3 we get:

$fx(J_1, S_1)$, $fx(J_2, S_2)$, ..., $fx(J_n, S_n) \equiv fx(J_n, S_1 : S_2 : ... : S_n)$

Property1 tells us that $fx(J_n, S_1 : S_2 : ... : S_n)$ is the only result of guess-and-check on the final JTFG J_n (as the solution is unique), and we prove it to be equivalent to the result of the incremental approach.

Theorem 2.2.3 *Given that the guessing grammar is sufficiently strong for establishing simulation relation. For all the edges in $Prog_B$, we try all valid correlations up to the unrolling factor μ in the $Prog_A$.*

Proof 2.2.4 *For each edge $L'_B \rightarrow L_B$ in $Prog_B$ such that (L'_A, L'_B) is in the nodes of the current JTFG, we look for all outgoing edges in $Prog_A$ up to the unrolling factor μ, starting from the node L'_A except:*

1. *We do not explore the unreachable nodes from L'_A, as it definitely forms an invalid simulation relation.*

2. *We do not consider an edge in $Prog_A$ whose $edgecond$ is not proven equivalent to that of the edge in $Prog_B$. We prove that we eliminate only the edges that would also be marked invalid in the final JTFG using the following: (1) It is given that the guessing grammar is sufficiently strong for establishing a simulation relation implying that we would not eliminate a right edge in the final JTFG. (2) The invariant set at each node of a partial JTFG (using which we eliminate an edge) is always a superset of that of the final JTFG. This is direct result of the fact that the invariants set always decreases monotonically in Houdini [15, 16] and similar argument applies for the incremental JTFG construction.*

We have tested the algorithm and its implementation across standard loop transformations like loop inversion, loop peeling, loop unrolling, loop splitting, loop invariant code hoisting, induction variable optimizations, strength reduction, inter-loop strength reduction, SIMD vectorization, and software pipelining. On the other hand, it does not support loop reordering transformations that are not simulation preserving, such as tiling and interchange. Also, if the transformations involve a reduction in the number of loops (e.g., replacing a loop-based computation with a closed form expression), the algorithm, in its current form, may fail to construct the proof.

Non-loop transformations are well supported, our algorithm also works with: register

allocation, strength reduction, dead code elimination, instruction selection, branch elimination, constant folding and propagation, dead store elimination, and common subexpression elimination.

A detailed evaluation of our algorithm, along with modeling of undefined behaviour, is presented in Section 2.5.

2.3 Modeling undefined behaviour semantics

Programming languages have erroneous conditions in the form of erroneous program constructs and erroneous data. Language standards do not impose requirements on all such erroneous conditions. The erroneous conditions on which no requirements have been imposed by the standard, i.e., whose semantics have not been defined are called *undefined behaviour* (UB). Since the standard does not impose any requirements on UB, compilers are permitted to generate code of their choice in presence of the same. In other words, compilers can assume the absence of UB in the target program and are free to produce code without the checks for UB conditions. Further, they can produce more aggressive optimizations under such assumptions. For example, the C language standard states that writing to an array past its size is undefined. Hence, C compiler writers do not need to check the sanity of the array index during an array access. Moreover, aggressive compilers may even remove a sanity check if the same has been added by the programmer in her C program.

C language contains hundreds of types of undefined behaviour [29]. All modern compilers like `GCC`, `LLVM` and `ICC` are known to extensively exploit UB while generating optimized code (we provide some evidence in this thesis). Further, previous work on *optimization-unstable* code detection [57] reported that 40% of the 8575 C/C++ Debian Wheezy packages they tested, contain unstable code: unstable code refers to code that may get discarded during optimization due to the presence of UB. Undefined behaviour is clearly widespread. The need for UB has also been widely debated. On one hand, many

textbook optimizations rely on UB semantics. For example, consider a simple `for` loop in C: `for (int i=0; i<=n; ++i)`. Now if `n` equals `INT_MAX`, then this loop would never terminate, and it would be possible for `i` to be negative inside the loop body (because `i` would wrap around after `INT_MAX`). However, several optimizations would like to depend on the loop termination property, and the loop invariant that `i >= 0` inside the loop body. Fortunately, these invariants are valid, because signed integer overflow is undefined in C. This yields the assumption that `++i` can never wrap around, indirectly implying that it is illegal for `n` to be equal to `INT_MAX`, in this example. On the other hand, programmers are often annoyed by these "counter-intuitive" optimizations, and some of them go to the extent of disabling certain types of UB through flags provided by the compiler. For example, the Linux kernel build process disables signed integer overflow and type based strict aliasing UB assumptions in GCC [53, 54].

Undefined behaviour semantics and their exploitation by compilers for optimization imply that the compiler verification tools (e.g., translation validation) must model these semantics for more precise results. Similarly, synthesis tools and superoptimizers (e.g., [2]) must model such semantics, while comparing equivalence of the target program with the candidate synthesized program, for better optimization opportunity. An equivalence checking algorithm results in a *false negative*, i.e., incorrect equivalence failure if it does not model the required UB. Previous work on simulation-based equivalence checking in the context of translation validation [39, 59, 24, 42, 50, 55] has been done across selected compiler optimizations, disabling the ones that exploit language level UB. Because our goal is to perform end-to-end equivalence checks without disabling any particular optimization, it is important for a black-box equivalence checker to model UBs that are exploited by the respective compilers; otherwise, it yields poor results. This thesis addresses this issue and makes the following contributions:

- We extend the simulation relation by adding *assumptions* at each row of the simulation relation table, to model language level UB semantics. Equivalence is now

computed under these assumptions, i.e., the original program and the transformed program need to be equivalent only if the corresponding assumptions are $true$. If the assumptions are $false$, the programs are still considered equivalent even if their implementations diverge. We call this the *extended simulation relation* (extended with assumptions).

- We discuss the assumptions produced by different types of UB semantics and experimentally determine the types of UB that are most consequential to compiler-based optimization.

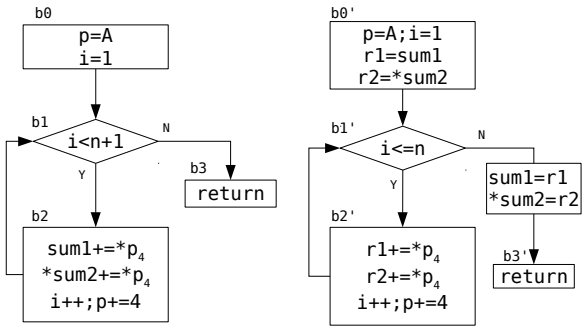
- To model aliasing based UB, which we find is heavily exploited by compilers for optimization, we present an algorithm to compute aliasing information at the IR/assembly level. Computation of aliasing information at the assembly level is necessary because the programs emitted by the compilers are in assembly. The aliasing information computed through this algorithm is used for generating UB assumptions for the extended simulation relation. Our alias analysis needs to be as precise as the compiler's alias analysis, to be able to reason about correctness of transformations that rely on such analysis.

2.3.1 Motivating example

Figure 2.8 shows a C program which computes the sum of the first `n` elements of a global array `A` and stores the result in a global variable `sum1` and at an address `sum2`. We have deliberately used two different types of accumulators (`int sum1` and `long* sum2`) and `i<n+1` in the `for` loop, to demonstrate three different types of C undefined behaviour in the same example. Figure 2.9a and Figure 2.9b show the abstracted unoptimized and optimized versions of the same program compiled by `gcc -O0` and `-O2` respectively. The original programs are in x86 assembly, and many other optimizations are present in the optimized version; for exposition and brevity, we have abstracted them into a C like syntax and only the UB related optimizations are shown.

```
int A[256];
int sum1 = 0; long* sum2;
void sum(int n) {
  int* p = A;
  for(int i=1;i<n+1;++i) {
    sum1 = sum1 + *p;
    *sum2 = *sum2 + *p;
    p++;
  }
}
```

Figure 2.8: An example function. `sum2` is allocated by the caller.



(a) Unoptimized **(b)** Optimized

Figure 2.9: Unoptimized and optimized, abstracted versions of the program in Figure 2.8.

The first optimization we discuss through this example, is a peephole optimization involving substitution of the check `i<n+1` by a faster check `i<=n`, avoiding the need to compute `n+1`. However, as such, the substitution may not seem correct because the two programs are not equivalent when `n=INT_MAX`. For `n=INT_MAX`, the loop of unoptimized program takes zero iterations (`INT_MAX+1` wraps around to a negative number `INT_MIN`), while that of the optimized program loops forever (because `i` will always be $\leq$`INT_MAX`). Interestingly however, it is legal and common for C compilers to perform this optimization. This transformation is legal due to the *signed integer overflow* (SIO) assumption, that forms a part of the C undefined behaviour semantics. As per this assumption, signed integer arithmetic *shall not*[3] overflow (i.e., it is an illegal program if it causes signed integer arithmetic to overflow), and hence, the compiler need not worry about the case when overflow takes place.

The second interesting optimization in this example is the register allocation of `sum1` and `*sum2` to registers `r1` and `r2` respectively, throughout the execution of the loop. These registers containing the accumulated sum values, are written back to their respective memory locations at loop exit. Again, as such, these transformations may not seem correct: it is possible for the pointer `p`, which can belong to `[A,A+4*n)` to alias with either (or both) of `&sum1` and `sum2`, in which case, the values stored at `p` may get mod-

[3]Phrasing is taken from the C standard.

ified as the loop executes, making register allocation of `sum1` and `*sum2` incorrect. It is however legal (and common) for C compilers to perform such register allocations. This is due to UB related to the following aliasing assumptions:

1. *Type based strict aliasing assumptions* (TBSA): Pointers of different types (e.g., `long*` and `int*`) shall not alias with each other (with the exception of `char*`).

2. *Out-of-bounds variable access assumptions* (OBVA): A program shall not access a memory location beyond the region of an object (variable).

In our example, the TBSA assumptions guarantee that `sum2` (of type `long*`) and `p` (of type `int*`) cannot alias. Similarly, `sum2` cannot alias with `&sum1` (of type `int*`). Further, the OBVA assumptions guarantee that `p` cannot point beyond the object `A`, i.e., `p` must belong to [A, A+4*256). This implies that `p` cannot alias with `&sum1`, as `sum1` and `A` are distinct regions. With these assumptions, it is indeed legal to register-allocate `sum1` and `*sum2` throughout the loop execution.

The unoptimized and optimized programs in Figure 2.9a and Figure 2.9b respectively can be shown to be equivalent only if the UB assumptions are modeled and used in the simulation-based proof. In this thesis, we contribute algorithms to model and use these UB assumptions in a simulation-based proof, and show their effectiveness for computing equivalence across compiler transformations on general-purpose code.

Figure 2.10 shows an *extended simulation relation* (defined in the next section) which can establish equivalence for the example program. This extended simulation relation has an assumption column in addition to the correlation and invariants. The assumption column represents the UB assumptions at the respective correlated location pairs. The assumptions encode the absence of modeled UB at the respective program locations. During the inductive proof of the simulation relation, assumptions of the predecessor node are also used for proving the invariants of successor nodes.

Section 2.3.2 presents the extended simulation relation, Section 2.3.3 describes how we model the undefined behaviour assumptions for use in the extended simulation rela-

Location	Assumption	Invariants (P)
(b0,b0')	True	$n_A = n_B, A_A = A_B, \&sum1_A = \&sum1_B, sum2_A = sum2_B, M_A =_\Delta M_B$
(b1,b1')	$(n_A \neq INT\_MAX) \wedge (\&sum1_A \neq p_A) \wedge (sum2_A \neq p_A) \wedge (sum2_A \neq \&sum1_A)$	$sl_4(M_A, \&sum1_A) = r1_B$, $sl_4(M_A, sum2_A) = r2_B, n_A = n_B, i_A = i_B$, $A_A = A_B, p_A = p_B, \&sum1_A = \&sum1_B$, $sum2_A = sum2_B, M_A =_{\Delta \cup \{\&sum1_A, sum2_A\}} M_B$
(b3,b3')	True	$M_A =_\Delta M_B$

`Init`: $n_A = n_B, A_A = A_B, \&sum1_A = \&sum1_B, sum2_A = sum2_B, M_A =_\Delta M_B$

Figure 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 `A` and `sum1` 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. `Init` represents equivalence of inputs.

tion, and finally, Section 2.3.4 evaluates the impact of different undefined behaviour on compiler optimizations.

2.3.2 Extended simulation relation (with assumptions)

We extend the simulation relation of Section 2.1 to support undefined behaviour assumptions, i.e., to allow equivalence computation in the presence of undefined behaviour (UB) semantics. Equivalence is now conditional on these assumptions, i.e., the equivalence proof may fail if these assumptions are discounted. The relevant assumptions are computed at each program location of the unoptimized program specification. These assumptions are based on a best-effort static analysis of the program: for example, if the program involves arithmetic on a signed integer variable, then the corresponding signed integer overflow (SIO) assumption is inferred at that program location. Some assumptions can be inferred directly from program syntax, while others may require a deeper static analysis. In general, the sophistication of the static analysis required to infer the undefined behaviour assumptions ought to match the sophistication of the analyses used by the optimizer. Signed integer overflow (SIO) and type-based strict aliasing (TBSA) assumptions are examples of assumptions that can usually be inferred through straight-forward syntac-

tic analysis of the program, while the out-of-bounds variable access (OBVA) assumptions usually require a deeper alias analysis, the kind used by modern compilers for optimization. We discuss this latter analysis in Section 2.3.3. In this section, we assume that such assumptions are already available at the respective program locations, and we discuss their effect on the required extended simulation relation.

Let $Prog_A$ be the unoptimized program specification and $Prog_B$ be the optimized implementation. $Prog_A$ specification also includes a map from the program locations to the corresponding UB assumptions ($Assum$). An extended simulation relation is represented as a table, where each row is a tuple ((L_A, L_B), $Assum[L_A]$, I) such that L_A and L_B are program locations in $Prog_A$ and $Prog_B$ respectively, $Assum[L_A]$ is the set of assumptions in $Prog_A$ at location L_A, and I is a set of invariants on the state elements (program variables) at locations L_A and L_B. A tuple $((L_A, L_B), Assum[L_A], I)$ represents that the invariants I hold whenever the two programs are at L_A and L_B respectively, *assuming* all the UB assumptions at *all* $Prog_A$ program locations ($Assum$) hold.

An extended simulation relation is valid if it is inductively provable. For a valid extended simulation relation, the invariants at each location pair are provable from invariants and UB assumptions at the predecessor location pairs. Notice that the UB assumptions do not need to be proven. Invariants at the entry location (pair of entry locations of the two programs) represent the equivalence of program inputs ($Init$); the base case of this inductive proof. Finally, if we can thus inductively prove equivalence of the observables at the exit location (pair of exits of the two programs), we have established the programs to be equivalent. Formally, an extended simulation relation is valid if:

$$Init \Leftrightarrow invariants_{(Entry_A, Entry_B)}$$

$$\underset{(L'_A, L'_B) \rightarrow (L_A, L_B)}{\forall} Assum[L'_A] \wedge invariants_{(L'_A, L'_B)} \Rightarrow_{(L'_A, L'_B) \rightarrow (L_A, L_B)} invariants_{(L_A, L_B)}$$

Note that the validity equations are similar to that of simulation relation (as discussed in Section 2.1) except the introduction of $Assum[L'_A]$. Here $invariants_{(L_A, L_B)}$ repre-

sents the conjunction of invariants in the extended simulation relation for the location pair (L_A, L_B), $Init$ is the input equivalence condition at the entry of the two programs, L'_A and L'_B are predecessors of L_A and L_B in programs $Prog_A$ and $Prog_B$ respectively, and $\Rightarrow_{(L'_A,L'_B)\rightarrow(L_A,L_B)}$ represents implication over the paths $L'_A \rightarrow L_A$ and $L'_B \rightarrow L_B$ in programs $Prog_A$ and $Prog_B$ respectively as defined in Section 2.1. Note that we have omitted the state superscripts (that are shown in the equations of Section 2.1) for brevity.

Following theorem justifies the extended simulation relation for proving equivalence of programs with undefined behaviour.

Theorem 2.3.1 *Given two C programs $Prog_A$ and $Prog_B$, and $Assum$ as the modeling of undefined behaviour at the respective locations (representing the absence of the modeled undefined behaviour in $Prog_A$). If a* valid *extended simulation relation across $Prog_A$ and $Prog_B$, with $Assum$, can prove the equivalence of observables, then the programs are equivalent.*

Proof 2.3.2 *Let $Assum_C$ represents all undefined behaviour assumptions as per the C standard for $Prog_A$. Note that the $Assum$ that we model will never be stronger than $Assum_C$, i.e., for all $X \in$ nodes of $Prog_A$: $Assum_C[X] \Rightarrow Assum[X]$. This could be due to either not modeling of all undefined behaviour, or incompletness in the modeling of undefined behaviour, as determining the same is undecidable.*

While checking the validity over each edge $(L'_A, L'_B) \rightarrow (L_A, L_B)$ of the extended simulation relation (i.e., the inductive check of the validity equations), we have two possibilities:

Case 1: $Assum[L'_A] = true$ (i.e., undefined behaviour not present or a weaker modeling)

The validity equation of extended simulation relation degenerates into the validity equation of simulation relation, i.e., both edges ($L'_A \rightarrow L_A$) and ($L'_B \rightarrow L_B$) simulate each other.

Case 2: $Assum[L'_A] = false$ (i.e., undefined behaviour is present)

The validity check passes vacuously, encoding the fact that there is no need to prove the invariants. Stated differently, all correlations and invariants are considered valid if some

undefined behaviour is present as per the C standard.

Collectively, (1) $Assum$ is not stronger than the actual $Assum_C$ (2) The extended simulation relation is valid under $Assum$ as per the validity equations and (3) The extended simulation relation can prove the equivalence of observables of $Prog_A$ and $Prog_B$, implies that the programs are equivalent.

Altogether, the extended simulation relation precisely captures the undefined behaviour specification of the C standard that a program should not trigger any undefined behaviour, otherwise, a compiler is allowed to generate a code (i.e., behaviour) of its choice.

Figure 2.10 shows an extended simulation relation which establishes the equivalence across the programs in Figure 2.9a and Figure 2.9b. The exit row of this extended simulation relation denotes equivalence of memory states (modulo stack and local variables) at the exit, representing the equivalence of globals variables {`sum1`, `A`} and values at pointer `sum2` and the remaining unused heap. This simulation relation is only provable when the UB assumptions are used in the inductive proof. For example, without the assumptions, the invariant $sl_4(M_A, \&sum1_A) = r1_B$ of the second row is not provable on edge (b1, b1') $\rightarrow$ (b1, b1') (sl_4 represents the memory-read of four bytes; see Figure 2.10 caption).

2.3.3 Modeling undefined behaviour assumptions

We now discuss how to obtain the undefined behaviour (UB) assumptions for the extended simulation relation. We first generate these assumptions on the unoptimized program specification $Prog_A$, for each location, through static analysis of the program. Later, at the time of the construction of the extended simulation relation, for every row (L_A, L_B), the assumptions corresponding to L_A are added in the extended simulation relation. In other words, the UB assumptions are inferred for the unoptimized program and used during the construction and proof of the extended simulation relation.

The algorithm to infer the UB assumptions depends on the type of the UB. For example, the assumptions for many types of UB can be inferred purely syntactically — see

Type of undefined behaviour	Description
Signed integer overflow (SIO)	Signed integer arithmetic cannot overflow
Type based strict aliasing (TBSA)	Pointers of different types cannot alias (barring exceptions like `char *`)
Dereferenced addresses not null	An address that has been dereference cannot be zero
Shift operand bounds	If a value X is shifted left/right by another value S, then $S \geq 0$ and $S < numbits(X)$ ($numbits(X)$ is the number of bits used to represent X)
Type alignment	A value X of type T must be aligned to the size of T
No divide by zero	The divisor of a division operation cannot be zero

Table 2.1: Examples of types of C undefined behaviour that can be modeled through syntactic analysis of the program.

Table 2.1 for some examples. Such syntactic analysis and modeling of UB has also been used previously for the verification of manually written peephole optimizations in LLVM [36].

The out-of-bounds variable access (OBVA) undefined behaviour assumptions are an example of UB that require a relatively deeper static alias analysis. This is because the production quality compilers typically implement a similar alias analysis for better optimization opportunity. The static alias analysis provides a *may-alias* relation between program pointers and program *variables*. The program variables include all global and local variables defined by the programmer. Further, to model aliasing in heap and stack, we include two special "variables", called "stack" and "heap". Thus, a pointer value in the program may alias with one or more of the user-defined variables, and/or with the stack/heap[4]. Based on this analysis, we infer assumptions indicating that a program pointer must point within the memory regions belonging to the variables with which it

[4]While a stack is not a part of the program's language level semantics, it gets introduced by the compiler in the assembly implementation.

may alias:

$$aliasing\_assumptions_p \Leftrightarrow \bigvee_{v \in \{u : may\_alias(p,u)\}} (p \geq v_{begin} \wedge p < v_{end})$$

Here p represents a pointer value, v is a program variable p, and $[v_{begin}, v_{end})$ represents the region of memory occupied by variable v. Further, invariants encoding the mutual-disjointness of regions associated with each program variable, and for the stack and heap, are added through conditions on the respective v_{begin} and v_{end} values. In our running example of Figure 2.8, the alias analysis infers that `p` may alias with only the array variable `A`. Further, because `A` and `sum1` are different variables, their memory regions are mutually disjoint, thus implying that `p` cannot alias with `sum1`.

Our alias analysis, to infer the variables with which a program pointer may alias, is similar to the previous work on alias analysis for assembly code [11]. The alias analysis need not be precise, but needs to be sound, i.e., the may-alias relation for a pointer p must include all variables that a pointer may actually alias with (over-approximation). We next describe the two analyses (linearly-related and may-depend-on) used by us to infer the may-alias relation.

May-alias analysis

To compute the may-alias relation, we first compute two relations, *linearly-related* (lr) and *may-depend-on* (dep) between program pointers and program variables (including stack and heap). The lr relation indicates the variable with which a program pointer is linearly-related, aka, *based-on* [11]. In other words, if a program pointer is at an offset from the address of a program variable then it is lr with that program variable. For example, the pointers `p1=v+10` and `p2=v+i` (for some arbitrary variable `i`) are both lr with the variable address `v`. On the other hand, `p=*v` is *not* lr with `v` (even though `p` *may depend on* `v`, as we discuss later). In our running example of Figure 2.8, `p` is lr with `A`. The C type

system guarantees that a pointer may be lr with at most one program variable[5]. Also, if a program pointer `p` is lr with a program variable `A`, then `p` may alias with `A`, and *cannot* alias with any other variable (including stack/heap). A pointer can at most be lr with one variable.

In addition to the lr relation, we compute another relation called "may-depend-on" dep. This relation indicates the variables on which a program pointer may depend on, i.e., the variables whose address may potentially influence the value of this program pointer. If the address of a variable may not influence the value of a pointer, then that pointer may be assumed to not alias with the aforementioned variable. Note that $lr(p, v)$ implies $dep(p, v)$.

The may-alias relation between a pointer p and program variable v is computed in terms of the linearly-related and may-depend-on relations as follows:

$$may\_alias(p, v) \Leftrightarrow dep(p, v) \wedge \bigwedge_{w \in (V - v)} (\neg lr(p, w))$$

Here V is the set of all program variables. In other words, we assume that a pointer p may alias with a variable v if it may depend on v, *and* it is not linearly-related to any other variable $w \neq v$ in V[6].

Computing linearly-related and may-depend-on relations

Computing both lr and dep relations involves a forward dataflow analysis on the program's control flow graph. These relations are initialized at the program entry with conservative assumptions, and they are computed at each intermediate program location by analyzing transfer functions of the incoming control-flow edges. In our setting, each program represents a C function body, and the calling conventions of the compiler are used to initialize the relations at the entry node, i.e., we assume that the function arguments *may*

[5] A violation of this type-system, through type-punning for example, falls into the realm of UB.

[6] As discussed earlier, the C type system ensures that if p is linearly-related to a variable w, then p cannot alias with any other variable $v \neq w$.

depend on any of the global variables and/or the heap, but are independent of the stack and local variables of the function. Further, we assume that the function arguments are *not* lr with any global variable. Together, these assumptions at program entry specify that the function arguments may alias with all the program's global variables and the heap, but cannot alias with the function's stack/local variables.

The lr analysis across a control-flow edge involves a simple syntactic analysis of the expression trees of the transfer function on that edge. Transfer function (τ) of each edge of a TFG returns the output state in terms of the input state. The different registers in the output state are expressions in terms of the registers of the input state as per the grammar of TFG (Figure 2.6). These expressions form expression trees with root as the output values. This syntactic analysis, over an edge, involves inference rules of the type: $lr(p, v) \Rightarrow lr(p \oplus X, v)$, i.e., if the input value (p) is known to be lr with v, then the output value of type $p \oplus X$ (for any expression X that may potentially depend on other variables $w \neq v$) is also lr with v. "$\oplus$" represents the addition and subtraction operators; we further generalize these rules to operations involving bitwise masking of lower-order bits of a pointer (a common operation in compiled code). If these inference rules cannot decide a pointer p to be lr with a variable v, then we conservatively assume that p is *not* lr with v (over-approximation). At all internal nodes (except the start node), we initially assume all pointers to be lr with all variables ($\top$), and refine the relations iteratively till a fixpoint is reached. As discussed earlier, at the start node, we assume that none of the function arguments are lr with any of the variables. This information on lr relations flows from the program entry to all intermediate program locations, through transfer functions. The meet operator for this lr dataflow analysis is *intersection*, i.e., a pointer is lr with a variable only if it is lr on *all* possible program paths. The lattice of lr analysis is formed by the $lr(p, v)$ facts of the pointers ($p \in P$ and $v \in V$), here P is the set of pointers and V is the set of program variables. During the data flow analysis, the $lr(p, v)$ fact flows only in one direction: $\top \rightarrow lr(p, v) \rightarrow \bot$, here $\top$ is unknown and $\bot$ is the worst-case information.

Similarly, the dep analysis across a control-flow edge also involves a syntactic analysis

	lr	dep
Function entry	$\neg lr(arg, global)$, $\neg lr(arg, heap)$	$dep(arg, global)$, $dep(arg, heap)$, $\neg dep(arg, stack)$
Dataflow rule	$lr(p, v) \Rightarrow lr(p \oplus X, v)$	$dep(p, v) \Rightarrow dep(OP(\ldots, p, \ldots), v)$
Meet operator	$intersection$	$union$

Table 2.2: Forward dataflow rules to compute lr and dep relations. $arg \in function\ arguments$, $global \in global\ variables$, $heap \in heap\ variables$, and $stack \in local\ variables$. $\oplus$ represents the addition or subtraction operators. OP is a function that uses p as an argument.

on the expression trees of the corresponding transfer function. The syntactic analysis involves inference rules of the type: $dep(p, v) \Rightarrow dep(OP(\ldots, p, \ldots), v)$, i.e., if p may depend on v, then any value derived from p (through any operation OP that uses p as an argument) may also depend on v. At the entry node, we conservatively assume that the function arguments may depend on any of the global variables or on the heap. At all intermediate nodes, we initialize by assuming that the pointers do not depend on any of the variables ($\top$). At each iteration, we refine this may-depend-on relation at every node by analyzing the expression trees of the transfer function of each incoming edge. The meet operator for the dep relation is ***union***, i.e., a pointer may depend on a variable if it depends on that variable on *any* program path. The lattice of dep analysis is similar to as that of lr analysis: $\top \rightarrow dep(p, v) \rightarrow \bot$, here $\top$ is unknown and $\bot$ is the worst-case information.

Table 2.2 lists the rules of lr and dep analyses succinctly. At the entry, we initialize the starting values of lr and dep with conservative assumptions (first row). Data flow rules encode the *kill* and *gen* sets of the data flow equations. If the data flow rule holds, then it a *gen* fact, else, it is a *kill* fact. Both the analyses are flow-sensitive, note that we compute the lr and dep values for different variables at different points.

Unlike compilers, our alias analysis needs to work for assembly code where pointer arithmetic is much more common. The lr relation is intended to capture such pointer arithmetic. Also, the modeling of stack is unique to assembly code. Our algorithm, which over-approximately computes the may-alias relation through lr and dep relations, is sound

and efficient (polynomial in the size of the program and quite fast in practice), and captures the common patterns in compiled code. A more expensive analysis can potentially yield more precise may-alias relations.

2.3.4 Evaluation

We perform experiments to demonstrate the impact of undefined behaviour (UB) on compiler optimizations. To demonstrate the same, we compute equivalence of C functions across unoptimized (-O0) and optimized (-O2) x86 binaries produced by compiling C programs through production compilers, GCC (v4.8) and LLVM (v3.6) with and without UB models. We disable function inlining during compilation, as our prototype implementation cannot reason about inter-procedural optimizations. Even after disabling inlining, the average speedup across the compiler optimizations on these programs is 1.72x over `clang-O0`. To be able to reconstruct the C-level information, required for modeling UB and equivalence checking, we enable a few additional flags during the compilation (namely `-g` and `-reloc`) to generate debug information and relocation headers respectively. We assume that the binaries contain the symbol table (i.e., are unstripped), which along with relocation headers allow accurate renaming of memory addresses to global variable symbols. Further, the debug headers provide the signature and types of the variables and the functions. Both GCC and LLVM support these compile-time options, and these options have no impact on the runtime of the executable.

The functions are drawn from four SPEC benchmarks: `bzip2` (compression utility), `gzip` (compression utility), `mcf` (combinatorial optimization) and `parser` (word processing). The number of global variables in these benchmarks are 100, 212, 43 and 223 respectively. We compiled each program with both compilers to produce 16 binaries (8 unoptimized and 8 optimized), representing a total of 1058 pairs of unoptimized and optimized assembly functions (ignoring the identical `glibc` functions). Among these pairs, 714 functions had at least one loop in them (cyclic functions). The average number of

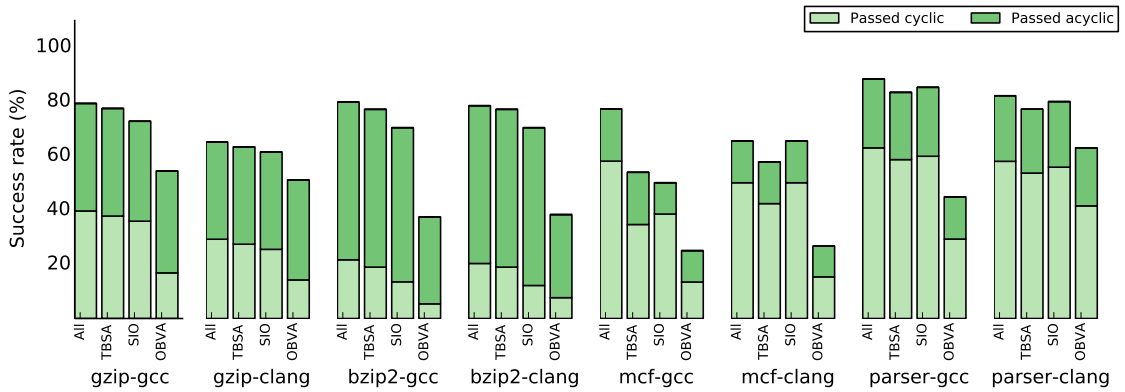


Figure 2.11: For every benchmark-compiler option, the first bar shows the success rates when we model all three UB. The remaining three bars show the success rates when a particular type of UB among three (TBSA, SIO, OBVA) is not modeled. Each bar individually shows the contribution to the success rates by cyclic (at least one loop) and acyclic functions.

assembly LOC and C-LOC for these functions is 112 and 35 respectively. We ignored the functions containing floating point operations (14 functions) as our semantic model for x86 floating point instructions is incomplete.

We performed experiments to demonstrate the significance of the three types of UB discussed in Section 2.3.1, namely signed integer overflow (SIO), type based strict aliasing (TBSA), and out-of-bounds variable access (OBVA) assumptions. We estimate the presence of UB based optimizations for each benchmark and compiler option by performing the equivalence check twice, for each function, with and without using the UB assumption. If an equivalence check for a function pair passes with the UB assumption but fails without the assumption, then we assume that the compiler has exploited the respective undefined behaviour towards optimizing the function. The plot in Figure 2.11 shows the success rates for each compiler and each benchmark for four different cases: the first bar represents the success rate when all three undefined behaviours are modeled; the second, third and fourth bars represent the cases when TBSA, SIO and OBVA assumptions are not modeled respectively. For SIO and TBSA, we employ the compiler flags `fno-strict-overflow` and `fno-strict-aliasing` to differentially estimate the impact of these assumptions. These flags enable/disable the SIO and TBSA assumptions while performing optimizations. If our equivalence check passes when these

assumptions are disabled by the compiler, but fails when these assumptions are enabled by the compiler, we assume that the compiler is leveraging these assumptions for optimization. For OBVA, we simply turn on/off our alias analysis (Section 2.3.3) to determine the effect of OBVA assumptions.

The overall average success rates for equivalence checking across the four cases are 81%, 76%, 77% and 50%. As expected, the success rates are lower when a certain type of UB is not modeled. The drop in success rates, when a UB is not modeled with respect to the first bar (where all three types of UB are modeled), indicates the impact of the respective type of UB on compiler optimization. The drop in success rates due to non-modeling of OBVA assumptions is 31 percentage points. In contrast, the drop due to non-modeling of SIO and TBSA assumptions is only 4 and 5 percentage points respectively. The drop due to non-modeling OBVA assumptions is significantly higher than the other undefined behaviour because of the following two reasons: (1) High number of global variables in the benchmark programs, e.g., parser benchmark has 200 globals. (2) Register allocation or otherwise reordering of memory accesses is a frequent and very important optimization.

These experiments confirm:

1. The widespread impact of undefined behaviours on compiler optimizations.

2. Throw light on the relative impact of different types of C undefined behaviour on compiler optimizations.

2.4 Implementation: optimizations and heuristics

In this section, we describe some important optimizations that helped scale and improve the results of the equivalence checking algorithm of Section 2.2.4.

2.4.1 Fixing nodes and edges in the optimized TFG

The first heuristic we use is collapsing of one of the TFGs in the algorithm of Section 2.2.4. In this step, we fix nodes and edges (that need to be correlated) in one of the TFGs. This step helps reducing the number of edges and nodes which we need to correlate with, and hence, reduces the number of nodes and edges in the final JTFG. The fixed nodes are also called *anchor* nodes. The rules of collapsing of a TFG are as follows:

1. We fix nodes and edges only in the optimized TFG of the program.

2. We pick the entry node and the exit node of the optimized TFG.

3. We pick at least one node for each loop in the optimized TFG. To be precise, we pick the 'head' node of every back-edge in the depth-first search (DFS) traversal of the optimized TFG.

4. We pick the nodes at which IO is performed.

5. Once the nodes are fixed, we collapse all the edges between every consecutive pair of fixed (anchor) nodes into a single composite edge (Section 2.2.3).

The TFG of the optimized program of Figure 2.4b shown in Figure 2.7 is in fact a collapsed TFG. The anchor nodes are {b0', b1', b3'}, and all the edges between two consecutive anchor nodes have been collapsed into the respective composite edges.

This heuristic helps reducing the number of nodes and edges of the optimized TFG, which results in having less number of nodes and edges in the constructed JTFG. Having fewer nodes and edges in a JTFG further results in requiring fewer number of candidate invariants and fewer number of SMT queries in the *guess-and-check* procedure, and therefore, allows the algorithm to scale for complex functions. Further, it allows reasoning across optimizations that subvert the branch structure of the program. While this heuristic has its advantages, the expressions of the composite edges may become complex, which, in turn, may result in complex queries to the underlying SMT solver. In our experience,

this heuristic works well and helps our algorithm to scale while computing equivalence across complex functions with large number of nodes and edges.

2.4.2 Heuristics for prioritizing guesses

The guesses are generated from the grammar $\mathbb{G}$, as discussed in Section 2.2.4. However, the number of possible guesses through $\mathbb{G}$ can be very large. Based on experiments, we categorized the guesses into two tiers: tier-1 (small and simple guesses), and tier-2 (guesses involving slightly larger expressions). We find that the SMT solvers can sometimes take a long time to decide tier-2 guesses, while most transformations can be decided through tier-1 guesses. To deal with this efficiently, we first attempt our complete proof procedure using *only* tier-1 guesses. If the proof fails due to tier-1 guesses, we attempt to use tier-2 guesses. The first phase (where only tier-1 guesses are used) is usually quite fast, and is able to decide around 99% of all successful equivalence checks (i.e., we did not need even a single tier-2 guess for these checks). Recall that our backtracking procedure which tries multiple correlations, allows even simple guesses to result in successful generation of equivalence proofs. Following is the exact nature of tier-1 and tier-2 guesses:

Tier-1 guesses include

1. Equating the values of registers and memory across each pair of correlated nodes.

2. Equating all memory read/write values in TFG_A with all memory read/write values in TFG_B.

3. Equating all arguments and return values of an uninterpreted functions correlated across the correlated edges of TFG_A and TFG_B.

4. Equating all values used as an address to a memory access, in a cartesian product fashion across the two TFGs.

5. Equating the edge-condition, and generating more guesses by under-approximating this equation (i.e., equating the subexpressions of the two edge-conditions' expressions), for all correlated edges of the two TFGs. For example, in Figure 2.1, the $edgecond$s of the paths b1-b2-b1 and b1'-b2'-b1' are $i < n$ and $j < n$ respectively. Equating the $edgecond$s results in $i < n \Leftrightarrow j < n$, and equating the subexpressions results in $A = B$ for all $A \in \{i, n, i < n\}$ and $B \in \{j, n, j < n\}$.

Tier-2 guesses further add

1. Equating all subexpressions of all expressions appearing in the transfer functions (τ) of each TFG edge across the two TFGs, in a cartesian product fashion. Recall that the transfer function of an edge of a TFG consists of expressions in terms of the input state. All the subexpressions of these expressions are equated with the subexpressions of the transfer functions in the other program.

For large programs, subexpressions could get relatively large and can cause SMT solvers to take a long time. Further, the number of subexpressions grows with increasing expression size. It is worth noting however, that both these categories of guessing (tier-1 and tier-2) produce guesses that are much simpler than the guesses based on weakest-preconditions as used in previous work (e.g., TVI [39]). We attribute the robustness of our equivalence checker, inspite of the simple nature of our guessing procedures, to our backtracking-based correlation algorithm.

2.4.3 SMT solver optimizations

Several optimizations were necessary to reduce SMT solver timeouts. We have developed our custom expression representation and a simplification pass over it before discharging the decision queries to the underlying SMT solver. The most important simplification is the 'short circuiting' of memory reads/writes. Our expression library supports higher level reads and writes on memory, in comparison to the expression libraries of SMT solvers. In

our expression library, `select` and `store` operations allow the specification of the data size, i.e., the size of the data that is read/written. The signatures of these operations are: `select(M, addr, size)` and `store(M, addr, size, data)`; here `size` specifies the number of bytes read/written on the memory `M` at the address `addr`, and `data` is the data written in case of a write. These higher level operations have direct correspondence with the reads/writes of multi-byte variables. Without the size field, the representation of a multi-byte read/write would be more complex; multiple bytes would have to be read and concatenated on a select, and multiple stores would have to be used to represent a multi-byte store. This usually results in complex expressions. The higher level representation involving size is easy to simplify and reason, and it is close to the compiler's view of the program. For example, this representation allows us to perform an important simplification, called 'short-circuiting', which results in an efficient discharge of proof obligations and is crucial in reducing the timeouts of the underlying SMT solvers. The simplification involves the following rewrite:

$$select(store(M, addr', size', data), addr, size) \Rightarrow select(M, addr, size)$$

where the address ranges $[addr', addr' + size')$ and $[addr, addr + size)$ do not overlap with each other. In other words, if the address of a previous *write* does not alias with the address of the *read* being performed, then we can discard the inner `store` corresponding to the write operation from the expression tree of the read operation.

We use Z3 [10] and Yices [12] as our SMT solvers, working in parallel to discharge each proof obligation (query). We use the result from whichever solver finishes first (wins) for each query. We found that Yices won 86% of the times across all queries, across all equivalence tests. However, it was also interesting to see that for some queries, Yices would take forever (beyond four hours), while Z3 would be able to return a result within a few seconds. One example of a pattern where this happens consistently, is the strength-reduction compiler optimization, where a compiler replaces a multiplication by a constant, e.g., 5*x, with a sequence of left-shift expressions, e.g., ((x $<<$ 2) + x), where x is some expression. Thus, using two solvers in parallel, indeed improved our efficiency and suc-

cess rates.

Another important optimization we implemented is caching of query results, as our algorithm involves proving similar predicates several times. For example, multiple product graphs are likely to have significant common subgraphs, and thus common predicates need to be proven at each correlation attempt. Similarly, Houdini's fixpoint algorithm requires the same predicates to be proven, multiple times, under different preconditions. Our cache for query results is designed to capture all these common patterns, and exhibits an overall hit rate of 95.3%; without caching, average runtime is around 4x worse.

2.5 Combined evaluation

We implemented our ideas into a tool which works with x86 assembly programs. Given two x86 programs, our tool either generates a machine checkable proof of equivalence or it fails. For checking equivalence across compiler optimizations, we compile multiple C programs by multiple compilers at different optimization levels, for x86, to generate unoptimized (-O0) and optimized (-O2 and -O3) binary executables. We then harvest functions from these executable files and reconstruct C-level information, necessary for modeling undefined behaviour assumptions and for performing equivalence checks. Once the functions are harvested and C-level information is reconstructed, we perform the equivalence checks between the functions from unoptimized (O0) and optimized (O2/O3) executables. We selected four compilers for this study: GCC (v4.8), LLVM (v3.6), ICC (v16.0) and CompCert (v2.5). GCC and LLVM are mainstream open source compilers, ICC is a proprietary compiler by Intel, and CompCert is a verified compiler.

The high level C program information necessary for performing the equivalence checking and modeling undefined behaviour are global variables and their scope/type attributes, local stack, function declarations and function calls, and program logic (function body). We reconstruct the language level semantics from ELF executables by using certain (standard) ELF headers. We rely on the debug headers (`-g`), symbol table and relocation table

(`-fdata-sections --emit-relocs`) for getting the required high level information. Debug headers contain information about the functions and their signatures. Symbol table provides the global variable name, address, size and binding attributes. The relocation headers allows precise renaming of addresses appearing in code, to the respective global variable identifiers with appropriate offsets, ensuring that the different placement of globals in different executables are abstracted away. None of these flags affect the quality of generated code. All these flags (or equivalent) are available in `gcc`, `clang`, `icc` and `ccomp`. Our reconstruction procedures are identical for both O0 and O2/O3 executables. The difference is that while the reconstructed information from O0 is used for obtaining the high level C program specification, the reconstructed information from O2/O3 is used only to help with proof construction.

We use the flags `fno-strict-overflow` and `fno-strict-aliasing` to disable the undefined behaviour assumptions related to signed integer overflow (SIO) and type based strict aliasing (TBSA) respectively. Also, we model the undefined behaviour related to out-of-bounds variable access (OBVA) as per the discussion of Section 2.3.3.

We have have classified our results into three categories: Section 2.5.1 presents the results of establishing equivalence, in a black-box manner, across compiler optimizations produced by multiple compilers, Section 2.5.2 presents the bugs discovered, and finally, Section 2.5.3 discusses some early synthesis results in the setting of superoptimization.

2.5.1 Equivalence checking across compiler optimizations

The C programs that we compiled for generating unoptimized and optimized binaries, are listed in Table 2.3 along with their characteristics. `ctests` is a program taken from the CompCert testsuite [33] and involves a variety of different C features and behaviour; the other programs are taken from the SPEC CPU2000 integer benchmarks. The SPEC benchmark programs do not include `gcc` and `eon` because their ELF executables files are very big, and our tool to harvest instruction sequences from executable files does not sup-

Bench	Fun	UN	Loop	SLOC	ALOC	Globals
mcf	26	2	21	1494	3676	43
bzip2	74	2	30	3236	9371	100
ctests	101	0	63	1408	4499	53
crafty	106	5	56	12939	72355	517
gzip	106	1	66	5615	14350	212
sjeng	142	3	68	10544	38829	312
twolf	191	17	140	17822	84295	348
vpr	272	69	155	11301	44981	153
parser	323	2	240	7763	30998	223
gap	854	0	466	35759	177511	330
vortex	922	5	116	49232	167947	815
perlbmk	1070	65	271	72189	175852	561

Table 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.

port such large ELF files. We also include an integer program from the SPEC CPU2006 integer benchmarks: `sjeng`. `sjeng` is one of the few C benchmarks in SPEC CPU2006 that is not already present in CPU2000, *and* has a low fraction of floating point operations. We avoid programs with significant floating-point operations, as our semantic models for x86 floating point instructions are incomplete. A total of 4% of the functions contain unsupported (usually floating-point) opcodes (Table 2.3), and we do not consider them in our evaluation. Figure 2.12 plots the performance of all benchmarks across different compiler optimization levels. On average, an optimized (O2/O3) executable is 1.9x faster than an unoptimized executable. Thus, our equivalence checks are performed across optimizations that exhibit this performance gap.

Success rates breakdown

Figure 2.13 plots the success rates for each benchmark-compiler-optimization pair. Each bar further shows the pass/fail percentage of the cyclic and the acyclic functions for a compiler-optimization pair. Overall, our tool is able to generate sound equivalence proofs across almost all transformations across multiple compilers for 76% of the tested function-

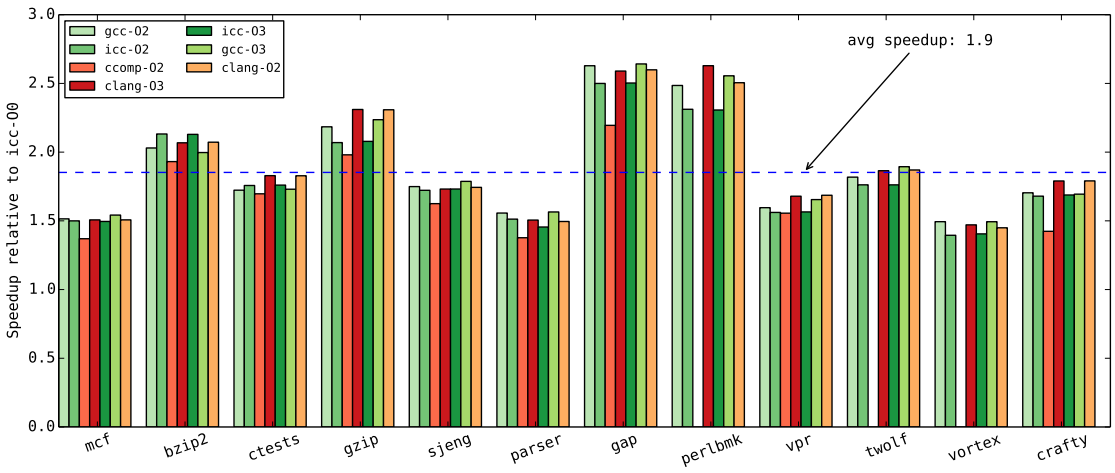


Figure 2.12: Performance of benchmarks across different compilers.

pairs for O2 optimization level, and 72% of the tested function-pairs for O3 optimization level.

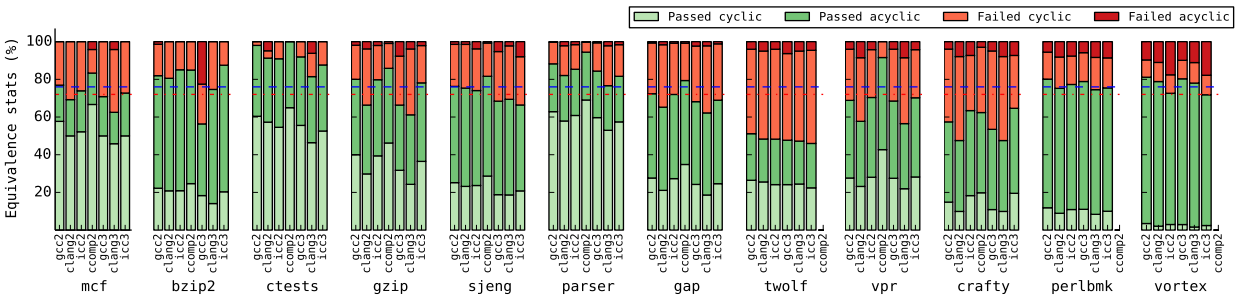


Figure 2.13: Equivalence statistics. Functions with at least one loop are called "cyclic". The bar corresponding to a compiler (e.g., `clang`) represents the results across O0/O2 and O0/O3 transformations for that compiler (e.g., for `clang2` and `clang3` resp.). The average success rate across 26007 equivalence tests on these benchmarks, is 76% for O2 and 72% for O3 (dashed blue and red lines resp.). The missing bars for `ccomp` are due to compilation failures for those benchmarks.

Success rates with ALOC

Figure 2.14 plots the success rates as a function of the number of assembly instructions in a function. There were 26007 function-pairs tested across all benchmarks and compiler/optimization pairs. The timeout value used was five hours. The success rates are much higher for smaller functions: for functions with ALOC of up to 105 and 345, the

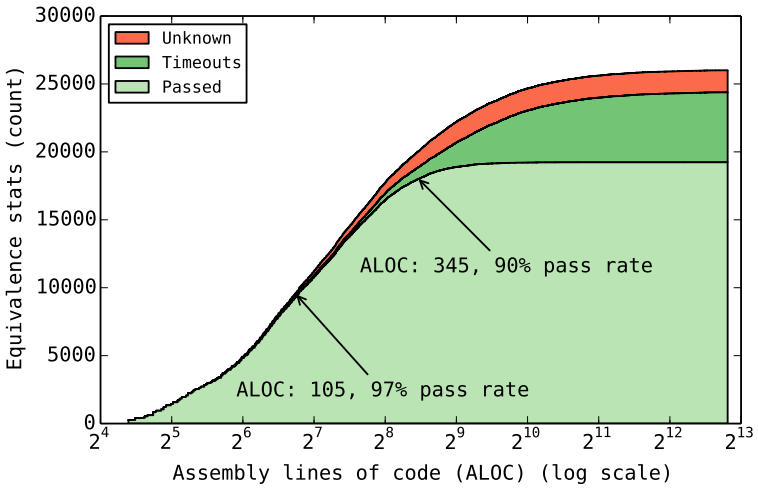


Figure 2.14: Cumulative success rate (pass/fail) vs. ALOC.

success rates are 97% and 90% respectively. The mean and median values for runtimes for passing equivalence tests are 313 seconds and 8.5 seconds respectively. 5% of the passing tests take over 1000 seconds to generate the result. Failures are dominated by timeouts (5 hours), inflating the mean and median runtimes for all (failing + passing) equivalence tests to 3962 seconds and 22 seconds respectively. The largest function for which equivalence was computed successfully has 4754 ALOC.

Success rates with TFG complexity

The size (ALOC) of a function is not the only indicator of its complexity. We plot the success rates against the number of composite edges in the TFG_B (Figure 2.15); the number of composite edges in a TFG is a direct indicator of the number of loops in the function (because we only consider loop-heads as intermediate nodes). For example, the maximum number of composite edges in a TFG with n loops is $(n+1)^2$ (e.g., if the TFG has one loop, it can have at most four composite edges). The plot indicates that while some failures exist even for acyclic sequences (primarily due to undefined behaviour, or uncaptured compiler behaviour), the percentage of failures increases with increasing TFG complexity. Yet, the success rates remain quite high (over 90%) even till TFGs with ten

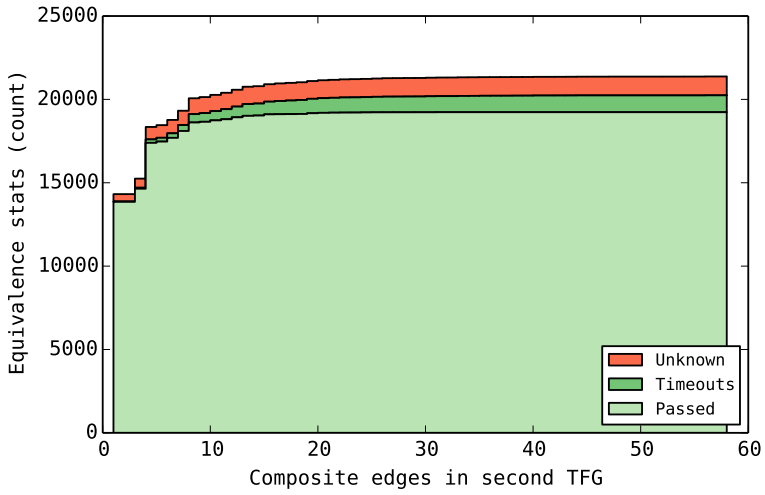


Figure 2.15: Success rate vs. composite edges in TFG_B.

composite edges. The most complex TFG for which our tool was successfully able to establish equivalence had 31 composite edges in the corresponding TFG_B.

Profiling

Most of the time taken by our algorithm goes into SMT query solving; across all equivalence tests, around 74% of the total time is spent in SMT query solving, of which 11% is spent in our simplification procedure of Section 2.4.3. Our tool spends roughly 96% of the time in determining the correlation between the two TFGs across all our experiments. Multiple JTFGs are tested during the depth-first search co-relation procedure (e.g., at each call to `IsEquivalentEdgeConditions()` in Algorithm 1), and our tool's running time is strongly co-related with the number of JTFGs tested. Figure 2.16a plots running time for an equivalence test (Y axis) against the number of JTFGs that had to be checked to identify a correct correlation for that test (X axis). Only passing equivalence checks are shown. It is not surprising that the running time usually increases with the number of JTFGs that had to be checked, but it is interesting to see that some equivalence checks take a long time, even for checking only a few JTFGs. The running time also depends on the complexity of the proof obligations within a JTFG, as that influences the running time

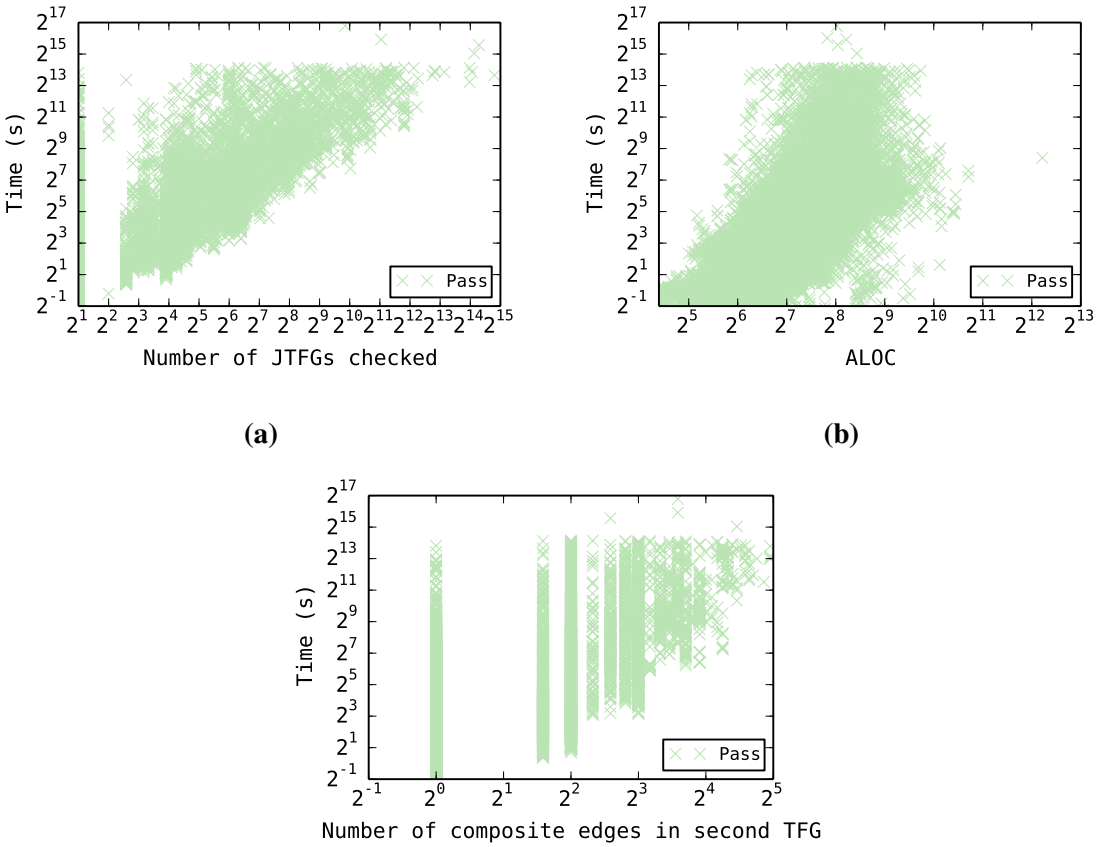


Figure 2.16: Time taken for equivalence test (Y-axis) plotted against number of (a) JTFGs checked, (b) ALOC and (c) number of composite edges in TFG_B. For acyclic programs, the number of JTFGs checked and the number of composite edges in TFG_B will be 2^1 and 2^0 respectively (independent of program 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.

of the SMT solver discharging those proof obligations.

Figures 2.16b and 2.16c plot running time for an equivalence test against the ALOC (assembly lines of code) of the programs being tested, and the number of composite edges in TFG_B, respectively. Both ALOC and the number of composite edges in TFG_B are indicators of the size and complexity of the tested programs. The running time seems co-related with ALOC: larger functions usually take more time, and smaller ones usually take less time. The running time also usually increases with the number of composite edges in TFG_B, because larger TFGs require deeper DFS traversals. It is interesting to

note that transformations across fairly complex programs with up to 31 composite edges in their TFG_B, can be handled by our tool.

Computing equivalence across different compilers

We additionally conducted equivalence checking across `compcert-O0` and `gcc-O2`, and the results are shown in Table 2.4. The success rates are slightly lower than the average because sometimes the symbol names used by different compilers in final ELF executables are different, resulting in inconsistent renaming, leading to equivalence failures.

Benchmark	Pass %
mcf	62.5
bzip2	56.3
compcert	96.8
gzip	67.6
crafty	45.9
sjeng	68.8
gap	71.6
parser	80.6
vpr	57.1
Overall Pass %	70.7

Table 2.4: Success rates of equivalence checking across `compcert-O0` and `gcc-O2`.

Reasons for incompleteness in our equivalence checker

The modeling of undefined behaviour due to memory underruns and overruns is crucial to get reasonable success rates for equivalence checking across black-box composed transformations produced by modern compilers. For example, without this modeling, our success rates are 15%-52% lower. Yet, there remain certain language-level semantics that we have not yet modeled in our tool. We discuss two interesting cases in detail, along with counter-examples (that we observed during our experiments), that will cause our equivalence checker to incorrectly fail. We then discuss some other reasons for equivalence failure, due to inadequate modeling of semantics in our tool.

1. A common reason for equivalence failures is the use of address-taken uninitialized variables as function arguments. Consider the following function:

```
int foo() {
  int a;
  bar(&a);
  return a;
}
```

   Here the programmer may maintain an invariant that the callee `bar` does not read the value in pointer `&a`, but we model this conservatively by assuming that `bar` can both read and write to variable `a`. This results in an equivalence failure. This can potentially be fixed by generating function summaries indicating whether they read or write or read/write their arguments.

2. An interesting reason for equivalence failures is due to compiler analysis of scope of global variables declared with the "static" keyword (e.g., global variables that are visible only within a compilation unit). Consider the following example:

```
static int a;
void foo() {
  a = 4;
  bar();
  return;
}
```

   If `bar` was declared in a separate compilation unit from `foo` (and `a`), the compiler may infer that `bar` may never be able to access `a` and so, it may reorder the call to `bar` and the assignment to `a` (`a = 4;`). However, our equivalence checker fails to compute equivalence across such reorderings. This can potentially be fixed by using the scope information of global variables.

A reason for equivalence failures is also our imprecise modeling of local variables — we model them as stack accesses, and hence cannot reason about undefined behaviour related to accesses to local variables. Another major cause for equivalence failures are global compiler optimizations (unrelated to undefined behaviour). Some examples of global compiler optimizations that cannot be captured by a function-granular equivalence checker, are: (1) Local array variables that are initialized at allocation time and are read-only in usage, being moved to global read-only data section in the ELF file, and converted to global variable access in the optimized implementation. (2) Different functions being called for C/C++ constructs in optimized and unoptimized implementations, e.g., different variants of malloc. (3) Accesses to read-only data (e.g., strings) converted to immediate values inside the optimized code. (4) Analysis of access pattern to global variables to reduce their size, e.g., if the only values written to a global 4-byte integer are 0 and 1, the integer is replaced by a one-byte character. (5) Functions not terminated using the `ret` instruction in the optimized implementations, because the compiler determines that a callee function never returns. There are more examples of compiler transformations, which utilize global invariants that a function-granular approach cannot address. Overall, we find that the gamut of compiler transformations is significantly richer than the ones supported by existing translation validation tools, including ours.

2.5.2 Bugs discovery

Our experiments led to the discovery of one bug in GCC-4.1.0 [18] and two bugs in ICC-16.0.3 [20, 21]. Each bug entails equivalence failures across multiple functions. Following are the details of the bugs found:

1. *GCC confirmed and fixed bug*: gcc supports the `-fno-strict-aliasing` option to disable undefined behaviour assumptions due to type-based aliasing, but we found a confirmed bug in its implementation in GCC. This is undesirable because the Linux kernel depends on this option, and GCC is the default compiler for compiling Linux. Here is description of the exact program that triggers the bug:

```
$ gcc -v
gcc version 4.1.0
$ cat a.c
struct list { int hd; struct list * tl; };
struct list *reverselist(struct list *l) {
  struct list * r, * r2;
  for (r = NULL; l != NULL; l=l->tl) {
    r2 = (struct list *)foo(sizeof(struct list));
    r2->hd = l->hd;
    r2->tl = r;
    r = r2;
  }
  return r;
}

$ gcc -O2 -m32 -S -fno-strict-alias a.c

# Showing the compiled assembly for the relevant portion
# of the loop body generated in a.s

Assembly gcc-4.1.0:
    movl  (%ebx), %eax  # eax   <- l.hd
    movl  %esi, 4(%edx) # r2.tl <- hd
    movl  %edx, %esi    #
A:  movl  4(%ebx), %ebx # ebx   <- l.tl
B:  movl  %eax, (%edx)  # r2.hd <- eax
    testl %ebx, %ebx    #
```

The read from `l->tl` (instruction A) has been reordered before the write to `r2->hd` (instruction B). Based on the documentation of `-fno-strict-aliasing`, this should not be possible, as `l->tl` could potentially alias with `r2->hd` in the loop body.

The bug was confirmed and has been fixed in later versions of gcc.

2. *ICC confirmed bug*: icc supports the `-fno-strict-aliasing` option to disable undefined behaviour assumptions due to type-based aliasing, but we found a confirmed bug in its implementation. The bug report was escalated within Intel, upon our reporting. Here is a description of the exact program that triggered the bug, and how our tool uncovers it.

```
$ icc -v
icc version 16.0.3
(gcc version 4.8.0 compatibility)
$ cat a.c
#include "stdlib.h"
char *foo(size_t size);
struct list { int hd; struct list *tl; };
struct list *reverselist(struct list *l) {
    struct list * r, * r2;
    for (r = NULL; l != NULL; l=l->tl) {
        r2 = (struct list *)foo(sizeof(struct list));
        r2->hd = l->hd;
        r2->tl = r;
        r = r2;
    }
    return r;
}

$ icc -O2 -m32 -S -falias -no-ansi-alias -fargument-alias a.c

# Showing the compiled assembly for
# the loop body generated in a.s

..B1.3: # Preds ..B1.2 ..B1.4
addl $4, %esp #9.25
pushl $8 #9.25

# foo(size_t) call foo #9.25
```

```
..B1.4: # Preds ..B1.3
movl %edi, 4(%eax) #11.5
movl %eax, %edi #12.5
movl (%esi), %ecx #10.14
movl 4(%esi), %esi #8.33
testl %esi, %esi #8.23
movl %ecx, (%eax) #10.5
jne ..B1.3 # Prob 82% #8.23
```

The write to `r2→hd` (instruction 10.5) has been reordered after the read from `l→tl` (instruction 8.33). Based on the documentation of `-no-ansi-alias`, this should not be possible, as `r2→hd` could potentially alias with `l→tl` in the loop body.

3. *ICC confirmed issue (developers non-committal on semantics)*: icc supports the `-fno-strict-overflow` option to disable undefined behaviour assumptions related to signed integer overflow semantics, but does not necessarily respect it. Certain software like the Linux kernel depend on the correctness of this option. We filed this issue on the icc mailing list, and a developer confirmed with the following response: "... icc has special treatments of int overflow for 32-bit mode which I haven't seen explained adequately and would hope not to rely on ...", and " ... For all I know, Intel may have worked to eliminate such dependencies from kernel.". In other words, Intel developers are non-committal on the semantics of `-fno-strict-overflow`. This confusion on the semantics of an important compiler flag, seems undesirable.

   Here is an example where the equivalence check failed due to icc (incorrectly) violating the semantics of `-fno-strict-overflow` option:

```
$ icc -v
icc version 16.0.3
(gcc version 4.8.0 compatibility)
```

```
$ cat b.c
void foo(int fl) {
  long i;
  for (i = 0; i < fl + 1; i++) {
      printf("%d\n", i);
    }
}

$ cat main.c
#include <limits.h>
void foo(int fl);
int main() { {
  foo(INT_MAX);
}

$ icc -m32 -O0 -fno-strict-overflow main.c b.c -o O0.out
$ icc -m32 -O2 -fno-strict-overflow main.c b.c -o O2.out
$ ./O0.out
<no output>

$ ./O2.out
0
1
2
...
```

These are subtle issues of contract violation between the compiler vendor and the software vendor that, if violated, could cause subtle bugs and security issues. Both `-fno-strict-aliasing` (or `-falias` and `-ansi-alias` options in icc) and `-fno-strict-overflow` are supported by all compilers, and software like the Linux kernel depend on them. The bugs related to violating the `-fno-strict-aliasing` flag involve reordering of memory accesses; such reorderings are very subtle but result

in equivalence failures. It would be very challenging to discover such bugs through automated testing. None of these issues appeared in CompCert-compiled executables, in our limited experiments. This is due to less-aggressive optimizations of CompCert.

2.5.3 Superoptimization experiments

Finally, we have used our tool inside a 32-bit x86 brute-force superoptimizer that supports a rudimentary form of loops: it allows enumeration of straight-line instruction sequences potentially containing the x86 string instructions `scas`, `stos`, and `cmps` (the equivalent of `memchr`, `memset`, and `memcmp` functions, resp.); each of these instructions is modeled as a TFG containing a cycle. Through supporting these instructions, optimized implementations for common routines like initializing an array, and comparing elements of two arrays, get synthesized automatically, that are up to 12x faster than compiled code generated by any of the four compilers we discussed (across O2 and O3).

Our 32-bit x86 superoptimizer is based on brute-force enumeration, and uses a two-step equivalence test: a fast probabilistic execution test, followed by a precise boolean test using our equivalence checker. It supports 400+ x86 opcodes (including a large part of MMX/SSE/AVX/AVX2 opcodes), supports memory operations, opcodes with loops, and also uses symbolic constants (e.g., `C0`, `C1`, etc.) and symbolic constant modifiers (e.g., `C0+1`, `C0-4`, etc.), to generalize concrete constant values. We use a static cost function based on measurements, and prior literature on instruction speeds [17]. The optimizer consists of three major components: a harvester, an enumerator, and a rewriter. The harvester harvests instruction sequences from the target function — sequences are harvested by using a sliding window (of different sizes) over the target function. The sequences are allowed to contain arbitrary branching, and loops. The harvester additionally records the set of live-registers at the end of each harvested sequence. The harvested sequences are grouped based on their input/output characteristics, and each group is optimized in one enumeration. Enumeration for multiple groups can happen in parallel over multiple

Pseudo-code/Description	Superopt-solution	*X*	gcc		llvm		icc		superopt	speedup
			O2	O3	O2	O3	O2	O3		
array_cmp*X*										

`for(i=0; i<n; i++)`
`  if a[i] != b[i]`
`  return 0`
`return 1`
	`cld`
	`mov $0,%eax`
	`cmp %ecx,%ecx`
	`repe cmps`*X*
	`sete %al`
		b
		w
		l
			4.34
			3.68
			3.67
				4.32
				3.67
				3.67
					3.26
					2.77
					3.17
						2.53
						2.46
						2.46
							4.01
							3.16
							3.67
								4.01
								3.16
								3.67
									13.5
									7.55
									3.83
										3.11
										2.05
										1.04

array_set*X*										
`for(i=0; i<n; i++)` `  a[i] = C0`*X*	`cld` `rep stos`*X*	b w l	4.72 4.73 4.71	4.72 4.72 4.71	5.46 4.16 4.15	5.67 4.16 4.17	5.66 4.73 5.01	5.36 6.30 5.65	**73.4** **35.9** **17.5**	12.9 5.70 3.10
num_onebits										

`for(i=0; i<32; i++)`
`  if (a & (1 << i))`
`  ret++`
	`popcnt a,%ecx`
	`add %ecx,%eax`
			2.34	2.34	15.7	15.7	17.3	18.5	**104.3**	5.64

array_find*X*										

`for(i=0; i<n; i++)`
`  if a[i] == v`
`  return i`
`return n`
	`cld`
	`mov %ecx,%esi`
	`cmpl $1,%ecx`
	`repne scas`*X*
	`sete %bl`
	`movzbl %bl,%ebx`
	`sub %ebx,%esi`
	`sub %ecx,%esi`
		b
		w
		l
			4.73
			3.30
			3.26
				4.73
				4.97
				4.96
					4.73
					3.63
					4.04
						4.73
						3.72
						4.96
							5.68
							3.67
							4.95
								5.67
								3.75
								4.95
									3.55
									3.72
									3.72
										0.63
										0.75
										0.75

array_rfind*X*										

`for(i=n-1;i>=0;i--)`
`  if a[i] == v`
`  return i`
`return -1`
	`std`
	`leal -1(%edi,`
	`  %ecx,1),%edi`
	`repne scas`*X*
	`setne %bl`
	`movzbl %bl,%ebx`
	`subl %ebx,%ecx`
		b
		w
		l
			3.66
			3.70
			3.61
				3.65
				3.70
				3.60
					6.00
					6.00
					6.00
						6.00
						6.00
						6.00
							6.00
							6.00
							6.00
								6.00
								6.00
								6.00
									3.00
									3.00
									3.00
										0.5
										0.5
										0.5

strlen										

string length	`cld`
	`mov $-1,%ecx`
	`mov $0,%eax`
	`repne scasb`
	`not %ecx`
	`dec %ecx`
			5.69	5.69	**5.99**	5.99	5.70	5.65	3.66	0.61

strrchr										

Index of last occurrence of char in C string, -1 ow	`cld`
	`mov $-1,%ecx`
	`mov $0,%eax`
	`repne scasb`
	`negl %ecx`
	`std`
	`mov %bl,%al`
	`repne scasb`
	`movzbl %al,%eax`
	`sub %eax,%ecx`
			4.46	4.38	3.94	3.96	**5.79**	5.71	3.31	0.57

Table 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.

cores/machines.

Enumeration involves executing the enumerated sequences over testvectors, for the probabilistic execution test. The testvectors are auto-generated from the target sequences to ensure sufficient path coverage within each target sequence. The testvectors terminate on the target sequences within a small number of iterations, and are thus expected to terminate in the solution. For memory operations, we use a 256-byte array, and sandbox all memory accesses in an enumerated sequence to fall within the array, similar to previous work [2]. The execution test involves 76 testvectors, and results in a throughput of around 120,000 sequences tested per second. It is rare for an inequivalent pair of sequences to pass the probabilistic test. Further, we use several heuristics to prune the enumeration, e.g., (a) prune sequences that read from a location which is not read by a target sequence, (b) prune sequences that write to a location that is not written by a target sequence, (c) prune sequences that are known to be equivalent to another lower-cost sequence, etc. The *learned* optimizations are stored in a database, indexed by the target sequences.

Finally, we rewrite the function by sliding a peephole window, and querying the database for an optimization, for each window. We select the right optimizations to stitch together, to form the lowest-cost function implementation, through a dynamic programming formulation of this problem, similar to [3].

Table 2.5 shows the synthesized sequences for some common string/array operations. The synthesized sequences are 1.04-12x faster than the fastest code generated by any of the four compilers. Notice that the solutions are not easy to verify manually, while our tool generates equivalence proofs for them within a few seconds. In general, we expect the support for loops to enable general-purpose loop-based optimizations in a superoptimizer, and this work is an initial step towards this goal.

2.6 Related work

One of the earliest examples of a translation validator can be found in a paper by Samet [46]. Translation validation for mature compilers on large and complex programs, has been reported in at least two previous works: Translation validation infrastructure (TVI) [39] for GCC, and Value-graph translation validation [55, 50] for LLVM.

TVI demonstrated the validation of the gcc-2.91 compiler and the Linux-2.2 kernel, across five IR passes in GCC, namely branch optimization, common-subexpression elimination (CSE), loop unrolling and inversion, register allocation, and instruction scheduling. In TVI, validation is performed in a pass-by-pass manner across each IR pass, i.e., first the input IR is validated against the output of the first pass, then the output of the first pass is validated against the output of the second pass, and so on. The TVI paper reports around 87% validation success rates. Necula's algorithm does not support loop unrolling, and that was reported as the primary cause for validation failures. There are several issues with TVI when applied to end-to-end (black-box and composed transformations) equivalence checking. First, this pass-based approach is not possible in synthesis/superoptimization setting. Second, TVI's heuristics for branch and memory-access correlations at basic-block granularity are syntactic, and fail for a large number of compiler transformations. Third, TVI cannot tolerate generating incorrect invariants. TVI has to always generate correct invariants, as it does not have any elimination mechanism. TVI generates invariants at a given node by using the weakest-preconditions of the invariants at the successor nodes. This procedure of inferring the simulation relation is both expensive and less robust than our guessing procedure. For end-to-end checks, the substituted expressions generated by weakest-precondition become large and unwieldy, resulting in SMT solver timeouts. Further, guessing based on only weakest preconditions is often inadequate. Finally, TVI was tested across five compiler passes, and did not address several transformations, including those relying on undefined behaviour.

Value-graph translation validation for LLVM has been performed previously in two

independent efforts [55, 50]. The value-graph based technique works by adding all known equality-preserving transformations for a program, to a *value graph*, until it saturates. Equivalence checking now involves checking if the graphs are isomorphic. In the work by Tristan et. al. [55], validation is performed across a known set of transformations, namely, dead-code elimination, global value numbering, sparse-condition constant propagation, loop-invariant code motion, loop deletion, loop unswitching, and dead-store elimination. Stepp et. al. [50] support all these transformations, and additionally enable partial-redundancy elimination, constant propagation, and basic block placement. While these tools capture several important transformations, they also omit many, e.g., loop inversion and unrolling, branch optimization, and instruction scheduling, to name a few. Some of these omitted transformations (e.g., loop inversion) enable more aggressive transformations, and so by omitting one of those, a chain of important transformation passes gets omitted. Also, none of these transformations rely on language-level undefined behaviour. For example, the transformations do not include the ones that could reorder accesses to global variables (e.g., by register-allocating them). Both papers report roughly 60-90% success rates for LLVM IR across the transformations they support. Compared head-to-head, this is comparable to our success rates, albeit in a much simpler setting. A value-graph approach is limited by the vocabulary of transformations that are supported by the translation validator, and thus seems less general than constraint-based approaches like TVI and ours. Also, the number of possible translations for passes like register allocation and instruction scheduling is likely to grow exponentially in a value-graph approach. At least with the current evidence, it seems unlikely that the value-graph based translation validation approach would yield good results for black-box equivalence checking.

Data-driven equivalence checking (DDEC) [48] is an effort perhaps closest to our goals of checking equivalence on x86 assembly programs. However, DDEC takes a radically different approach of relying on the availability of execution traces for high-coverage tests, an assumption that is not always practical in a general compiler optimization setting. DDEC was tested on a smaller set of examples (around 18) of x86 assembly code gener-

ated using GCC and CompCert, and all DDEC test examples are a part of our `ctests` benchmark. Compared head-to-head with DDEC, our algorithm is static (does not rely on execution traces), supports a richer set of constructs (stack/memory/global accesses, function calls, undefined behaviour), is more robust (tested on a much larger set of programs, and across a richer set of transformations), and more efficient (when compared head-to-head on the same programs). While DDEC can infer linear equalities through execution traces, it cannot handle several other types of non-linear invariants (e.g., inequalities) often required to prove equivalence across modern compiler transformations. Recent work on loop superoptimization for Google Native Client [7] extends DDEC by supporting inequality-based invariants; the evaluation however is limited to a small selection of test cases, and hence does not address several scalability and modeling issues that we tackle in our equivalence checker. For example, the authors do not model undefined behaviour, which we find is critical for black-box equivalence checking across real programs.

The *Correlate* module of parameterized program equivalence checking (PEC) [24] computes simulation based equivalence for optimization patterns represented as parameterized programs containing *meta-variables*. In contrast, we are interested in equivalence checking across black-box transformations involving low level syntax, as is typical in synthesis and superoptimization settings: our correlation algorithm with guessing procedures has been evaluated for this use case. In PEC's setting, the presence of meta-variables usually provides an easier correspondence between the two programs, greatly simplifying the correlation procedure; the relations (predicates relating variables in two programs) across meta-variables are also easier to determine in this setting.

Previous work on regression verification [51, 13] determines equivalence across structurally similar programs, i.e., programs that are closely related, with similar control structure and only a small (programmer introduced) delta between the two programs. In our setting, the programs being compared are significantly different because of transformations due to multiple composed compiler optimizations. While our equivalence checker can correctly compute equivalence across all the examples presented in regression verifi-

cation [51, 13], the converse is not true.

There are more approaches to translation validation and equivalence checking (e.g., [62, 63, 43, 59, 34, 22, 35, 4]), and most have been evaluated on a variety of relatively smaller examples. To our knowledge, previous work has not dealt with compiler transformations in as much generality as our work. Our work also overlaps with previous work on verified compilation [32, 60, 61], compiler testing tools [58, 27, 28], and domain specific languages for coding and verifying compiler optimizations [30, 31].

In terms of the correlation algorithm, our approach is perhaps closest to CoVaC [59], in that we both construct the JTFG incrementally, and rely on an invariant generation procedure, while determining the correlations. There are important differences however. CoVaC relies on an oracular procedure called *InvGen*; we show a concrete implementation of `PredicatesGuessAndCheck()`. Further, we differ significantly in our method to identify the correlations. CoVaC relies on correlating *types* of operations (e.g., memory reads and writes are different types), which is similar to TVI's syntactic memory correlations, and is less general than our semantic treatment of memory. Also, CoVaC relies on the *satisfiability* of the conjunction of edge conditions (viz. *branch alignment*) in the two TFGs, which is unlikely to work across several common transformations that alter the branch structure. CoVaC was tested on smaller examples across a handful of transformations. In contrast, our correlation method based on *equality* of condition of composite edges is more general, and we demonstrate this through experiments. Further, backtracking and careful engineering of guessing heuristics are important novel features of our procedure.

Most previous translation validation work (except DDEC) has been applied to IR. There has also been significant prior work on assembly level verification, through equivalence checking. SymDiff [25, 19, 26] is an effort towards verifying compilers and regression verification, and works on assembly code. However, the support for loops in SymDiff is quite limited — they handle loops by unrolling them twice. Thus, while SymDiff is good for checking *partial equivalence* [25], and to catch errors across program versions

and translations, generation of sound equivalence proofs for programs with loops is not supported.

Modeling of undefined behaviour (UB) for verification has previously been studied in Alive [36], where *acyclic* peephole optimization patterns of the `InstCombine` pass in LLVM are verified. These optimizations could potentially involve UB assumptions, and hence modeling of UB becomes necessary. The typical verification target for Alive is a few lines of optimization pattern representing a single optimization. In contrast, our verification targets involve concrete programs (with up to 1000s of lines) and containing *multiple* composed compiler optimizations. Alive models UB involving undefined values, poison values and instruction attributes like `nsw` (signed integer overflow), the kind that can be modeled through a simple syntactic analysis of the LLVM peephole optimization pattern. For example, the presence of UB attributes like `nsw`, `undef`, etc., in the optimization pattern directly indicates the UB assumptions. Aliasing based UB, involving out-of-bounds variable access assumptions (OBVA), requires an alias analysis, and Alive did not consider this in their work. Our work is directed towards studying the common transformations in end-to-end compiler optimization, and we find that UB involving OBVA is the most commonly exploited for optimization in both GCC and LLVM. We believe that our alias analysis can also benefit Alive interested in capturing aliasing based UB assumptions. Another major difference between Alive and our work is that Alive verifies acyclic optimization patterns, while we generalize the ideas to simulation-based equivalence across programs containing loops.

Another setting of translation validation is when modifications are allowed in the source code of a compiler. Work on credible compilation [45, 38, 23] augments the compiler passes to generate the proof of equivalence automatically. The checking is then straightforward, it just checks if the generated proof can establish equivalence between the programs. This simplification comes from the liberty of modifying the source of a compiler. Since a compiler pass knows the semantics of the transformation performed, it can generate the exact proof of equivalence during the compilation, eliminating the need

of a more difficult inference procedure. Although interesting, this flexibility of modifying a compiler is further away from our motivation of doing a black-box equivalence checking. In our setting of black-box equivalence checking, we cannot modify or even observe the source code of a compiler.

Our work overlaps with previous work on detection of *unstable* code, STACK [57]. STACK classifies unstable code as the code whose semantics are sensitive to undefined behaviour (UB). The underlying assumption of this work is that if an optimizer discards/modifies the (unstable) code due to the presence of UB, the resulting logic may behave differently from what the programmer intended. While STACK identifies certain important types of unstable code through static pattern-matching on LLVM IR, it also leaves out many. Aliasing based UB stands out as an example of UB not considered by STACK.

Our linearly-related (*lr*) and may-depend-on (*dep*) analyses resemble previous work on alias analysis for executable code by Debray et. al. [11]. The authors of this work noted that alias analysis for executable code requires reasoning about pointer arithmetic, and hence proposed special modeling for the `add` and `mult` opcodes, as these were the most commonly encountered opcodes for pointer manipulation on the RISC architecture they considered. However, because their analysis is syntactic in nature, it introduces imprecisions in common situations involving store and subsequent load of a pointer to/from memory. In such situations where a syntactic analysis does not provide enough information, the alias information would be conservatively widened to $\perp$ in their approach. Their empirical evaluations reflect these imprecisions. Our approach works on de-sugared expressions obtained from machine opcodes, involving standard bitvector and boolean operators. Also, our memory model allows reasoning about stores followed by loads to identical locations (without other intervening conflicting stores), thus capturing the common pattern of pointers getting saved to stack slots for future reference. This semantic treatment lends robustness to our analysis, and makes it independent of the underlying machine ISA. In another related work on alias analysis, Fernandez and Espasa [14] at-

tempted to remove the imprecisions discussed in [11], by sacrificing soundness guarantees. Sacrificing soundness is not acceptable in our setting. The authors of both these previous works on alias analysis for executable code were interested in link-time optimizations; unlike us, they do not describe a model for reasoning about UB using this obtained aliasing information.

Chapter 3

Equivalence checking across power environments

Energy harvesting devices that harvest energy from their surroundings, such as sunlight or RF radio signals, are increasingly getting popular. Because the size reduction of batteries has not kept pace with the size reduction of transistor technology, energy harvesting allows such devices to be much smaller in size, e.g., insect-scale wildlife tracking devices [44] and implantable medical devices [40]. Such devices are already commonplace for small dedicated computations, e.g., challenge-response in passive RFID cards, and are now being imagined for more general-purpose computational tasks [44, 37].

The harvested energy is unpredictable and usually not adequate for continuous operation of a device. Power failures are spontaneous and may occur after every 100 milliseconds, for example [44]. Thus, computation needs to be split into small chunks that can finish in these small intervals of operation, and intermediate results need to be saved to a persistent memory device at the end of each interval. A power reboot should then be able to resume using the results of the last saved computational state. This model of computation has also been termed, *intermittent computation* [37]. Typically, the intermittent programs involve instrumentation of the continuous programs (that are supposed to be continuously powered) with periodic *checkpoints*. The checkpoints need to be close

enough so that the computation across two checkpoints can finish within one power cycle. On the other hand, frequent checkpoints degrade efficiency during continuous operation. Further, a checkpoint need not save all program state, but can save only the *necessary* program state elements, required for an acceptable computational state at reboot. The presence of volatile and non-volatile program state simultaneously makes the problem more interesting.

An intermittent program may be written by hand, through manual reasoning. Alternatively, semi-automatic [37] and automatic [44, 56] tools can be used to instrument continuous programs with checkpoints, to allow them to execute correctly in the intermittent environments. The goal of these automated tools is to generate an intermittent program that is equivalent to the continuous program under all possible power failures. In addition to correctness, these tools try to generate intermittent programs with smaller checkpoints for efficiency. These tools reason over high-level programs (C or LLVM IR). Given that the failures happen at the architecture instruction granularity (and possibly at micro-instruction granularity) and it is the machine state that needs to be checkpointed; the reasoning at a higher level is error-prone and could go wrong because of the transformations (e.g., instruction reordering) performed by the compiler. Moreover, the bugs in intermittent programs could be very hard to detect: because the power failures are spontaneous and recurring, the number of potential states involved is very large.

Verifying the correctness of an intermittent program with respect to a continuous program is important from two aspects: First, we will be able to verify the correctness of the output of existing automatic instrumentation tools. Second, a verification tool will enable us to model automatic-instrumentation as a synthesis problem to optimize for the efficiency of generated intermittent programs, with the added confidence of verified output.

We present an automatic technique to verify the correctness of an intermittent program with respect to a continuous program. Towards this goal, we make the following contributions:

- A formal model of *intermittence* that correctly and exhaustively captures the behaviour of intermittent programs for all possible power failures. Our model of intermittent programs is amenable to checking equivalence with its continuous counterpart.

- Due to recurring executions in an intermittent program, an intermediate observable event (not occurring at exit) may occur multiple times, causing an equivalence failure. We show that if the observables are *idempotent* and *commutative*, then we can claim equivalence between the two programs.

- A robust algorithm to infer a provable bisimulation relation to establish equivalence across a continuous and an intermittent program. The problem is undecidable in general. The algorithm is robust in the sense of its generality in handling even minimal checkpointing states, i.e, a more robust algorithm can verify an intermittent program with smaller checkpoints. Stated differently, we perform *translation validation* of the translation from a continuous to an intermittent program. However, in our case, in addition to program transformation, the program execution environment also changes. The continuous program is supplied with continuous power, whereas the intermittent program is powered by a transient power supply.

3.1 Example

We briefly discuss, with the help of an example, the working of intermittent programs and issues associated with it. Figure 3.1a shows an x86 program that increments a *non-volatile* global variable `nv` and returns `0` on success. The program terminates after returning from this procedure. We call it a continuous program as it is not meant to work in an environment with power failures. Figure 3.1b shows an intermittent program, generated by instrumenting the continuous program. This program can tolerate power failures, and it is equivalent to the continuous program, under all possible power failures. The equivalence

is computed with respect to the observable behaviour, which in this case is the output, i.e., the value of return register `eax` and the value of the global variable `nv`.

The intermittent program has been generated from the continuous program by inserting checkpointing logic at the checkpoint locations `CP1` and `CP2`. During checkpointing, the specified `CPelems` and the location of the current executing checkpoint get saved to `CPdata` in persistent memory. In case of a power failure, the program runs from the entry again, i.e., the restoration logic, it restores the `CPelems`, and then jumps to the location stored in `CPdata.eip` (term `eip` comes from the register `eip`, which holds the program counter). For the first run of the intermittent program, the checkpoint data is initialized to `((), Entry)`, i.e., `CPdata.CPelems=() and CPdata.eip=Entry`. This ensures that on the first run, the restoration logic takes the program control flow to the original entry of the program.

In case of power failures, the periodic checkpointing allows the intermittent programs to not lose the computation and instead, start from the last executed checkpoint. For example, if a failure occurs at location `I5`, the intermittent program will resume its computation correctly from `CP2`, on power reboot. This is so because the checkpoint `CP2` gets executed while coming to `I5`, and the restoration logic, on the next run, restores the saved state and jumps to `CP2`. Moreover, under all possible scenarios of power failures, the output of the intermittent program remains equal to that of the continuous program.

Notice that we need not checkpoint the whole state of the machine, and only a small number of checkpoint elements is sufficient to ensure the equivalence with the continuous program. A smaller checkpoint is important as it directly impacts the performance of the intermittent program; a smaller checkpoint results in less time spent on saving and restoring it. Figure 3.1a shows the smallest set of $CPelem$s that need to be saved at `CP1` and `CP2`. The first two elements of `CPelems1` and the only two elements of `CPelems2` ensure that the address where return-address is stored and the contents at this address, i.e., the return-address (both of which are used by the `ret` instruction to go back to the call site) are saved by the checkpoint. As per the semantics of `ret` instruction, `ret` jumps to

```
Entry:
CP1: I1: push ebp
     I2: mov esp ebp
     I3: inc (nv)
CP2: I4: xor eax eax
     I5: pop ebp
     I6: ret

CP1: I1
CP2: I4
CPelems1:
  esp, (esp), nv
CPelems2:
  esp, (esp+4)
```

(a) Continuous program

```
Restoration: # new entry
      restore CPdata.CPelems CPelems
      jmp CPdata.eip  # init to Entry:
Entry:  # original entry
CP1': # checkpointing logic
      save (CPelems1, CP1) CPdata
CP1:  I1: push ebp
      I2: mov esp ebp
      I3: inc (nv)
CP2': # checkpointing logic
      save (CPelems2, CP2) CPdata
CP2:  I4: xor eax eax
      I5: pop ebp
      I6: ret
```

(b) Intermittent program

Figure 3.1: The first assembly program increments a global non-volatile variable `nv` 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.

the address stored at the address `esp` (`esp` is the stack register), i.e., it jumps to `(esp)`[1]. At `CP1` and `CP2`, the return address is computed as `(esp)` and `(esp+4)` respectively. Note that the expressions are different because of an intervening `push` instruction. Further, checkpointing of non-volatile data is usually not required; however, `(nv)` needs to be saved at `CP1` because it is being read and then written before the next checkpoint. If we do not save `(nv)` at `CP1`, failures immediately after `I3` would keep incrementing it.

Tools that generate intermittent programs by automatically instrumenting the given continuous programs [37, 44, 56] usually work at a higher level (C or LLVM IR). These tools perform live variable analysis for volatile state and write-after-read (WAR) analysis for non-volatile state to determine the checkpoint elements. However, these approaches result in making conservative assumptions because of the lack of knowledge of compiler transformations (e.g., unavailability of mapping between machine registers and program variables) and the proposed checkpointed elements contain unnecessary elements. For example, a tool, like DINO [37], without the knowledge of compiler transformations would

[1] `(addr)` represents 4 bytes of data in memory at address `addr`.

checkpoint all the registers and all the data on the stack for our running example. Even if these analyses are ported at the machine level, the proposed checkpoint elements would be conservative as these analyses are syntactic in nature. For example, a live variable analysis for the example program would additionally propose the following unnecessary elements: `ebp` at `CP1` and `eax, (esp)` at `CP2`.

The observable of the example program is produced only at the exit. Let us consider a case, when the observable events are produced before reaching the exit (called intermediate observables). In case of intermediate observables, the observables may get produced multiple times due to the power failures. For example, assume that there is an atomic instruction `I5': print("Hello, World!")` (which produces an observable event) in between `I4` and `I5`. Due to the power failures at `I5` and `I6`, the program will again execute the code at `I5'` and the observable event will get produced again, resulting in an equivalence failure. Interestingly however, it is possible that the observer cannot distinguish, whether the observable has been repeated or not. This depends upon the semantics of `print`, e.g., if it prints to the next blank location on the console, then the observer may see multiple "Hello, World!" on the console. However, if it prints at a fixed location (e.g., fixed line and column of an LED board), then the multiple calls to `print` would just overwrite the first "Hello, World!", and this would be indistinguishable to the observer.

The non-determinism in the intermittent program and consequently, the repeated observables makes the problem of checking equivalence with respect to continuous program interesting. We take the ideas of black-box equivalence checking algorithm from Chapter 2, extend them to handle non-determinism and repeated observables, and come up with an algorithm that can establish equivalence across power environments, i.e., between the continuous and the intermittent programs.

Rest of the discussion is organized as: Section 3.2 presents the modifications needed in the TFG grammar to support checkpointing, modeling of intermittent program behaviour is discussed in Section 3.3, and finally, Section 3.4 describes the algorithm to establish equivalence between the continuous and the intermittent program.

$\mathbb{T}$	::=	$(\mathbb{G}([node],[edge]))$
$node$	::=	$(pc(int) \mid exit(int), \textbf{\textit{[CPelem]}})$
$edge$	::=	$(node, node, edgecond, \tau)$
$edgecond$	::=	$state \rightarrow \varepsilon$
τ	::=	$state \rightarrow state$
$state$	::=	$[(string, \textbf{\textit{type}}, \varepsilon)]$
ε	::=	$const(string) \mid nry\_op([\varepsilon]) \mid select(\varepsilon, \varepsilon, int) \mid store(\varepsilon, \varepsilon, int, \varepsilon)$
$\boldsymbol{CPelem}$	**::=**	$\boldsymbol{(string)} \mid \boldsymbol{(string, \varepsilon, int)}$
$\boldsymbol{type}$	**::=**	**`Volatile`** | **`NonVolatile`**

Figure 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.

3.2 Program representation

We modify the grammar of TFG in Figure 2.6 to support the constructs of intermittent programs. Figure 3.2 shows the modified grammar. For intermittent execution, checkpoints can be inserted at arbitrary program locations. A checkpoint saves the required state elements to a persistent store. The saved state would allow the restoration logic to resume from the last executed checkpoint. We model checkpoints by annotating the TFG nodes corresponding to the checkpoint locations as *checkpoint nodes* with their corresponding checkpointed state (specified as a list of checkpoint elements $[CPelem]$). The semantics of $CPelem$s are such that on reaching a node with $CPelem$s, the projections of $CPelem$s on the state are saved. A $CPelem$ can either specify a named register (first field) or it can specify an address with the number of bytes of a named memory (second field). The first type of $CPelem$ allows to checkpoint a register or the complete memory state, whereas the second type allows flexibility to checkpoint a memory partially or in ranges.

Figure 3.3 shows the TFGs of the continuous and the intermittent programs of Figure 3.1. Note that the TFGs also show the other instrumentation details, namely failure edges and restore edges, which we discuss next.

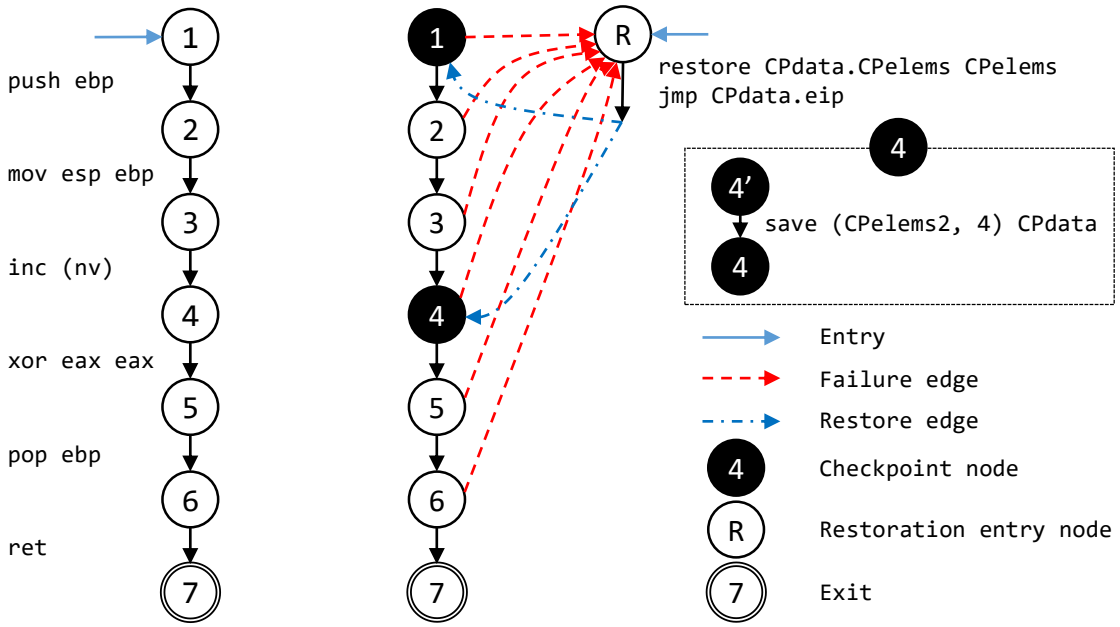


Figure 3.3: TFGs of the continuous and the intermittent program of Figure 3.1.

3.3 Modeling intermittence

3.3.1 Instrumentation model

Instrumenting a continuous program to generate an intermittent program involves: adding the *checkpointing logic* at the given checkpoint nodes, adding the *restoration logic*, changing the entry of the program to the restoration logic, and setting the initial checkpoint data in the persistent memory.

The checkpointing and the restoration logic work with data called checkpoint data ($CPdata$). The checkpoint data is read/written from/to a fixed location in a persistent memory. The checkpoint data consists of $CPelems$ of the machine state and the checkpoint location. The checkpointing logic saves the checkpoint data from the machine state, and the restoration logic updates the machine state from the checkpoint data. Additionally, after updating the machine state, the restoration logic changes the program control flow (jmp) to the stored checkpoint location ($CPdata.eip$). The checkpointing logic is added for all the given checkpoint nodes. The restoration logic, however, is added once, and the entry of the program is changed from the original entry to the entry of the restora-

tion logic. The checkpoint data is initialized with the empty $CPelem$ list and the stored checkpoint location is set to the original entry. This ensures that the intermittent program starts from the correct entry, i.e., the original entry, in its very first execution. Further, it is assumed that the location where $CPdata$ is stored cannot alias with the addresses of the programs. In other words, the program, except for checkpointing and restoration logic, should not read or write $CPdata$.

The checkpointing logic is made atomic by using a double-buffer to save the checkpoint data. The checkpointing logic works with two checkpoint data: current $CPdata$ and unused $CPdata$, and a pointer $CPdataLocation$ points to the current $CPdata$. While checkpointing, it writes to the unused checkpoint data and once complete, it updates $CPdataLocation$ to the address of unused checkpoint data, making it the current $CPdata$. This technique ensures that a failure while executing the checkpointing logic does not corrupt the checkpoint data. For brevity, we do not show the implementation of double buffering.

Figure 3.3 shows the TFGs of the continuous and the intermittent program. Nodes 1 and 7 are the entry and the exit locations of the continuous program respectively. In the intermittent program, the checkpointing logic has been inserted at nodes 1 and 4, and the restoration logic has been appropriately added at program entry. The $CPelem$s at node 1 (`CPelems1`) and 4 (`CPelems2`) are listed in Figure 3.1a. A checkpoint node in the intermittent program is shown as a single node in the program graphs; in practice, it consists of multiple nodes and edges representing the TFG of the checkpointing logic. Figure 3.3 also shows the TFG of the checkpointing logic of node 4. It saves the `CPelems2` and sets the stored program location ($CPdata.eip$) to the location of the checkpoint node 4 in this example. The intermittent program always starts in the restoration logic. It restores the state from the saved $CPdata.CPelem$s and then jumps to the stored program location ($CPdata.eip$).

3.3.2 Modeling power failures

Power failures in an intermittent environment are spontaneous and can occur at any moment. We assume that a power failure can occur before and after every instruction of the assembly program, which is analogous to the properties of *precise-exceptions*, and is guaranteed by most architectures. On architectures where this assumption cannot be made, one can model power failures at the micro-instruction level, i.e., before and after every micro-instruction of that architecture, and rest of the technique would remain the same.

At the TFG level, nodes precisely represent the instruction boundaries, i.e., a power failure can occur at *any* of the nodes of the TFG. On a power failure: the volatile data is lost and the program, on reboot, starts from the entry, i.e, the restoration logic. We model power failures at each node by adding a non-deterministic *failure edge* from each node of the TFG to the entry of the restoration logic.

Definition 3.3.1 (Failure edge) *A failure edge is an edge of a TFG from node n to the entry node R of the restoration logic. The $edgecond$ and the transfer function τ of a failure edge are defined as:*

$$edgecond = \delta$$

$$\tau(S) = \forall_{(s,t,\varepsilon)\in S} \begin{cases} (s, t, NDV_\varepsilon) & \text{if } t \text{ is } \texttt{Volatile} \\ (s, t, \varepsilon) & \text{if } t \text{ is } \texttt{NonVolatile} \end{cases}$$

Where δ is a non-deterministic boolean value, S is the state at the node n, (s, t, ε) represents an element of the state S, and NDV_ε is a non-deterministic value of the type of the expression ε.

A failure edge of a TFG models the non-determinism and the effect of a power failure; the condition under which the edge is taken is non-deterministic, i.e., spontaneous power failure, and the effect is modeled by the transfer function and the program control flow change. The transfer function of a failure edge preserves the non-volatile data and garbles

the volatile data (overwritten with arbitrary/non-deterministic values) and the failure edge goes to the entry, encoding the fact the program starts from the entry on reboot.

The failure edges are added for all the nodes of the instrumented TFG, even for the nodes of the checkpointing and the restoration logic. The failure edges for the nodes of checkpointing and restoration logic capture the fact that power failures are possible even while executing the checkpointing and the restoration logic. This failure model is exhaustive and complete, and it precisely models the semantics of power failures. The failure edges are shown as the dashed edges in Figure 3.3.

3.3.3 Resolving indirect branches of restoration logic

The restoration logic changes the control flow of the program based on the contents of stored program location. It is implemented as an indirect jump (i.e., `jmp CPdata.eip`) at the assembly level. In general, an indirect jump may point to any program location; however, in our case we can statically determine the set of locations the indirect jump can point to. As the indirect jump depends on the value stored in `CPdata.eip`, we determine all the values that may get stored in `CPdata.eip`.

At the beginning, `CPdata.eip` is initialized to the original entry of the intermittent program. And later, it is only modified by the checkpointing logic and set to the locations of the checkpoint nodes. Thus, the indirect jump can either point to the original entry or any of the checkpoint nodes. Using this information, we resolve the indirect jump of the restoration logic and add *restore edge*s to the intermittent TFG to reflect the same.

Definition 3.3.2 (Restore edge) *A restore edge is an edge of a TFG from the node R, i.e., the restoration logic, to the original entry or a checkpoint node n of the TFG. The edgecond and the transfer function τ of the restore edge are defined as:*

$$edgecond = (CPdata.eip == n)$$

$$\tau(S) = \forall_{(s,t,\varepsilon)\in S} \begin{cases} (s,t,\varepsilon) & (s) \notin CPdata.CPelems \\ (s,t,\varepsilon) & (s,\_,\_) \notin CPdata.CPelems \\ (s,t,D) & ((s):D) \in CPdata.CPelems \\ (s,t,store(\varepsilon,a,b,D)) & ((s,a,b):D) \in CPdata.CPelems \end{cases}$$

Where S is the state at the node R, (s,t,ε) is an element of the state S, (s) and (s,a,b) are checkpoint elements, $CPdata.CPelems$ has the stored checkpoint elements as a map from $CPelems$ to the stored data (D), and s, t, ε, a and b correspond to name, type, expression, address and size (number of bytes) respectively.

The edge condition represents that the edge is taken to a checkpoint node n if the stored program location $CPdata.eip$ is equal to n. The transfer function restores the state by updating the state with all the $CPelems$ available in the $CPdata.CPelems$. The restore edges are added to the intermittent TFG from the restoration logic to all the checkpoint nodes and the original entry. The restore edges are shown as the dash-dot edges in Figure 3.3.

3.4 Equivalence

We find that the problem of checking equivalence across power environments, i.e., across the continuous and the intermittent programs, is unique in its own way, and we cannot just offload it to any existing equivalence checker. The important differences that make this problem unique are:

1. The intermittent program, which runs in an environment with power failures, has non-determinism, whereas the continuous program is deterministic. Previous techniques work in a setting where both the programs are deterministic; in our setting, one of the programs (the intermittent program) has edges that can be taken non-deterministically, i.e., the failure edges. Consequently, the correlation is different

as power failures would now be modeled as *internal* moves, and hence, we instead need to infer a *weak bisimulation relation* [47].

2. Due to recurring executions in the intermittent program (because of the power failures), an intermediate observable event in the intermittent program can be produced more times than in the continuous program. To reason about the same, we describe two properties of the observables, namely *idempotence* and *commutativity*, and we use these properties to establish equivalence under the repeated occurrences of the observables.

As we have seen in Figure 3.1, the amount of instrumentation code added to intermittent program is quite small and most of the code of the intermittent program remains the same. However, even in this setting, the problem of checking equivalence between the continuous and the intermittent program is undecidable in general. In other words, determining whether a certain checkpoint element ($CPelem$) needs to be checkpointed is undecidable. We define equivalence between a continuous and an intermittent program, i.e., across the instrumentation, and we prove the theorem that determining this equivalence is undecidable.

Definition 3.4.1 (Equivalence) *A continuous TFG (C) is equivalent to an intermittent TFG (I), where I has been generated by instrumenting C, if starting from identical input state S, the two TFGs produce equivalent observable behaviour, for all values of S.*

Theorem 3.4.2 *Given a continuous TFG (C) and an intermittent TFG (I), where I has been generated by instrumenting C, determining equivalence between C and I is undecidable.*

Proof 3.4.3 *Determining whether any function* `f` *halts can be reduced to this problem. Consider the following construction of a continuous (C) and an intermittent (I) program:* `C(a)={f(); print(a);}` `I(a)={CP(); f(); print(a);}`*, such that* `CP()` *checkpoints the complete state except the volatile variable* `a`*. The two functions can only*

be equivalent if `f()` *does not halt. Checking whether* `f` *halts can be written in terms of determining whether the two functions are equivalent:* `f_Halts` $= (C \neq I)$*. However, the halting problem is undecidable, hence, checking equivalence between a continuous and an intermittent program is also undecidable.*

3.4.1 Correlation

The correlation across two TFGs defines a matching between the nodes and the paths (also called moves) of the two TFGs. It tells the path taken by one program, if the other program takes a certain path, and vice versa. In our case, we reason in terms of the paths from one checkpoint to another checkpoint (*checkpoint-to-checkpoint path*s, defined next), and define the correlation in terms of the same.

Definition 3.4.4 (Checkpoint-to-checkpoint path) *Given a continuous TFG C and an intermittent TFG I, where I has been generated by instrumenting C: a path from node n to node m in the intermittent TFG I is a checkpoint-to-checkpoint path if the nodes n and m belong to the set $N = \{entry, exit\} \cup CPs$, and none of its intervening nodes between n and m belongs to N. Here $entry$, $exit$ and CPs are the original entry, the exit and the set of checkpoint nodes respectively.*

A checkpoint-to-checkpoint path in the continuous program C is defined in the same manner, however, assuming the checkpoint nodes of the corresponding intermittent TFG (i.e., I); this is because C has no notion of checkpoint nodes.

The checkpoint-to-checkpoint paths are further classified depending upon whether a power failure occurs or not, on a checkpoint-to-checkpoint path.

Definition 3.4.5 (Failure path) *A checkpoint-to-checkpoint path is a failure path if a power failure occurs in it.*

Theorem 3.4.6 *A failure path starts and terminates on the same checkpoint. In other words, a failure path starting and terminating on different checkpoint is not reachable.*

Proof 3.4.7 *Since there are no intervening checkpoints on a failure path, the stored checkpoint location ($CPdata.eip$) is the starting checkpoint (n), implying that on a failure, only one restore edge, which goes from the restoration logic to the starting checkpoint, will have its $edgecond\ true$.*

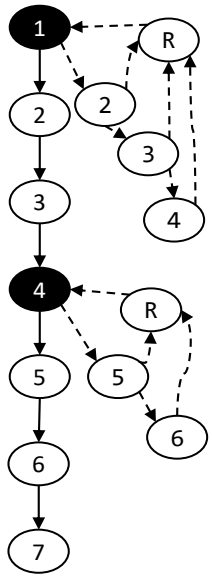
Definition 3.4.8 (Progress path) *A checkpoint-to-checkpoint path is a progress path if there are no power failures in it.*

A checkpoint-to-checkpoint path starting from a checkpoint can either reach a successive checkpoint if no power failure occurs in between, or it reaches back to the starting checkpoint (via a failure and then the restore edge to it) if there is a power failure. A checkpoint-to-checkpoint path in the intermittent TFG is either a failure path or a progress path. However, all the checkpoint-to-checkpoint paths in the continuous program are progress paths as there are no failures in it. Figure 3.4a shows the failure and the progress paths of the intermittent TFG. Note that we have not shown the edges of the TFG of checkpointing logic, we get rid of them by composing these edges with the incoming edges of the start node of a checkpoint, e.g., path $3 \rightarrow 4' \rightarrow 4$ is collapsed into an edge $3 \rightarrow 4$.

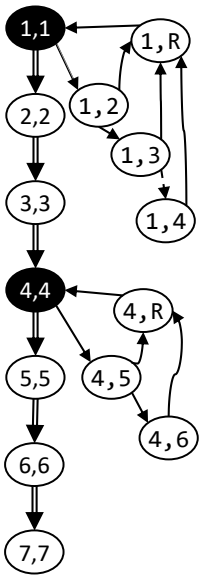
We use the notion of a *weak bisimulation relation* [47] to establish equivalence between the continuous and the intermittent TFGs. In a weak bisimulation relation, a move (of correlation) in the non-deterministic program may be preceded and succeeded by any number of internal moves. The non-deterministic failure paths of the intermittent TFG are modeled as the *internal* moves and progress paths of the two TFGs are treated as the usual moves. We propose the following correlation between the two TFGs:

Definition 3.4.9 (Correlation) *Given a continuous TFG C and an intermittent TFG I, where I has been generated by instrumenting C, both starting from the original entry or the same checkpoint node (n_{CP}):*

1. *If I takes a progress path p, then C takes the corresponding progress path p in it, and vice versa. Additionally, the individual edges of the progress paths are also*



(a) Simplified TFG



(b) Correlation graph

Node: (1,1)
$M^C_{nv} = M^I_{nv}$, $nv^C = nv^I$,
$esp^C = esp^I$,
$select(M^C, esp^C, 4) =$
$select(M^I, esp^I, 4)$
Node: (4,4)

$M^C_{nv} = M^I_{nv}$, $nv^C = nv^I$,
$esp^C = esp^I$,
$select(M^C, (esp^C + 4), 4)$
= $select(M^I, (esp^I + 4), 4)$

Node: (7,7)
$eax^C = eax^I$,
$M^C_{nv} = M^I_{nv}$

(c) Invariants

Figure 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.

taken together. That is, if C takes the edge $(n \to m) \in p$, then I takes the same edge $(n \to m)$, and vice versa. That is, for all nodes $n \in p$ and edges $(n \to m) \in p$: node n and edge $n \to m$ of C are correlated with node n and edge $n \to m$ of I, respectively.

2. *If I takes a failure path p, then C takes a* no-move*, i.e., C does not move at all and stays at the same node (n_{CP}), and vice versa. Further, every individual edge of the failure path of I is taken with a no-move of C. That is, for all nodes $n \in p$: node n of I is correlated with node n_{CP} of C.*

Intuitively, the above correlation of moves states that for TFGs starting from the entry or the same checkpoint: if there is no power failure, and the intermittent program moves to a successive checkpoint, then the continuous program also moves to the same next checkpoint, and vice versa. However, if the intermittent program undergoes a failure,

hence, returning to the starting checkpoint, then the continuous program does not move at all, and stays at the starting checkpoint, and vice versa. Note that the above correlation, in its current form, supports only the checkpointing transformation, which is suitable for our setting. However, it is possible to extend this definition to support other peephole optimizations along with the checkpointing transformation.

Correlation between two TFGs forms a graph, whose nodes and edges are pairs of nodes and edges of the two TFGs. That is, if (n_C, n_I) is a node and (e_C, e_I) is an edge of the correlation graph, then n_C and n_I are the nodes of the continuous and the intermittent TFG respectively, similarly, e_C and e_I are the edges of the continuous and the intermittent TFG respectively. Figure 3.4b shows the correlation graph for the running example.

3.4.2 Inferring invariants

Once we have fixed the correlation between the two TFGs, we need to check if the correlation is indeed correct as it is not necessary that the two TFGs take the same progress path starting from the same checkpoint. Furthermore, we need to check that the two TFGs produce the same observable behaviour. This involves inferring invariants at the nodes of the correlation graph. The inferred invariants should be strong enough to prove that the correlated edges are taken together (i.e., equivalent $edgecond$s of the correlated edges of the two TFGs) and the observables at the correlated edges are identical. Formally:

$$\underset{(n_C,n_I)\to(m_C,m_I)}{\forall} invariants_{(n_C,n_I)} \Rightarrow_{(n_C,n_I)\to(m_C,m_I)} (o_{(n_C\to m_C)} = o_{(n_I\to m_I)}) \wedge$$

$$(edgecond_{(n_C\to m_C)} = edgecond_{(n_I\to m_I)})$$

Here $(n_C, n_I) \to (m_C, m_I)$ is an edge in the correlation graph, $n_C \to m_C$ and $n_I \to m_I$ are edges in the continuous and the intermittent TFG respectively, $invariants_{(n_C,n_I)}$ represents the conjunction of the invariants at the correlation node (n_C, n_I), $edgecond_e$ represents the edge condition of an edge e, o_e represents the observable on an edge e,

and $\Rightarrow_{(n_C,n_I)\rightarrow(m_C,m_I)}$ represents the implication over the edge $(n_C, n_I) \rightarrow (m_C, m_I)$ as defined in Section 2.1.

For inferring these invariants we reuse the guess-and-check technique of Chapter 2. Please refer to Section 2.2.4 for the detailed discussion. Figure 3.4c shows the inferred invariants for some nodes, which can prove the required conditions and equivalence for the running example, under the $CPelems$ of Figure 3.1a.

On final notes, our equivalence checking algorithm is general and handles loops seamlessly; in fact, we are already handling the loops which get introduced due to the failure edges. Had there been a loop in the example program, say there is a backedge from node 3 to 2, it would reflect in the failure and progress paths too, e.g., Figure 3.4a will contain a solid as well as a dash edge from node 3 to 2. Similarly, the correlation graph too will have edges from node (3,3) to (2,2) and node (1,3) to (1,2). Finally, our technique is not without limitations – it is possible that a correlation other than the proposed one, or an invariant of different shape/template (other than the one used) is required for proving the equivalence. Though we did not encounter this in practice.

3.4.3 Intermediate observables

We now discuss the issue with observables occurring at the intermediate nodes, i.e., the nodes other than the exit node. We call these the intermediate observables. In an intermittent program, an intermediate observable event can be produced more times than is produced in the continuous program. It happens because of the recurring executions of an intermediate observable due to power failures. Given a sequence $\lambda^C = o_1 o_2 ... o_i ... o_x$ (written as a string) of observable events on a progress path (from checkpoint node n_1 to checkpoint node n_{x+1}) of the continuous TFG, the event o_i is produced on the edge $n_i \rightarrow n_{i+1}$, for $i \in [1, x+1)$. The possible sequences of observable events for the corresponding intermittent TFG, during the moves from checkpoint node n_1 to checkpoint

node n_{x+1} (by taking one or more failure paths followed by a progress path) are:

$$\lambda^I = \lambda^I_{n_1}\lambda^C \quad such\ that\ \ \lambda^I_{n_1} = (o_1|o_1o_2|o_1o_2o_3|...|o_1o_2...o_{x-1})^*$$

The sequence is written as a regular expression, where $*$ represents Kleene star, i.e., zero or more repetitions and $|$ represents the alternation operator. The first part of the expression $\lambda^I_{n_1}$ represents all sequences of observables produced at node n_1. The alternation operator encodes that a failure may happen at any node n_i and may produce a sequence $o_1o_2...o_{i-1}$ for $i \in [1, x]$ (a failure at n_{x+1} will not take the control back to n_1); the $*$ operator encodes that failures may occur zero or more times. The second part (λ^C) represents the case when there is no power failure and the execution reaches the successive checkpoint n_{x+1}.

The sequence of observables produced in the intermittent program could be different from that produced in the continuous TFG. However, if the effects of the two sequences, i.e., λ^C and λ^I are same, and the observer cannot differentiate between the two, we will be able to claim the equivalence of the two programs. To this end, we define a notion of *idempotence* and *commutativity* of observables, and we use these properties to prove that the sequences of observables produced by the continuous and the intermittent TFG are equivalent if the observables are idempotent and commutative.

Definition 3.4.10 (Idempotence) *An observable event o is idempotent if its recurring occurrences are undetectable to the observer. That is, the sequence oo produces the same effect as o.*

Definition 3.4.11 (Commutativity) *The observable events o_1 and o_2 are commutative if the order of occurrences of the two events is not important to the observer. That is, the sequences o_1o_2 and o_2o_1 are both equivalent to the observer.*

Intuitively, an observable is idempotent if the observer cannot detect if the observable occurred once or multiple times. For example, the observable `print(line, column,`

`text)`, which prints `text` at the given `line` and `column`, is idempotent. The user cannot distinguish if multiple calls to this function have been made. Observables `setpin(pin, voltage)` (sets the voltage of the given pin) and `sendpkt()` (send network packet) are more examples of idempotent observables. The observer cannot tell if the function `setpin()` is called multiple times, as it will not change the voltage of the pin on repeated executions. In case of `sendpkt()`, if the network communication is designed to tolerate the loss of packets, and consequently, the observer/receiver is programmed to discard the duplicate packets, then the observable is idempotent with respect to the receiver. Two observables are commutative if it does not matter to the observer, which one occurred first. For example, if a program lights an LED and sends a packet, and if these two events are independent to the observer, e.g., the packet is meant for some other process and the LED notification is meant for the user, then their order is unimportant to the observer.

Theorem 3.4.12 *$\lambda^I = \lambda^C$, if for all o_i and o_j in λ^C, o_i is idempotent, and o_i and o_j are commutative.*

Proof 3.4.13 *In sequence λ^I, we move an event o_i to position i (by applying commutativity) and if the same event is present at $i+1$, we remove it (by applying idempotence), we keep applying these steps until only one o_i remains. Performing these steps in increasing order of i, will transform λ^I into λ^C. If the length of λ^I is finite, termination is guaranteed.*

With all the pieces, we state the final theorem now:

Theorem 3.4.14 *A continuous TFG C and an intermittent TFG I, where I is generated by instrumenting C, are equivalent if:*

1. *Invariants can prove the correlation and the equivalence of observables at each correlated edge of the progress paths (Section 3.4.2).*

2. *On every progress path: each observable is idempotent, and every pair of observables is commutative (Section 3.4.3).*

3. Both the TFGs, i.e., C and I, terminate.

Proof 3.4.15 *Proof by induction on the structure of programs:*

Hypothesis: Both programs C and I produce the same observable behaviour on execution till a node n, for $n \in N = \{entry, exit\} \cup CPs$, where CPs is the set of checkpoint nodes.

Base: At entry, the two programs, C and I, have same observable behaviour.

Induction: Assuming the hypothesis at all the immediate predecessor checkpoints (m) of node (n), we prove that the observable behaviour of the two programs are equivalent at n, where $m, n \in N$.

An observable sequence at node n for program I can be written in terms of the observable sequence at the predecessor node m and the observable sequence produced during the moves from m to n: $\lambda^I_n = \lambda^I_m \lambda^I_{m \to n}$. From Condition#1, we can prove that the two programs move together from m to n and the individual observables of the two programs are same. Using the same along with Condition#2, Condition#3 and Theorem 3.4.12, we claim that $\lambda^I_{m \to n} = \lambda^C_{m \to n}$. Finally, using the hypothesis $\lambda^I_m = \lambda^C_m$, we prove that $\lambda^I_n = \lambda^C_n$.

Note that Condition#3 is important to eliminate the case when after some failure, the intermittent program never gets powered, and it never progresses. In that case, equivalence is undefined. Condition#3 eliminates the same.

3.5 Evaluation

We evaluate our technique in terms of the runtime of verification, and the robustness and capability of our algorithm. We are not aware of any previous verifier for this problem, and so we do not have a comparison point for the verification runtimes of our tool. However, we do compare the robustness and capability of our technique by using our verifier in a simple synthesis loop, whose goal is to minimize the size of checkpoints at a given set of checkpoint nodes. Moreover, the capability of this synthesis loop is dependent on

the capability of our verifier. If our verifier can prove the equivalence between the continuous and the intermittent programs, with smaller checkpoints, then the synthesis loop can generate an intermittent program with smaller checkpoints. This also enables us to compare our work with DINO [37]. With similar goals, DINO automatically generates an intermittent program from a given continuous program by instrumenting it at a given set of checkpoint locations. It works with mixed-volatility programs and performs a syntactic analysis to determine the checkpoint elements that need to be checkpointed. However, unlike our tool, DINO's output is unverified. A detailed comparison with DINO is available in Section 3.6.

We implemented our equivalence checking technique in a verifier for the x86 architecture. Our technique is independent of the architecture: the reason we chose the x86 architecture is primarily because we had access to a disassembler and semantic models of x86 ISA. Constructing a TFG from an executable required us to resolve other indirect jumps (other than that of the restoration logic) occurring in the program; in particular, the indirect jumps due to the function returns, i.e., the `ret` instructions. A `ret` instruction takes back the program control to the return-address stored in a designated location in the stack. The return-address is set by the caller using the `call` instruction. We perform light-weight static analysis to determine the call sites of every function and hence determine the return-addresses of every `ret` instruction. We appropriately add the *return edge*s (similar to restore edge) from the return instruction to the determined call sites. The transfer function of the return edge is $identity$ and its $edgecond = (return\_address == call\_site\_address)$.

While testing our verifier on some handwritten pairs of continuous and intermittent programs, we found that it is very easy for a human to make mistakes in suggesting the checkpoint elements and checkpoint locations, especially for mixed-volatility programs. For example, in the program in Figure 3.1, the user ought to specify a checkpoint before `I3`. If a checkpoint location is not specified before `I3`, the intermittent program cannot be made equivalent to the continuous program no matter what the checkpoint elements are. Our verifier gets used by the synthesis loop, and the average runtime of our verifica-

Benchmark	# CP nodes	Avg. CP size DINO	Avg. CP size synthesis loop	Improvement over DINO	Synthesis time (s)	Avg. verification runtime
DS	5	120.8	42.4	2.8x	3500	16.5
MIDI	4	80	19	4.2x	2154	11.9
AR	2	128	22	5.8x	26290	332.8
CRC	2	96	24	4x	42	1.1
Sense	3	96	25.3	3.8x	331	3.2

Table 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.

tion procedure ranges between 1s to 332s for benchmarks taken from previous work on intermittent computation [44, 37]. Table 3.1 describes our benchmarks and results, and the seventh column shows the individual average runtimes for different benchmarks. Almost all the verification time is spent on checking satisfiability of SMT queries. For these experiments, we discharge our satisfiability queries through the Yices SMT solver [12].

We implemented a synthesis loop to optimize the checkpoint size. Given a set of checkpoint locations, the synthesis loop tries to greedily minimize the checkpoint elements that need to be checkpointed. It keeps proposing smaller checkpoints (with fewer $CPelems$), and it relies on our verifier to know the equivalence between the continuous and the intermittent program, with the current checkpoint elements. The synthesis loop starts by initializing each checkpoint node with all possible checkpoint elements (the most conservative solution). It then iterates over each checkpoint element of all the checkpoint nodes, and considers each checkpoint element for elimination. It greedily removes the current checkpoint element if the remaining checkpoint elements preserve equivalence. The loop terminates after considering all the checkpoint elements and returns the last solution. Clearly, the capability of this synthesis loop is dependent on the robustness and capability of the verifier. If the verifier can verify intermittent programs with fewer checkpoint elements, only then can the synthesis loop result in a better solution.

We took benchmarks from previous work [37, 44] (all the DINO benchmarks are included) and used the synthesis loop and DINO to generate checkpoint elements at a given set of checkpoint nodes. For each benchmark, Table 3.1 shows the size of checkpoints generated by the synthesis loop and DINO for the same set of checkpoint nodes. The synthesis loop is able to generate checkpoints with 4x improvement over DINO, i.e., the data (in bytes) that needs to be checkpointed is on average 4 times less than that determined by DINO. The synthesis loop is able to perform better than DINO because of the precision in the model of the intermittent programs and the precision that we get while working at the assembly level (Section 3.6). Additionally, the synthesis loop benefits from the semantic reasoning over the syntactic reasoning done by DINO (Section 3.1).

3.6 Related work

We compare our work with the previous work on automatic instrumentation tools that generate intermittent programs, namely DINO [37], Ratchet [56] and Mementos [44]. These tools work in different settings and employ different strategies for checkpointing. In contrast, our work is complementary to these tools, and our verifier can be employed to validate their output.

DINO works with mixed-volatility programs, and given the checkpoint locations, it generates the intermittent programs automatically. It proposed a control flow based model of intermittence, where the control flow is extended with failure edges, going from all the nodes to the last executed checkpoints. This modeling is conservative and incomplete as it lacks semantics and does not model the effect of the power failures, unlike ours, where the failure edge is defined formally, in terms of the edge condition and the transfer function of a failure edge. Consequently, the model is not suitable for an application like equivalence checking. It then performs a syntactic WAR analysis (write-after-read without an intervening checkpoint) of non-volatile data on this extended control flow graph to determine the non-volatile data that needs to be checkpointed. Since it works at a higher level

and does not have a mapping between the machine registers and the program variables, it results in checkpointing all the registers and all the stack slots, often resulting in unnecessary checkpoint elements. Further, DINO does not work with intermediate observables and the output is not verified. Our work is complementary to DINO, in that our verifier can be used to validate DINO's output.

Ratchet is a fully-automatic instrumentation tool to generate intermittent programs from continuous programs. However, it takes a radically different approach of assuming that the whole memory is non-volatile, i.e., all program data including the stack and heap are deemed non-volatile. Only the machine registers are assumed to be volatile. Ratchet works by adding a checkpoint between every WAR occurrence on non-volatile data, i.e., it breaks every WAR occurrence. By breaking every WAR occurrence, correctness of non-volatile data across power reboots is ensured; for the machine registers, Ratchet simply saves the live machine registers at every checkpoint. These simplifications involve a performance cost, as it results in frequent checkpoints because the checkpoint locations are now determined by these WAR occurrences. Further, it is not always possible to insert a checkpoint between WAR occurrences within a single instruction (e.g., "`inc (nv)`"). Ratchet authors also do not allow intermediate observables. Finally, Ratchet's output can also be verified using our tool.

Mementos is a hardware-assisted fully-automatic instrumentation tool to generate intermittent programs. At each checkpoint location, it relies on hardware to determine the available energy and the checkpointing logic is only executed if the available energy is less than a threshold level, i.e., the checkpoints are conditional. This is an interesting idea and verification under such conditional checkpointing may seem non-trivial. Interestingly however, our verifier does not require any modification to work in this setting. The only difference would be that the number of failure and progress paths that get generated would be more: a checkpoint-to-checkpoint path can now bypass a successive checkpoint, resulting in a checkpoint-to-checkpoint path to a second level successor. For example, in our example, there will be also a progress path from node 1 to the exit, because the checkpoint

at node 4 is conditional.

Systems that tolerate power failures are not uncommon, file system is one example that is designed to tolerate power failures. The file system design has to ensure that across power failures, the disk layout remains consistent. In addition to power failures, it has to worry about disk write reorderings done by the disk controller. FSCQ [6] and Yggdrasil [49] are two recent papers that formally verified the file systems under power failures and reorderings. FSCQ is written in Coq and requires manual annotations and proofs for verification. Yggdrasil, on the other hand is an automatic technique. In FSCQ, the specifications are given in Crash Hoare Logic (CHL) which allows programmers to specify the expected behaviour under failures. The verification then entails proving that the file system follows the given specifications. In Yggdrasil, the behavioral specifications are provided as higher-level programs. The verification involves checking whether the file system is a *crash refinement* of the given specification, i.e., it produces states that are a subset of the states produced by the specification. The specifications in both the techniques are crash-aware, i.e., the specification encodes the behaviour under power failures. In contrast, we do not need specifications – our specifications are the continuous programs, which are not aware of crashes. The intermittent program should behave as if there are no power failures, i.e., equivalent to the continuous program. In addition, the problem of intermediate observables is unique to our setting. It would be interesting to explore if our technique can be used to verify file systems. Considering that our technique works smoothly with loops, it would remove Yggdrasil's important shortcoming of its inability to reason about loops in a uniform way.

Smart card embedded systems are another interesting example of systems that are designed to work with failures. These cards get powered by inserting in the appropriate terminal, and suddenly removing it during an operation may leave the card's data in an inconsistent state. A mechanism is added to restore a consistent state on the next insertion. A card has anti-tearing properties if it can always be restored to a consistent state after tearing (removal) at every execution state. Anti-tearing properties of smart cards are

important and previous work [1] formally verifies this by proving that tearing is safe at every program point in Coq. This technique is not automatic and requires manual proofs.

Our work overlaps with previous work on equivalence checking in the context of translation validation [24, 9, 8, 39, 48, 13, 25, 26, 51, 42, 30, 31, 52, 36, 55]. The goal of translation validation is to compute equivalence across compiler optimizations. On the other hand, our work targets equivalence across the instrumentation, albeit, under power failures. We have borrowed ideas from previous work, e.g., invariant inference is similar to that of [8, 9, 7], which are further based on Houdini [15]. In our work, we adopt these well-known techniques to verify intermittent programs against continuous programs.

Chapter 4

Conclusion and future directions

This thesis presents techniques to check program equivalence in a black-box manner, which is an essential construct of program synthesis techniques like superoptimization. We discuss equivalence checking across two different kind of transformations: compiler optimizations (Chapter 2) and transformations to make a program work in an intermittent environment (Chapter 3).

In the first part, we present a black-box technique to compute equivalence across compiler optimizations and test it across the optimizations produced by multiple compilers. We demonstrate the importance of handling undefined behaviour while checking equivalence across compiler optimizations. Our work is the first to handle undefined behaviour related optimizations in the equivalence checking for programs containing loops. We evaluate our equivalence checking algorithm across a large set of benchmarks, across optimizations produced by four modern compilers, namely GCC, LLVM (`clang`), ICC (Intel's C Compiler), and CompCert (`ccomp`). Our technique is sound and our success rates for black-box equivalence checking are comparable to previous equivalence checking tools, which do not operate in a black-box setting. Our study also led to the discovery of three bugs in the codebase of GCC and ICC. We also used our equivalence checker in an 32-bit x86 superoptimizer that supports a rudimentary form of loops. In our limited study, for common routines like initializing an array and comparing elements of two arrays, the

superoptimizer automatically synthesized optimized implementations that are up to 12x faster than the fastest code generated by the four compilers of our experimental setup.

Superoptimization is the primary target of our work, and we expect the support for loops to enable general-purpose loop-based optimizations in a superoptimizer. Synthesis using our equivalence checker is a natural future step, however, this is not the only opportunity of future work. A direction related to optimization is the synthesis of optimization *patterns* (instead of optimization instances). That is, one can use an equivalence checker to synthesize the optimization passes (i.e., optimization patterns), offline, and later, use these passes during the compilation and optimization. An equivalence checker is also an important construct for a certifying compiler, and this is another potential line of exploration. A certifying compiler generates an optimized implementation with its proof of correctness, i.e., the proof of equivalence, for the performed transformations. One can imagine a certifying compiler using the existing compilers and an equivalence checker. We can get the performance of aggressively optimizing compilers with the proof of correctness. It would be interesting to compare the performance of code generated by such a certifying compiler with that of CompCert. Apart from the various applications, it may be interesting to investigate how our technique performs together with the stronger (non black-box) assumptions that have been employed in previous work on equivalence checking, such as pass-by-pass verification, and knowledge of the potential transformations.

Using our equivalence checker, we also present a method to quantify the impact of undefined behaviour on compiler optimizations. We considered three undefined behaviour and found overwhelming relative significance of out-of-bounds variable access assumptions (for optimization), compared to other types of undefined behaviour like signed integer overflow and type based strict aliasing assumptions. There are hundreds of types of undefined behaviour in C, and some of them have been bitterly debated in the past [53, 54]. We believe that this approach to quantifying the impact of different types of undefined behaviour on compiler optimization, can bring some insight and basis for such debates. Moreover, such a study gives a quantified feedback to the language designers

about the usage of different behaviour assumptions by the compiler developers. This could be another future direction of research.

In the second part, we present a method to verify the correctness of intermittent programs. We present a precise formal model of intermittent programs (i.e, intermittence) to correctly and exhaustively capture the behaviour of intermittent programs for all possible scenarios of power failures. We present a robust technique to establish equivalence across a continuous program and its intermittent version. The algorithm is robust in the sense of its generality in handling even minimal checkpointing states, i.e, smaller checkpoints. Furthermore, we describe two properties of observables: *idempotence* and *commutativity*, which allow us to reason about recurring executions of observables in the intermittent environment. We evaluate our verifier by using it as a part of a synthesis tool, which identifies the program state elements that need be checkpointed for a given set of checkpoint locations. We achieve a significant improvement in checkpoint data size over previous state-of-the-art tools to generate intermittent programs, with the added advantage of the proof of correctness.

In future work, it would be interesting to synthesize checkpoint-locations along with checkpoint elements. This should free the user from specifying the checkpoint locations, and this should result in even better checkpoints for a given cost function. Another direction would be to verify the correctness of file systems, which are also meant to tolerate power failures. Considering that our technique is automatic and works smoothly with loops, it would be interesting to explore if our technique can be used for verifying the correctness of file systems.

Bibliography

[1] J. Andronick. Formally proved anti-tearing properties of embedded C code. *Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISOLA 2006)*, pages 129–136, 2006.

[2] S. Bansal and A. Aiken. Automatic generation of peephole superoptimizers. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 394–403. ACM, 2006. ISBN 1-59593-451-0. doi: 10.1145/1168857.1168906.

[3] S. Bansal and A. Aiken. Binary translation using peephole superoptimizers. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 177–192. USENIX Association, 2008.

[4] C. Barrett, Y. Fang, B. Goldberg, Y. Hu, A. Pnueli, and L. Zuck. TVOC: A translation validator for optimizing compilers. In K. Etessami and S. K. Rajamani, editors, *Computer Aided Verification*, pages 291–295, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. ISBN 978-3-540-31686-2.

[5] C11 Standard. *ISO/IEC 9899:2011, Programming Languages - C*, Dec. 2011.

[6] H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. Zeldovich. Using crash hoare logic for certifying the FSCQ file system. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 18–37, New

York, NY, USA, 2015. ACM. ISBN 978-1-4503-3834-9. doi: 10.1145/2815400.2815402. URL http://doi.acm.org/10.1145/2815400.2815402.

[7] B. Churchill, R. Sharma, J. Bastien, and A. Aiken. Sound loop superoptimization for Google native client. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 313–326. ACM, 2017. ISBN 978-1-4503-4465-4. doi: 10.1145/3037697.3037754.

[8] M. Dahiya and S. Bansal. Black-box equivalence checking across compiler optimizations. In *Proceedings of the Fifteenth Asian Symposium on Programming Languages and Systems*, APLAS '17, pages 127–147, 2017. ISBN 978-3-319-71237-6. doi: 10.1007/978-3-319-71237-6_7.

[9] M. Dahiya and S. Bansal. Modeling undefined behaviour semantics for checking equivalence across compiler optimizations. In *Hardware and Software: Verification and Testing - 13th International Haifa Verification Conference, HVC 2017*, HVC '17, pages 19–34, 2017. ISBN 978-3-319-70389-3. doi: 10.1007/978-3-319-70389-3_2.

[10] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340. ISBN 3-540-78799-2, 978-3-540-78799-0.

[11] S. Debray, R. Muth, and M. Weippert. Alias analysis of executable code. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 12–24. ACM, 1998. ISBN 0-89791-979-3. doi: 10.1145/268946.268948.

[12] B. Dutertre. Yices 2.2. In A. Biere and R. Bloem, editors, *Computer-Aided Verification (CAV'2014)*, volume 8559 of *Lecture Notes in Computer Science*, pages 737–744.

[13] D. Felsing, S. Grebing, V. Klebanov, P. Rümmer, and M. Ulbrich. Automating regression verification. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 349–360. ACM, 2014. ISBN 978-1-4503-3013-8. doi: 10.1145/2642937.2642987.

[14] M. Fernández and R. Espasa. Speculative alias analysis for executable code. In *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques*, PACT '02, pages 222–231. IEEE Computer Society, 2002. ISBN 0-7695-1620-3.

[15] C. Flanagan and K. Leino. Houdini, an annotation assistant for ESC/Java. In *FME 2001: Formal Methods for Increasing Software Productivity*, volume 2021 of *Lecture Notes in Computer Science*, pages 500–517. Springer Berlin Heidelberg, 2001. ISBN 978-3-540-41791-0. doi: 10.1007/3-540-45251-6_29.

[16] C. Flanagan, R. Joshi, and K. R. M. Leino. Annotation inference for modular checkers. *Inf. Process. Lett.*, 77(2-4):97–108, Feb. 2001. ISSN 0020-0190. doi: 10.1016/S0020-0190(00)00196-4. URL http://dx.doi.org/10.1016/S0020-0190(00)00196-4.

[17] A. Fog. Instruction tables, 2016.

[18] GCC bugzilla. GCC Bugzilla - Bug 68480. URL https://gcc.gnu.org/bugzilla/show_bug.cgi?id=68480.

[19] C. Hawblitzel, S. K. Lahiri, K. Pawar, H. Hashmi, S. Gokbulut, L. Fernando, D. Detlefs, and S. Wadsworth. Will you still compile me tomorrow? static cross-version compiler validation. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 191–201. ACM, 2013. ISBN 978-1-4503-2237-9. doi: 10.1145/2491411.2491442.

[20] ICC developer forums. ICC developer forums discussion: icc-16.0.3 not respecting no-ansi-alias flag?, . URL https://software.intel.com/en-us/forums/intel-c-compiler/topic/702187.

[21] ICC developer forums. ICC developer forums discussion: icc-16.0.3 not respecting fno-strict-overflow flag?, . URL https://software.intel.com/en-us/forums/intel-c-compiler/topic/702516.

[22] A. Kanade, A. Sanyal, and U. P. Khedker. Validation of GCC optimizers through trace generation. *Softw. Pract. Exper.*, 39(6):611–639, Apr. 2009. ISSN 0038-0644. doi: 10.1002/spe.v39:6.

[23] J. Kang, Y. Kim, Y. Song, J. Lee, S. Park, M. D. Shin, Y. Kim, S. Cho, J. Choi, C.-K. Hur, and K. Yi. Crellvm: Verified credible compilation for LLVM. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 631–645, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5698-5. doi: 10.1145/3192366.3192377. URL http://doi.acm.org/10.1145/3192366.3192377.

[24] S. Kundu, Z. Tatlock, and S. Lerner. Proving optimizations correct using parameterized program equivalence. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 327–337. ACM, 2009. ISBN 978-1-60558-392-1. doi: 10.1145/1542476.1542513.

[25] S. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebelo. Symdiff: A language-agnostic semantic diff tool for imperative programs. In *CAV '12*. Springer, 2012.

[26] S. Lahiri, R. Sinha, and C. Hawblitzel. Automatic rootcausing for program equivalence failures in binaries. In *Computer Aided Verification (CAV'15)*. Springer, 2015.

[27] V. Le, M. Afshari, and Z. Su. Compiler validation via equivalence modulo inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language*

Design and Implementation, PLDI '14, pages 216–226. ACM, 2014. ISBN 978-1-4503-2784-8. doi: 10.1145/2594291.2594334.

[28] V. Le, C. Sun, and Z. Su. Finding deep compiler bugs via guided stochastic program mutation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 386–399. ACM, 2015. ISBN 978-1-4503-3689-5. doi: 10.1145/2814270.2814319.

[29] J. Lee, Y. Kim, Y. Song, C.-K. Hur, S. Das, D. Majnemer, J. Regehr, and N. P. Lopes. Taming undefined behavior in LLVM. PLDI 2017, 2017.

[30] S. Lerner, T. Millstein, and C. Chambers. Automatically proving the correctness of compiler optimizations. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 220–231. ACM, 2003. ISBN 1-58113-662-5. doi: 10.1145/781131.781156.

[31] S. Lerner, T. Millstein, E. Rice, and C. Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 364–377. ACM, 2005. ISBN 1-58113-830-X. doi: 10.1145/1040305.1040335.

[32] X. Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *33rd ACM symposium on Principles of Programming Languages*, pages 42–54. ACM Press, 2006.

[33] X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, July 2009. ISSN 0001-0782. doi: 10.1145/1538788.1538814. URL http://doi.acm.org/10.1145/1538788.1538814.

[34] A. Leung, D. Bounov, and S. Lerner. C-to-Verilog translation validation. In *Formal Methods and Models for Codesign (MEMOCODE)*, pages 42–47, 2015. doi: 10.1109/MEMCOD.2015.7340466.

[35] N. P. Lopes and J. Monteiro. Automatic equivalence checking of programs with uninterpreted functions and integer arithmetic. *Int. J. Softw. Tools Technol. Transf.*, 18(4):359–374, Aug. 2016. ISSN 1433-2779. doi: 10.1007/s10009-015-0366-1.

[36] N. P. Lopes, D. Menendez, S. Nagarakatte, and J. Regehr. Provably correct peephole optimizations with Alive. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 22–32. ACM, 2015. ISBN 978-1-4503-3468-6. doi: 10.1145/2737924.2737965.

[37] B. Lucia and B. Ransford. A simpler, safer programming and execution model for intermittent systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 575–585, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3468-6. doi: 10.1145/2737924.2737978. URL http://doi.acm.org/10.1145/2737924.2737978.

[38] K. S. Namjoshi and L. D. Zuck. Witnessing program transformations. In F. Logozzo and M. Fähndrich, editors, *Static Analysis*, pages 304–323, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-38856-9.

[39] G. C. Necula. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 83–94. ACM, 2000. ISBN 1-58113-199-2. doi: 10.1145/349299.349314.

[40] J. Olivo, S. Carrara, and G. D. Micheli. Energy harvesting and remote powering for implantable biosensors. *IEEE Sensors Journal*, 11(7):1573–1586, July 2011. ISSN 1530-437X.

[41] Phoronix. GCC soars past 14.5 million lines of code. URL https://www.phoronix.com/scan.php?page=news_item&px=MTg3OTQ.

[42] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '98, pages 151–166. Springer-Verlag, 1998. ISBN 3-540-64356-7.

[43] A. Poetzsch-Heffter and M. Gawkowski. Towards proof generating compilers. *Electron. Notes Theor. Comput. Sci.*, 132(1):37–51, May 2005. ISSN 1571-0661. doi: 10.1016/j.entcs.2005.03.023.

[44] B. Ransford, J. Sorber, and K. Fu. Mementos: System support for long-running computation on RFID-scale devices. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 159–170, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0266-1. doi: 10.1145/1950365.1950386.

[45] M. C. Rinard and D. Marinov. Credible compilation with pointers. In *In Proceedings of the Workshop on Run-Time Result Verification*, 1999.

[46] H. Samet. Proving the correctness of heuristically optimized code. *Commun. ACM*, 21(7):570–582, July 1978. ISSN 0001-0782. doi: 10.1145/359545.359569.

[47] D. Sangiorgi. *Introduction to Bisimulation and Coinduction*. Cambridge University Press, New York, NY, USA, 2011. ISBN 1107003636, 9781107003637.

[48] R. Sharma, E. Schkufza, B. Churchill, and A. Aiken. Data-driven equivalence checking. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '13, pages 391–406. ACM, 2013. ISBN 978-1-4503-2374-1. doi: 10.1145/2509136.2509509.

[49] H. Sigurbjarnarson, J. Bornholt, E. Torlak, and X. Wang. Push-button verification of file systems via crash refinement. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 1–16, Berkeley, CA, USA, 2016. USENIX Association. ISBN 978-1-931971-33-1. URL http://dl.acm.org/citation.cfm?id=3026877.3026879.

[50] M. Stepp, R. Tate, and S. Lerner. Equality-based translation validator for LLVM. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 737–742. Springer-Verlag, 2011. ISBN 978-3-642-22109-5.

[51] O. Strichman and B. Godlin. Regression verification - a practical way to verify programs. In *Verified Software: Theories, Tools, Experiments*, volume 4171, pages 496–501. Springer Berlin Heidelberg, 2008. doi: 10.1007/978-3-540-69149-5_54.

[52] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner. Equality saturation: a new approach to optimization. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 264–276. ACM, 2009. ISBN 978-1-60558-379-2. doi: http://doi.acm.org/10.1145/1480881.1480915.

[53] L. Torvalds. GCC mailing list conversation., 2002. URL https://lkml.org/lkml/2007/5/7/213.

[54] L. Torvalds. GCC mailing list conversation., 2002. URL https://gcc.gnu.org/ml/gcc/2002-01/msg00395.html.

[55] J.-B. Tristan, P. Govereau, and G. Morrisett. Evaluating value-graph translation validation for LLVM. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 295–305. ACM, 2011. ISBN 978-1-4503-0663-8. doi: 10.1145/1993498.1993533.

[56] J. Van Der Woude and M. Hicks. Intermittent computation without hardware support or programmer intervention. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 17–32, Berkeley, CA, USA, 2016. USENIX Association. ISBN 978-1-931971-33-1.

[57] X. Wang, N. Zeldovich, M. F. Kaashoek, and A. Solar-Lezama. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13. ISBN 978-1-4503-2388-8. doi: 10.1145/2517349.2522728.

[58] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 283–294. ACM, 2011. ISBN 978-1-4503-0663-8. doi: 10.1145/1993498.1993532.

[59] A. Zaks and A. Pnueli. Covac: Compiler validation by program analysis of the cross-product. In *Proceedings of the 15th International Symposium on Formal Methods*, FM '08, pages 35–51. Springer-Verlag, 2008. ISBN 978-3-540-68235-6. doi: 10.1007/978-3-540-68237-0_5.

[60] J. Zhao, S. Nagarakatte, M. M. Martin, and S. Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 427–440. ACM, 2012. ISBN 978-1-4503-1083-3. doi: 10.1145/2103656.2103709.

[61] J. Zhao, S. Nagarakatte, M. M. Martin, and S. Zdancewic. Formal verification of ssa-based optimizations for LLVM. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 175–186. ACM, 2013. ISBN 978-1-4503-2014-6. doi: 10.1145/2491956.2462164.

[62] L. Zuck, A. Pnueli, Y. Fang, and B. Goldberg. VOC: A methodology for the translation validation of optimizing compilers. 9(3):223–247, 2003.

[63] L. Zuck, A. Pnueli, B. Goldberg, C. Barrett, Y. Fang, and Y. Hu. Translation and run-time validation of loop transformations. *Form. Methods Syst. Des.*, 27(3):335–360, 2005. doi: 10.1007/s10703-005-3402-z.