

**SCALABLE AND PRECISE RANGE ANALYSIS  
ON THE INTERVAL LATTICE**



RAPHAEL ERNANI RODRIGUES

**SCALABLE AND PRECISE RANGE ANALYSIS  
ON THE INTERVAL LATTICE**

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

ORIENTADOR: FERNANDO MAGNO QUINTÃO PEREIRA

Belo Horizonte

Janeiro de 2014



RAPHAEL ERNANI RODRIGUES

**SCALABLE AND PRECISE RANGE ANALYSIS  
ON THE INTERVAL LATTICE**

Dissertation presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

ADVISOR: FERNANDO MAGNO QUINTÃO PEREIRA

Belo Horizonte

January 2014

© 2014, Raphael Ernani Rodrigues.  
Todos os direitos reservados.

Rodrigues, Raphael Ernani

R696s Scalable and Precise Range Analysis on the Interval  
Lattice / Raphael Ernani Rodrigues. — Belo Horizonte,  
2014

xxviii, 79 f. : il. ; 29cm

Dissertação (mestrado) — Universidade Federal de  
Minas Gerais

Orientador: Fernando Magno Quintão Pereira

1. Computação — Teses. 2. Compiladores — Teses.  
I. Orientador. II. Título.

CDU 519.6\*33(043)



UNIVERSIDADE FEDERAL DE MINAS GERAIS  
INSTITUTO DE CIÊNCIAS EXATAS  
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

## FOLHA DE APROVAÇÃO

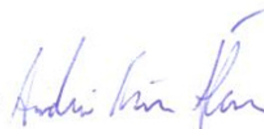
Scalable and precise range analysis on the interval lattice

**RAPHAEL ERNANI RODRIGUES**

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

  
PROF. FERNANDO MAGNO QUINTÃO PEREIRA - Orientador  
Departamento de Ciência da Computação - UFMG

PROF. ANDREI RIMSA ÁLVARES  
Departamento de Computação e Sistemas - UFOP



PROF<sup>a</sup>. LAURE GONNORD  
École Normale Supérieure de Lyon, FRANCE



PROF. LEONARDO BARBOSA E OLIVEIRA  
Departamento de Ciência da Computação - UFMG



Belo Horizonte, 27 de fevereiro de 2014.





*This work is dedicated to my parents, friends, professors and fellows, who have supported me throughout this journey. They have provided me with the vital conditions to accomplish this mission.*



# Acknowledgments

I'm grateful to my family, my friends, my professors, and my fellows at the university.

My family, specially my parents Gleyce and David, have provided the conditions for me to continue studying. They gave me a suitable environment, and allowed me to abstract from the bureaucracy of life in order to focus on my M.Sc. I am unconditionally grateful for all they have done for me since I was born.

My friends gave me pretty happy days. That was surely important for me to finish this program. Without their companionship, my days would be lonely, sad and dark, and I would not have psychological stability to finish this work.

My professors have guided me through this journey, pointing the dead ends and the promising opportunities. I'm very thankful for all their help, specially for the wise words and patience of Fernando, my adviser. His words have encouraged me to continue advancing, even during the hardest days and most difficult challenges of this long path. Laure Gonnord, from École Normale Supérieure de Lyon, also deserves special recognition for having nearly adopted me in my academic internship in Lyon.

I'm also thankful for the company of my fellows in the university. Igor Rafael deserves special thanks for giving me valuable tips about the life as a graduate student. In addition, also the loyal boys who inhabit the lab at room 3054 deserve special thanks, because they made not only my days better, but the M.Sc. as a whole a pleasant experience.

Finally, I must recognize the importance of CAPES and PPGCC to my formation. CAPES have provided the financial aid that allowed me to exclusively dedicate my time to the master's program. PPGCC have provided the formal conditions in the academic system. The good job done by PPGCC members have made my life easier as a student and let me keep focus on the really important things.



*“To become an expert, you have to master the fundamentals”*  
(Stephen Gilbert & Bill McCarthy)



# Resumo

Uma análise de largura de variáveis é um algoritmo que estima o menor e o maior valor que cada variável em um programa de computador pode assumir durante a execução deste programa. Este tipo de análise provê informação que permite ao compilador entender melhor os programas e realizar uma série de otimizações.

Muito trabalho já foi realizado no projeto e implementação de análises de largura de variáveis. Entretanto, soluções anteriores existentes na literatura têm sua aplicação prática bastante restrita, porque se baseiam em abordagens que são muito caras ou muito imprecisas.

Neste trabalho é apresentada uma implementação de análise de largura de variáveis que é atualmente o estado-da-arte neste campo de pesquisa, oferecendo o melhor balanço entre velocidade de análise e precisão de resultados.

Este trabalho também apresenta exemplos onde o uso da análise de largura de variáveis contribui para segurança computacional, projeto de hardware e otimização de programas.

Acreditamos que esta obra descreve a exploração mais completa dos benefícios da análise de largura de variáveis em programas grandes, presentes no mundo real.

**Palavras-chave:** Compiladores, Análise de Largura de Variáveis, Análise estática, Otimização.





# Abstract

A Range analysis is a technique that estimates the lowest and highest values that each variable in a computer program may assume during an execution of the program. This kind of analysis provides information that helps the compiler to better understand and optimize programs.

There has been much work in the design and implementation of range analyses. However, previous works have found limited practical application, because they rely on approaches that are either expensive or imprecise.

In this work we present an implementation of range analysis that is currently the state-of-art in this field of research, and provides the best balance between speed and precision.

We also present examples where the use of our range analysis contributes to software security, hardware design and program optimizations.

We believe that this work describes the most extensive exploration of the benefits of range analysis in large, real-world, programs.

**Palavras-chave:** Compilers, Range Analysis, Static Analysis, Optimization.



# Extended Abstract

A Range Analysis (RA) is an algorithm that estimates the lowest and highest values that each integer variable in a computer program may assume during an execution of the program. This kind of analysis provides information that helps the compiler to better understand and optimize programs.

However, previous works rely on approaches that are either expensive or imprecise, limiting their practical application. In this work we present an implementation of a new range analysis that is currently among the best tools of its kind publicly available. Our implementation provides the best balance between speed and precision. By relying on an idea that we deem future values - a key insight of our algorithm - we produce fast results that are comparable to the precision of more expensive solutions.

We also present the advantages of the application of our range analysis in different scenarios:

1. In **computing security**, to provide efficient protection against undefined behaviour caused, for instance, by integer overflows or access beyond the bounds of an array. Integer overflow occurs when a variable assumes a value that does not fit in the precision of the data type used for the variable while array-bound errors occur when accesses to an array continue beyond the end of the array. Our RA was applied successfully to remove unnecessary safety verifications in programs, enhancing the performance of safe programs.
2. In **hardware design**, to reduce the storage space and the wiring necessary to store a variable. RA can be used to prove that a given variable will not use the entire width of the data type assigned to it in the program. Therefore that variable can be stored into smaller registers and can use fewer lines to be transmitted between different areas of the processing element. The bitwidth reduction enabled by the RA produces faster programs that make a better use of the hardware and save energy.

3. In **static program analysis**, to expand the scope and improve the precision of static analysis routinely applied to code by optimizing compilers. A more precise RA can be used to infer the outcome of conditional tests in a program. For instance, with proven range bounds for the value of variables, a dead-code elimination may be able to prove more code to be dead, a constant-propagation may be able to propagate constants further, and an alias analysis may be able to reduce the size of alias sets. More precise alias sets may enable further data restructuring and automatic parallelization transformations that make better use of the memory hierarchy and of multi-core processor architectures.

In this work we present many contributions related to our range analysis. First, we present a new range analysis algorithm that relies on future values – the key insight of our algorithm – to gain precision without resorting to expensive techniques. Second, we present a technique to secure programs against integer overflows and show how we can use the RA to avoid inserting unnecessary checks. Third, we present u-SSA, a new program representation that is based on overflow-free programs and increases the precision of the RA. Finally, we present an heuristic to estimate the number of iterations of loops, based on patterns of variable updates.

This work summarizes a two years long effort that pushes the state of the art of the range analyses into a new level and demonstrates that it can be successfully used by program optimizations to produce smaller and faster machine code.

# List of Figures

2.1	Example program. . . . .	11
3.1	(a) Example program. (b) SSA form [Cytron et al. [1991]]. (c) e-SSA form [Bodik et al. [2000]]. (d) u-SSA form. . . . .	19
3.2	Growth on the number of instructions in comparison with SSA representation.	21
4.1	A suite of constraints that produce an instance of the range analysis problem.	24
4.2	(a) Example program. (b) Control Flow Graph in SSA form. (c) Constraints that we extract from the program. (d) Possible solution to the range analysis problem. . . . .	25
4.3	Our implementation of range analysis. Rounded boxes are optional steps. .	26
4.4	(a) The control flow graph from Figure 4.2(b) converted to standard e-SSA form. (b) A solution to the range analysis problem . . . . .	27
4.5	The dependence graph that we build to the program in Figure 4.4. . . . .	28
4.6	(Left) The lattice of the growth analysis. (Right) Cousot and Cousot’s widening operator. We evaluate the rules from left-to-right, top-to-bottom, and stop upon finding a pattern matching. Again: given an interval $\iota = [l, u]$ , we let $\iota_{\downarrow} = l$ , and $\iota_{\uparrow} = u$ . . . . .	30
4.7	Rules to replace futures by actual bounds. . . . .	30
4.8	Cousot and Cousot’s narrowing operator. . . . .	30
4.9	Four snapshots of the last SCC of Figure 4.4. (a) After removing control dependence edges. (b) After running the growth analysis. (c) After fixing the intersections bound to futures. (d) After running the narrowing analysis.	31
4.10	Correlation between program size (number of var nodes in constraint graphs after inlining) and analysis runtime (ms). Coefficient of determination = 0.967. . . . .	33

4.11	Comparison between program size (number of var nodes in constraint graphs) and memory consumption (KB). Coefficient of determination = 0.9947. . . . .	33
4.12	(Upper) Comparison between static range analysis and dynamic profiler for upper bounds. (Lower) Comparison between static range analysis and dynamic profiler for lower bounds. The numbers above the benchmark names give the number of variables in each program. . . . .	34
4.13	Design space exploration: precision (percentage of bitwidth reduction) versus speed (secs) for different configurations of our algorithm analyzing the SPEC CPU 2006 integer benchmarks. . . . .	36
4.14	Strongly Connected Components extracted from our example program. . .	36
4.15	(Left) Time to run our analysis without building strong components divided by time to run the analysis on strongly connected components. (Right) Precision, in bitwidth reduction, that we obtain with strong components, divided by the precision that we obtain without them. . . . .	37
4.16	(Left) Bars give the time to run analysis on e-SSA form programs divided by the time to run analysis on SSA form programs. (Right) Bars give the size of the e-SSA form program, in number of assembly instructions, divided by the size of the SSA form program. . . . .	38
4.17	The impact of the e-SSA transformation on precision for three different benchmark suites. Bars give the ratio of precision (in bitwidth reduction), obtained with e-SSA form conversion divided by precision without e-SSA form conversion. . . . .	39
4.18	Example where an intra-procedural implementation would lead to imprecise results. . . . .	40
4.19	Example where a context-sensitive implementation improves the results of range analysis. . . . .	41
4.20	The impact of inter-procedural analysis on precision. Each bar shows precision in %bitwidth reduction. . . . .	41
4.21	(Left) Runtime comparison between intra, inter and inter+inline versions of our algorithm. (Right) Runtime comparison between different widening operators. The bars are normalized to the time to run the intra-procedural analysis. . . . .	42
4.22	An example where jump-set widening is more precise. . . . .	43
5.1	An example of an exploitable integer overflow vulnerability. . . . .	46

5.2	Overflow checks. We use $\downarrow_n$ for the operation that truncates to $n$ bits. The subscript $s$ indicates a signed instruction; the subscript $u$ indicate an unsigned operation. . . . .	47
5.3	Number of instructions used in each check. . . . .	48
5.4	(a) A simple C function. (b) The same function converted to the LLVM intermediate representation. (c) The instrumented code. The boldface lines were part of the original program. . . . .	49
5.5	Percentage of overflow checks that our range analysis removes. Each bar is a benchmark in the LLVM test suite. Benchmarks have been ordered by the effectiveness of the range analysis. On average, we have eliminated 24.93% of the checks (geomean). . . . .	52
5.6	Comparison between execution times with and without pruning, normalized by the original program's execution time. . . . .	54
6.1	(a)Example program. (b) CFG of the program, after conversion to SSA form. (c)Dependence graph highlighting nodes that do not affect the loop predicate, after converting the original program into . . . . .	56
6.2	(a)Dependence graph. (b)Multi-node SCC of the variable $i_1$ . (c)Sequence of redefinitions of the variable $i_1$ . (d)Effect of one iteration on the variable $i_1$	59
6.3	Lattice of SR classifications. . . . .	60





# List of Tables

3.1	Impact of the transformation to e-SSA and u-SSA in terms of program size. # SSA: number of instructions in the SSA form program. # e-SSA: number of instructions in the e-SSA form program. # u-SSA: number of instructions in the u-SSA form program. . . . .	20
4.1	Variation in the precision of our analysis given the widening strategy. The size of each benchmark is given in number of variable nodes in the constraint graph. Precision is given in percentage of bitwidth reduction. Numbers in parenthesis are percentage of gain over 0 + Simple. . . . .	43
5.1	Instrumentation without support of range analysis. #I: number of LLVM bytecode instructions in the original program. #II: number of instructions that have been instrumented. #O: number of instructions that actually overflowed in the dynamic tests. . . . .	51
5.2	Instrumentation library with support of static range analysis. #II: number of instructions that have been instrumented without range analysis. #E: number of instructions instrumented in the e-SSA form program. #U: number of instructions instrumented in the u-SSA form program. . . . .	52
5.3	How the range analysis classified arithmetic instructions in the u-SSA form programs. #Sf: safe. #S: suspicious. #U: uncertain. #SO: number of suspicious instructions that overflowed. #UO: number of uncertain instructions that overflowed. . . . .	53
6.1	Natural Loops in the Control Flow Graph. L: number of natural loops. NL: number of nested loops. SEL: number of loops that have a single exit point.	65
6.2	Classification of Natural Loops according to their stop conditions. L: number of natural loops. IL: number of <i>Interval Loops</i> . EL: number of <i>Equality Loops</i> . OL: number of <i>Other Loops</i> . . . . .	65

6.3	Classification of Strongly Connected Components in the Dependence Graph. SN: number of <i>Single-Node</i> SCCs. MN: number of <i>Multi-Node</i> SCCs. SP: number of <i>Single-Path</i> SCCs. MP: number of <i>Multi-Path</i> SCCs. SL: number of <i>Single-Loop</i> SCCs. NL: number of <i>Nested-Loop</i> SCCs. . . . .	66
6.4	Trip Count Instrumentation. IL: interval loops. IIL: instrumented interval loops. EL: equality loops. IEL: instrumented equality loops. . . . .	67
6.5	Trip Count Profiler - Trip count estimated using vectors. . . . .	69
6.6	Trip Count Profiler - Trip count estimated using simplified heuristic. . . . .	69

# Contents

<b>Acknowledgments</b>	<b>xi</b>
<b>Resumo</b>	<b>xv</b>
<b>Abstract</b>	<b>xvii</b>
<b>Extended Abstract</b>	<b>xix</b>
<b>List of Figures</b>	<b>xxi</b>
<b>List of Tables</b>	<b>xxv</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Range Analysis . . . . .	1
1.2 Integer Overflows . . . . .	2
1.3 Trip Count Prediction . . . . .	3
1.4 Contributions . . . . .	3
1.5 Experimental results . . . . .	5
1.6 Publications and Software . . . . .	6
<b>2 Literature review</b>	<b>9</b>
2.1 Range Analysis . . . . .	9
2.2 Live Range Splitting . . . . .	11
2.3 Integer Overflows . . . . .	12
2.4 Trip Count Analysis . . . . .	14
<b>3 Live Range Splitting</b>	<b>17</b>
3.1 Live Splitting Alternatives . . . . .	17
3.2 Experiments . . . . .	20
3.3 Conclusion . . . . .	21

<b>4</b>	<b>Range Analysis</b>	<b>23</b>
4.1	Background . . . . .	23
4.3	Our Design of a Range Analysis Algorithm . . . . .	25
4.3.1	Finding Ranges in Strongly Connected Components . . . . .	29
4.3.2	Experiments . . . . .	32
4.4	Design Space . . . . .	35
4.4.1	Strongly Connected Components . . . . .	35
4.4.2	The Choice of a Program Representation . . . . .	37
4.4.3	Intra versus Inter-procedural Analysis . . . . .	39
4.4.4	Achieving Partial Context-Sensitiveness via Function Inlining . . . . .	40
4.4.5	Choosing a Widening Strategy . . . . .	42
4.5	Conclusion . . . . .	44
<b>5</b>	<b>Integer Overflows</b>	<b>45</b>
5.1	The Dynamic Instrumentation Library . . . . .	46
5.2	Experimental Results . . . . .	50
5.3	Conclusion . . . . .	53
<b>6</b>	<b>Trip count prediction</b>	<b>55</b>
6.1	Background . . . . .	55
6.1.1	Natural Loops . . . . .	56
6.1.2	Strongly Connected Components . . . . .	57
6.1.3	Sequences of Redefinitions of Variables . . . . .	58
6.1.4	Vectors . . . . .	60
6.1.5	Patterns of movement . . . . .	60
6.2	A Trip Count Algorithm Based on Vectors . . . . .	61
6.3	A Simplified Trip Count Algorithm Based on Vectors for JIT compilers . . . . .	63
6.4	Experimental Results . . . . .	64
6.5	Conclusion . . . . .	69
<b>7</b>	<b>Final considerations</b>	<b>71</b>
7.1	Future Works . . . . .	71
7.2	Conclusions . . . . .	72
	<b>Bibliography</b>	<b>73</b>

# Chapter 1

## Introduction

Range analysis is a compiler technique whose objective is to determine, statically, for each program variable, limits for the minimum and maximum values that this variable might assume during the program execution. In this work we propose a new range analysis algorithm that has linear space and time complexity and has a precision that is comparable to that of more expensive analyses. We also present a technique to secure programs against integer overflows. Furthermore, we show how we use our range analysis to eliminate unnecessary integer overflow checks. Finally, we present an algorithm to estimate the trip count of loops – the number of iterations that the loops of a program execute. We use parts of our range analysis to be more accurate in our estimates.

### 1.1 Range Analysis

The analysis of integer variables on the interval lattice has been the canonical example of abstract interpretation since its introduction in the seminal paper of Cousot and Cousot [1977]. Optimizing compilers use range analysis to infer the possible values that discrete variables may assume during program execution. This analysis has many uses. For instance, it allows the optimizing compiler to remove from the program text redundant overflow tests and unnecessary array bound checks (Bodik et al. [2000]; Gampe et al. [2011]). Furthermore, range analysis is essential to bitwidth aware register allocators (Barik et al. [2006]; Tallam and Gupta [2003]), register allocators that handle registers of different sizes (Kong and Wilken [1998]; Pereira and Palsberg [2008]; Scholz and Eckstein [2002]), and scratchpad cache allocators (Yang et al. [2011]). Additionally, range analysis has also been used to statically predict the outcome of branches (Patterson [1995]), to detect buffer overflow vulnerabilities (Simon [2008]; Wagner et al.

[2000]), to find the trip count of loops (Lokuciejewski et al. [2009]) and even to synthesize hardware (Cong et al. [2005]; Lhairech-Lebreton et al. [2010]; Mahlke et al. [2001]).

Given this great importance, it comes as no surprise that the compiler literature is rich in works describing in details algorithmic variations of range analyses (Mahlke et al. [2001]; Gawlitza et al. [2009]; Stephenson et al. [2000]; Su and Wagner [2005]). On the other hand, none of these authors provide experimental evidence that their approaches are able to deal with very large programs. There are researchers who have implemented range analyses that scale up to large programs (Patterson [1995]; Blanchet et al. [2003]; Venet and Brat [2004]); nevertheless, because the algorithm itself is not the main focus of their works, they neither give details about their design choices nor provide experimental data about it. This scenario was recently changed by Oh et al. [2012], who introduced an abstract interpretation framework which processes programs with hundreds of thousands of lines of code. Nevertheless, Oh *et al.* have designed a simple range analysis, which does not handle comparisons between variables, for instance. They do not discuss the precision of their implementation, but only its runtime and memory consumption. In this work we claim to push this discussion considerably further.

## 1.2 Integer Overflows

The most popular programming languages, including C, C++ and Java, limit the size of primitive numeric types. For instance, the `int` type, in C++, ranges from  $-2^{31}$  to  $2^{31} - 1$ . Consequently, there exists numbers that cannot be represented by these types. In general, these programming languages resort to a *wrapping-arithmetics* semantics (Warren [2002]) to perform integer operations. If a number  $n$  is too large to fit into a primitive data type  $T$ , then  $n$ 's value wraps around, and we obtain  $n$  modulo  $T_{max}$ . There are situations in which this semantics is acceptable, like Dietz et al. [2012] has shown. For instance, programmers might rely on this behavior to implement hash functions and random number generators. On the other hand, there exists also situations in which this behavior might lead a program to produce unexpected results. As an example, in 1996, the Ariane 5 rocket was lost due to an arithmetic overflow – a bug that resulted in a loss of more than US\$370 million (Dowson [1997]).

Programming languages such as Ada or Lisp can be customized to throw exceptions whenever integer overflows are detected. Furthermore, there exist recent works, like Brumley et al. [2007] and Dietz et al. [2012], that proposes to instrument bina-

ries derived from C, C++ and Java programs to detect the occurrence of overflows dynamically. Thus, the instrumented program can take some action when an overflow happens, such as to log the event, or to terminate the program. However, this safety has a price: arithmetic operations need to be surveilled, and the runtime checks cost time. Zhang et al. [2010] have eliminated some of this overhead via a tainted flow analysis. We have a similar goal, yet, our approach is substantially different.

## 1.3 Trip Count Prediction

Loops represent most of the execution time of a program. For that reason, there is a well-known aphorism that says that "all the gold lays in the loops", because compiler optimizations made inside loops have their benefits multiplied by the number of iterations actually executed. As a consequence of that fact, there is a vast number of works in the literature that are specialized in loop optimizations, like the ones described by Kennedy and Allen [2001] and Wolfe et al. [1995].

Some optimizations, however, are highly sensitive to the number of iterations of a given loop. For instance, if a given loop iterate a few times in an interpreter, an aggressive optimization made by a Just-In-Time (JIT) compiler may not even pay for the compilation overhead. On other hand, if the same loop iterates thousands of times, the JIT compilation might use more expensive techniques and still have a better end-to-end performance. The number of iterations a loop actually executes is called *Trip Count*. Here we use the same concept of trip count as described by [Wolfe et al., 1995, pp.200]. This number is only known at runtime, as it depends on the state of the variables of the program immediately before the loop starts.

Rice [1953] has demonstrated that predicting this information is an undecidable problem. Therefore, we can develop either a conservative solution or an heuristic to solve this problem. In this work, we present a heuristic that extracts patterns of the updates of variables' values and estimates the trip count of loops with symbolic expressions. Those expressions might, then, be evaluated at runtime and allow the compiler to decide dynamically what code to execute depending on the actual expected number of iterations.

## 1.4 Contributions

This work has four main contributions. First, we provide a complete description of a range analysis algorithm, and show extensive experimental data that justifies our

engineering choices. Our range analysis relies on a three-phase algorithm that results in good precision without resorting to expensive methods. Second, we present a technique to detect integer overflows and protect programs against them. We have used our range analysis to prove that some instructions will never cause integer overflows. Then, we can avoid inserting unnecessary checks in those instructions. Third, we present u-SSA, a new program representation that provides more information about the variables of overflow-free programs than previous representations. Finally, we bring a new algorithm to estimate the trip count of loops. Our trip count predictor uses the range analysis during the process of extracting symbolic expressions from the loops, representing its estimated trip counts.

**Range Analysis:** Our first algorithmic contribution on top of previous works is a three-phase approach to handle comparisons between variables without resorting to any exponential time technique. The few publicly available implementations of range analyses that we are aware of, such as those in FLEX <sup>1</sup>, gcc <sup>2</sup> or Mozilla’s IonMonkey <sup>3</sup> only deal with comparisons between variables and constants. Even theoretical works, such as Su and Wagner [2005] or Gawlitza et al. [2009] suffer from this limitation. This deficiency is one of the reasons explaining why none of these works has made their way into industrial-strength compilers. Two other insights allow our implementation to scale up to very large programs. We use Bodik’s Extended Static Single Assignment (e-SSA) form (Bodik et al. [2000]) to perform path-sensitive range analysis sparsely. This program representation ensures that the interval associated with a variable is constant along its entire live range. Finally, we process the strongly connected components that underline our constraint system in topological order. It is well-known that this technique is essential to speedup constraint solving algorithms ([Nielson et al., 1999, Sec 6.3]); however, due to our three-phase approach, a careful propagation of information along strong components not only gives us speed, but also improves the precision of our results.

**Integer Overflow Checks:** Our second algorithmic contribution is a technique to identify integer overflows in programs. Like Dietz et al. [2012] and Brumley et al. [2007], we insert dynamic checks inside the code of the target programs. Our dynamic checks use the resulting value of the integer instructions and their operands to

---

<sup>1</sup>The MIT’s FLEX/Harpoon compiler provides an implementation of Stephenson’s algorithm (Stephenson et al. [2000]), and is available at <http://flex.cscott.net/Harpoon/>.

<sup>2</sup>Gcc’s VRP pass (at <http://gcc.gnu.org/svn/gcc/trunk/gcc/tree-vrp.c>) implements a variant of Patterson’s algorithm (Patterson [1995]).

<sup>3</sup>[hg.mozilla.org/projects/ionmonkey/file/629d1e6251b9/js/src/ion/RangeAnalysis.cpp](http://hg.mozilla.org/projects/ionmonkey/file/629d1e6251b9/js/src/ion/RangeAnalysis.cpp)



decide whether an overflow has occurred or not. This, of course, creates an undesired overhead during the runtime. However, we have noticed that a large number of checked instructions never actually overflow. We, then, use our range analysis to identify those instructions that are guaranteed to never overflow and avoid inserting overflow checks in them. By pruning the unnecessary checks, we were able to eliminate part of the overhead, which means that we have increased the efficiency of the safe programs. Our experiments show that the overhead that our remaining overflow checks cause is negligible.

**u-SSA - A new program representation:** Our third contribution is a program representation that extends Bodik’s e-SSA and provides additional information about variables of programs that are safe against integer overflows. We have observed that some properties hold when we ensure that the program terminates in face of integer overflows. Thus, we have extended e-SSA and included more divisions in the live range of variables that allow us to increase the precision of our range analysis.

**Trip count prediction:** Our fourth algorithmic contribution is an heuristic to compute the symbolic trip count of loops. This information is important to estimate the complexity of the program and to let the compiler to generate different code for loops that will iterate small or large numbers of times. As the exact static computation of the trip count of loops is impossible, there is no optimal algorithm to solve this problem. Therefore, in this work we propose a new heuristic to estimate such number of iterations. Our algorithm identifies patterns under which the variables are updated between two iterations and derives vectors that represent how the variable move over the real line. When we know how the variables move, we can estimate the number of iterations needed for them to reach a termination state.

## 1.5 Experimental results

We have implemented our algorithms in the LLVM compiler (Lattner and Adve [2004]), and have used it to process a set of benchmarks with 2.72 million lines of C code.

**Range Analysis:** As we show in Section 4.3.2, our Range Analysis implementation, is able to analyze programs with over one million assembly instructions within fifteen seconds. And our implementation is not a straw-man: it produces very precise results. We have compared the ranges that our implementation finds with the results

obtained via a dynamic profiler, which we have also implemented. As we show in Section 4.3.2, when analyzing well-known numeric benchmarks we are able to estimate tight ranges for almost half of all the integer variables present in these programs. Our results are similar to Stephenson et al. [2000], even though our analysis does not require a backward propagation phase. Furthermore, we have been able to find tight bounds to the majority of the examples used by Costan et al. [2005] and Lakhdar-Chaouch et al. [2011], who rely on more costly methods.

**Integer Overflow Checks:** We use our range analysis to reduce the runtime overhead imposed by a dynamic instrumentation library. This instrumentation framework, which we describe in Section 5.1, has been implemented in the LLVM compiler. We have logged overflows in a vast number of programs, with special focus on SPEC CPU 2006 benchmarks. We have re-discovered the integer overflows recently observed by Dietz et al. [2012]. The performance of our instrumentation library, even without the support of range analysis, is within the 5% runtime overhead of Brumley et al. [2007]’s state-of-the-art algorithm. The range analysis halves down this overhead. Our static analysis algorithm avoids 24.93% of the overflow checks created by the dynamic instrumentation framework. With this support, the instrumented SPEC programs are only 1.73% slower. Therefore, we show in this paper that securing programs against integer overflows is very cheap.

**Trip count prediction:** As we show in Section 6.4, our trip count heuristics present a good balance between speed and precision. The simpler analysis, that aims JIT compilers, has been shown to offer a good precision, even without the use of our Range Analysis. It was able to infer bounds for 75% of the loops of our benchmarks, of which 66% was shown to be precise by our profiler. Our second analysis, that is more elaborated and aims regular compilers, has been shown to be even more precise. We have been able to provide bounds to 75% of the loops, of which 75% was shown to be precise. Similar works, such as Gulwani et al. [2009a]’s are able to provide symbolic bounds to 90% of the loops, but rely on a much more expensive technique, that limits their application to smaller programs.

## 1.6 Publications and Software

**Publications:** Most of this work has already been published in three papers. The first one, "Speed and Precision in Range Analysis" (Campos et al. [2012]), describes our

range analysis algorithm and presents the engineering choices we have made. It also brings the experimental results of our implementation. In chapter 4 we provide more details about the algorithm and its experimental results. The second one, "A fast and low-overhead technique to secure programs against integer overflows" (Rodrigues et al. [2013]), presents our technique to identify integer overflows and to secure programs against them. The third one, "Prevenção de Ataques de Não-Terminação baseados em Estouros de Precisão" (Rodrigues and Pereira [2013]) shows how our technique to secure programs against integer overflows can be used to avoid non-termination in programs. Chapter 5 extends the discussion presented in the papers about integer overflow handling.

Two other papers containing our trip count algorithm are currently under development. The first one describes in details our algorithm, engineering choices and results. The second one, "Selective Page Migration in ccNUMA Systems", is the result of a joint effort with researchers from UNICAMP and ETH Zurich. It shows how our trip count prediction can be used to make a more efficient use of the memory hierarchy. Both papers are being prepared to be submitted to international conferences in February 2014.

**Software:** All the software produced during the research of this work is publicly available at <http://code.google.com/p/range-analysis>. In our website we maintain a repository of source code, a description our algorithm, a list of examples and instructions for new users to apply our analysis in their own projects.



# Chapter 2

## Literature review

In this chapter we discuss a list of works that are strongly related to our studies. Our goal here is to evaluate the previous results available in the literature and show where we have made advances. Thus, in section 2.1 we discuss briefly how our Range Analysis algorithm overcome previous approaches and what are its limitations. We also evaluate the evolution of the program representations, that have created suitable conditions to develop our algorithm. Section 2.3 shows the works with goals that are similar to our integer overflow protection. Finally, section 2.4 discusses the papers in the literature that are related to our trip count heuristics.

### 2.1 Range Analysis

In this section we make the literature review of works related to our Range Analysis. We discuss how our Range Analysis is compared with previous analyses available in the literature. In addition, we show how the area has evolved along the decades and how we place our contribution in this evolution. Furthermore, we compare our approach with existing alternatives, both in terms of scalability and precision.

Range analysis is an old ally of compiler writers. The notions of widening and narrowing were introduced by Cousot and Cousot [1977] in one of the most cited papers in computer science. Different algorithms for range analysis have been later proposed by Patterson [1995], Stephenson et al. [2000], Mahlke et al. [2001] and many other researchers. Recently there have been many independent efforts to find exact, polynomial time algorithms to solve constraints on the interval lattice, as we can see in Gawlitza et al. [2009], Su and Wagner [2005], Costan et al. [2005], Lakhdar-Chaouch et al. [2011], and Su and Wagner [2004]. However, these works are still very theoretical, and have not yet been used to analyze large programs. Contrary to them, our approach

has a strong practical engineering bias and is shown to be able to analyze programs with millions of assembly instructions in less than 15 seconds.

The abstract interpretation framework introduced by Cousot and Cousot [1977] does not apply only to range analysis. It was initially designed for safety proofs, because of its capability of extracting properties of variables, functions or even entire programs without executing the source code. Those properties could, then, be used to find errors at compile time, as we can see in the works of Clarke et al. [1994] and Flanagan et al. [2002]. Furthermore, abstract interpretation is used in many data flow analyses, such as Liveness Analysis, Available Expressions, Reaching Definitions and Constant Propagation [Schwartzbach, 2008, pp.17].

There have been many practical approaches to abstract interpretation, with special emphasis on range analysis, such as Gampe et al. [2011], Blanchet et al. [2003], Bertrane et al. [2010], Cousot et al. [2009] and Jung et al. [2005]. Cousot’s group, for instance, has been able to globally analyze programs with thousands of lines of code, albeit using domain specific tools. The tool Astrée, for example, only analyzes programs that do not contain recursive calls. The work that is the closest to ours is the recent abstract interpretation framework of Oh et al. [2012]. Oh et al. discuss an implementation of range analysis on the interval lattice that scales up to a program with  $1,363KLoC$  (ghostscript-9.00). Because their focus is speed, they do not provide results about precision. We could not find the benchmarks used in those experiments for a direct comparison – the distribution of ghostscript-9.00 available in the LLVM test suite has  $27KLoC$ . On the other hand, we globally analyzed our largest benchmark, SPEC CPU 2006’s 403.gcc, enabling function inlining, in less than 15 seconds. 403.gcc has  $521KLoC$  and, during our analysis,  $1,419K$  LLVM IR instructions. Oh et al.’s implementation took orders of magnitude more time to go over programs of similar size. However, whereas they provide a framework to develop general sparse analyses, we only solve range analysis on the interval lattice.

There are in the literature works that are more powerful than ours. Figure 2.1 shows an example that illustrates one limitation of our approach. In this example, variables  $i$  and  $s$  are initialized with the same value and are always updated together. However, because  $i$  and  $s$  does not have any syntactic dependence, our range analysis gives the range  $[0, 10]$  for variable  $i$  and  $[0, +\infty]$  for variable  $s$ . Nevertheless, relational abstract domains such as the polyhedron domain of Cousot and Halbwachs [1978] or the octagon domain of Miné [2006] can handle this kind of problem. The advantage of our approach is that we perform the analysis with a lower computational complexity. Our experiments show that the precision that we lose is not significant, when comparing our results against the results of more expensive analyses.

```
1: int foo(){
2:     int i = 0;
3:     int s = 0;
4:     while (i < 10){
5:         i++;
6:         s++;
7:     }
8:     return s;
9: }
```

Figure 2.1. Example program.

## 2.2 Live Range Splitting

In this work we present a sparse implementation of range analysis. Sparsity, in our context, means that we associate points in the lattice of interest – intervals in our case – directly to variables. Dense analyses map such information to pairs formed by variables and program points. The compiler related literature contains many descriptions of sparse data-flow analyses. Some among these analyses obtain sparsity by using specific program representations, like we did. Others rely on data-structures. In terms of data-structures, the first, and best known method proposed to support sparse data-flow analyses is the Sparse Evaluation Graph (SEG) of Choi et al. [1991]. The nodes of this graph represent program regions where information produced by the data-flow analysis might change. Choi et al.’s ideas have been further expanded, for example, by the Quick Propagation Graphs of Johnson and Pingali [1993], or the Compact Evaluation Graphs of Ramalingam [2002]. Building upon Choi’s pioneering work, researchers have developed many efficient ways to build such graphs. Examples of that can be found in Pingali and Bilardi [1995], Pingali and Bilardi [1997], and Johnson et al. [1994]. These data-structures have been shown to improve many data-flow analyses in terms of runtime and memory consumption. Nevertheless, the elegance of SEGs and its successors have not, so far, been enough to attract the attention of mainstream compiler writers. Compilers such as gcc, LLVM or Java Hotspot rely, instead, on several types of program representations to provide support to sparse data-flow analyses.

Most eminent among these representations is the Static Single Assignment form presented by Cytron et al. [1991], which suits well forward flow analyses, such as *reaching definitions*. Since its first presentation, the SSA form has been expanded in different ways. For instance, the Gated SSA form allows the static association of logical predicates with data-flow paths, as we can see in Ottenstein et al. [1990] and Tu and Padua [1995]. Ananian [1999] has introduced in the late nineties the Static Single

Information (SSI) form, a program representation that supports both forward and backward analyses. This representation was later revisited by Singer [2006] and, a few years later, by Boissinot et al. [2009]. Singer provided new algorithms plus examples of applications that benefit from the SSI form, and Boissinot et al., in an effort to clarify some misconceptions about this program representation, introduced the notions of weak and strong SSI form. Another important representation, which supports data-flow analyses that acquire information from uses, is the Static Single Use form (SSU). There exists many variants of SSU, as shown in the works of Plevyak [1996], George and Matthias [2003], and Lo et al. [1998]. For instance, the “strict” SSU form enforces that each definition reaches a single use, whereas SSI and other variations of SSU allow two consecutive uses of a variable on the same path. The program representation that we have used in this work – the Extended Static Single Assignment (e-SSA) form – was introduced by Bodik et al. [2000]. The program representation

There are so many different program representations because they fit specific data-flow problems. Each representation, given a domain of application, provides the following property: the information associated with the live range of a variable is invariant along every program point where this variable is alive. There are two key aspects that distinguish one representation from the others: firstly, where the information about a variable is acquired, and secondly, how this information is propagated. The e-SSA form, for instance, supports flow analyses that obtain information both from variable definitions and conditional tests and propagate this information forwardly. Such analyses are also supported by the SSI form; hence, we could have used this other representation too. However, Tavares et al. [201X] have shown in previous work that the e-SSA form is considerably more economical.

## 2.3 Integer Overflows

In this work we use the Range Analysis to eliminate unnecessary integer overflow checks. By doing this, we are entering in a completely different field of research. Thus, in this section we show what other researchers have already presented in this area. We compare our solution with the previously existing ones.

**Dynamic Detection of Integer Overflows:** We say a method of detection of integer overflows is dynamic when we need to actually execute the target program to analyze it. Such methods may be implemented using many different methods. Brumley et al. [2007] have developed a tool, RICH, to secure C programs against



integer overflows. The author’s approach consists in instrumenting every integer operation that might cause an overflow, underflow, or data loss. The main result of Brumley *et al.* is the verification that guarding programs against integer overflows does not compromise their performance significantly: the average slowdown across four large applications is 5%. RICH uses specific features of the x86 architecture to reduce the instrumentation overhead. Chinchani *et al.* [2004] follow a similar approach, describing each arithmetic operation formally, and then using characteristics of the computer architecture to detect overflows at runtime. Differently from these previous works, we instrument programs at LLVM’s intermediate representation level, which is machine independent. Nevertheless, the performance of the programs that we instrument is on par with Brumley’s, even without the support of the static range analysis to eliminate unnecessary checks. Furthermore, our range analysis could eliminate approximately 45% of the runtime overhead that the tests that a naive implementation of Brumley’s technique would insert.

Dietz *et al.* [2012] have implemented a tool, IOC, that instruments the source code of C/C++ programs to detect integer overflows. They approach the problem of detecting integer overflows from a software engineering point-of-view; hence, performance is not a concern. The authors have used IOC to carry out a study about the occurrences of overflows in real-world programs, and have found that these events are very common. It is also possible to implement a dynamic analysis without instrumenting the target program. In this case, developers must use some form of code emulation. Chen *et al.* [2009], for instance, uses a modified Valgrind<sup>1</sup> virtual machine to detect integer overflows. The main drawback of emulation is performance: Chen *et al.* report a 50x slowdown. We differ from all these previous works because we focus on generating less instrumentation, an endeavor that we accomplish via static analysis.

**Static Detection of Integer Overflows:** We say a method of detection of integer overflows is static when all the analysis is done at compile time, without actually executing the target program. Zhang *et al.* [2010] have used static analysis to sanitize programs against integer overflow based vulnerabilities. They instrument integer operations in paths from a source to a sink. In Zhang *et al.*’s context, sources are functions that read values from users, and sinks are memory allocation operations. Thus, contrary to our work, Zhang *et al.*’s only need to instrument about 10% of the integer operations in the program. However, they do not use any form of range analysis to limit the number of checks inserted in the transformed code. Wang *et al.*

---

<sup>1</sup>Nethercote and Seward [2007]

[2009] have implemented a tool, IntScope, that combines symbolic execution and taint analysis to detect integer overflow vulnerabilities. The authors have been able to use this tool to successfully identify many vulnerabilities in industrial quality software. Our work and Wang *et al.*'s work are essentially different: they use symbolic execution, whereas we rely on range analysis. Contrary to us, they do not transform the program to prevent or detect such event dynamically. Still in the field of symbolic execution, Molnar et al. [2009] have implemented a tool, SmartFuzz, that analyzes Linux x86 binaries to find integer overflow bugs. They prove the existence of bugs by generating test cases for them.

## 2.4 Trip Count Analysis

We also have used our Range Analysis to develop an heuristic to statically estimate the trip count of a loop. That part of this work has its own related works, that we describe here.

It is possible to estimate the trip count of loops in many different ways, in a trade-off between speed and precision. In order to estimate the trip count of loops, Ermedahl and Gustafsson [1997], Halbuchs et al. [1997], Gulavani and Gulwani [2008], and Gulwani et al. [2009b] have used abstract interpretation. Lundqvist and Stenström [1998] and Liu and Gomez [1998] have used symbolic execution to achieve similar goals. Although those techniques are quite powerful, they are also computationally expensive. Thus, their application is limited by the size of programs to be analyzed. Nevertheless, the high complexity does not mean perfect precision. Some of those works have restrictions with regards to the structure of the analyzed loops. For instance, some of them only analyze loops with a single path and are very conservative while analyzing nested loops. Our work aims to find a better balance between speed and precision.

In an effort similar to ours, Gulwani et al. [2009a] have developed a new approach to estimate the number of iterations of a loop. They have proposed the *Control-Flow Refinement*, a conversion of the programs into a suitable representation, that allowed them to handle programs that other algorithms were not able to analyze. That representation allowed them to find symbolic bounds for 90% of the programs they have analyzed. However, they still rely on expensive techniques. For instance, their implementation requires a theorem prover. Such tools often rely on solutions to NP-complete problems. Differently to their work, here we present two heuristics to estimate the number of iterations of loops that use simpler techniques. Despite of the simplicity of our algorithms, our results show that we offer a good precision without

resorting to expensive techniques.



# Chapter 3

## Live Range Splitting

A dense data-flow analysis associates information, i.e., a point in a lattice, with each pair formed by a variable plus a program point. If this information is invariant along every program point where the variable is alive, then we can associate the information with the variable itself. In this case, we say that the data-flow analysis is *sparse*, as defined by Choi et al. [1991]. In cases like our range analysis, a dense data-flow analysis can be transformed into a sparse one via a suitable intermediate representation. A compiler builds this intermediate representation by splitting the live ranges of variables at the program points where the information associated with these variables might change. In order to split the live range of a variable  $v$ , at a program point  $p$ , we insert a copy  $v' = v$  at  $p$ , and rename every use of  $v$  that is dominated by  $p$ . In this work we have experimented with two different live range splitting alternatives.

### 3.1 Live Splitting Alternatives

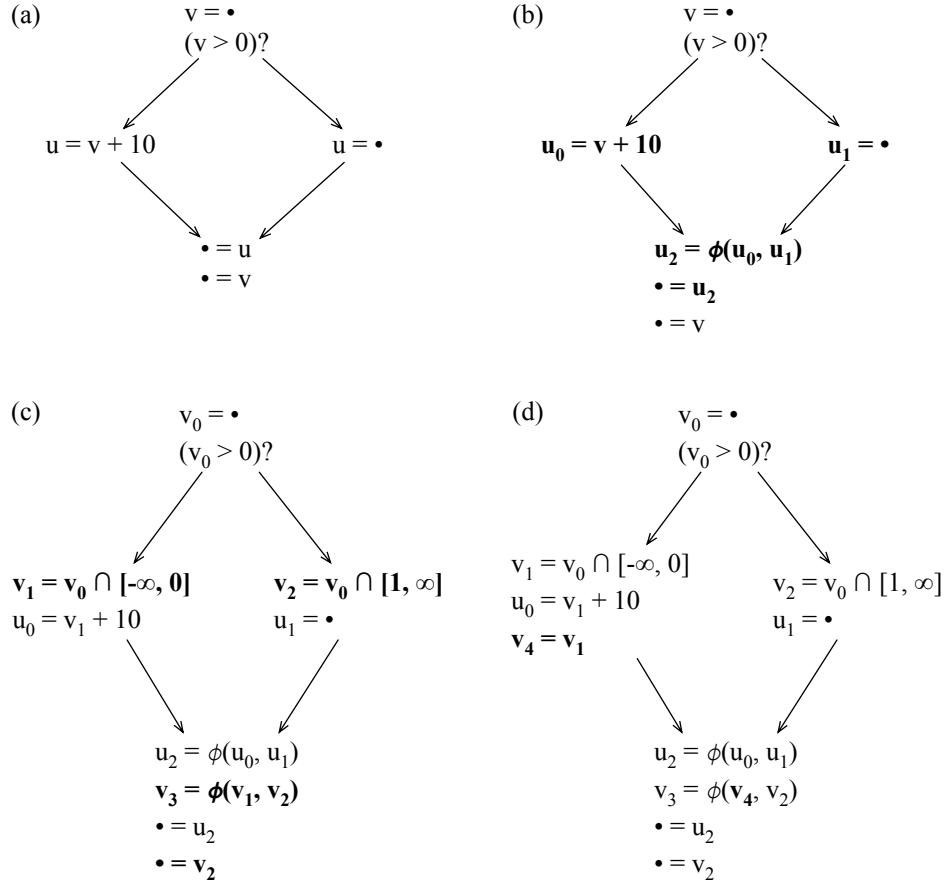
The first strategy is the *Extended Static Single Assignment* (e-SSA) form, proposed by Bodik et al. [2000]. We build the e-SSA representation by splitting live ranges at definition sites – hence it subsumes the SSA form – and at conditional tests. Let  $(v < c)?$  be a conditional test between two integers, and let  $l_t$  and  $l_f$  be labels in the program, such that  $l_t$  is the target of the test if the condition is true, and  $l_f$  is the target when the condition is false. We split the live range of  $v$  at any of these points if at least one of two conditions is true: (i)  $l_f$  or  $l_t$  dominate any use of  $v$ ; (ii) there exists a use of  $v$  at the dominance frontier of  $l_f$  or  $l_t$ . For the notions of dominance and dominance-frontier, see [Aho et al., 2006, p.656]. To split the live range of  $v$  at  $l_f$  we insert at this program point a copy  $v_f = v \sqcap [c, +\infty]$ , where  $v_f$  is a fresh name. We then rename every use of  $v$  that is dominated by  $l_f$  to  $v_f$ . Dually, if we must split at  $l_t$ , then

we create at this point a copy  $v_t = v \sqcap [-\infty, c - 1]$ , and rename variables accordingly. If the conditional uses two variables, e.g.,  $(v_1 < v_2)?$ , then we create intersections bound to *futures*. We insert, at  $l_f$ ,  $v_{1f} = v_1 \sqcap [\mathbf{ft}(v_2), +\infty]$ , and  $v_{2f} = v_2 \sqcap [-\infty, \mathbf{ft}(v_1)]$ . Similarly, at  $l_t$  we insert  $v_{1v} = v_1 \sqcap [-\infty, \mathbf{ft}(v_2) - 1]$  and  $v_{2v} = v_2 \sqcap [\mathbf{ft}(v_1) + 1, +\infty]$ . A variable  $v$  can never be associated with a future bound to itself, e.g.,  $\mathbf{ft}(v)$ . This invariant holds because whenever the e-SSA conversion associates a variable  $u$  with  $\mathbf{ft}(v)$ , then  $u$  is a fresh name created to split the live range of  $v$ .

The second intermediate representation consists in splitting live ranges at (i) definition sites – it subsumes SSA, (ii) at conditional tests – it subsumes e-SSA, and at some use sites. This representation, which we henceforth call u-SSA, is only valid if we assume that integer overflows cannot happen. We can provide this guarantee by using our dynamic instrumentation – described in Chapter 5 – to abort the execution of a program in face of an overflow. The rationale behind u-SSA is as follows: we know that past an instruction such as  $v = u + c, c \in \mathbb{Z}$  at a program point  $p$ , variable  $u$  must be less than  $MaxInt - c$ . If that were not the case, then an overflow would have happened and the program would have terminated. Therefore, we split the live range of  $u$  past its use point  $p$ , producing the sequence  $v = u + c; u' = u$ , and renaming every use of  $u$  that is dominated by  $p$  to  $u'$ . We then associate  $u'$  with the constraint  $I[U'] \sqsubseteq I[U] \sqcap [-\infty, MaxInt - c]$ .

Figure 3.1 compares the u-SSA form with the SSA and e-SSA intermediate program representations. We use the notation  $v = \bullet$  to denote a definition of variable  $v$ , and  $\bullet = v$  to denote a use of it. Figure 3.1(b) shows the example program converted to the SSA format. Different definitions of variable  $u$  have been renamed, and a  $\phi$ -function joins these definitions into a single name. The SSA form sparsifies a data-flow analysis that only extracts information from the definition sites of variables, such as constant propagation. Figure 3.1(c) shows the same program in e-SSA form. This time we have renamed variable  $v$  right after the conditional test where this variable is used. The e-SSA form serves data-flow analyses that acquire information from definition sites and conditional tests. Examples of these analyses include array bounds checking elimination (Bodik et al. [2000]) and traditional implementations of range analyses (Gough and Klaeren [1994]; Patterson [1995]). Finally, Figure 3.1(d) shows our example in u-SSA form. The live range of variable  $v_1$  has been divided right after its use. This representation assists analyses that learn information from the way that variables are used, and propagate this information forwardly.

Although the live range splitting makes possible to extract more information from the programs, it has a drawback: the growth in the number of instructions. The increment on the program size, however, does not mean a higher register pressure,



**Figure 3.1.** (a) Example program. (b) SSA form [Cytron et al. [1991]]. (c) e-SSA form [Bodik et al. [2000]]. (d) u-SSA form.

because when we split a variable  $v$  into  $v_1$  and  $v_2$ , we will have only one of the three variables live at once, depending on the execution flow of the program. That means that the register allocator will assign the same register to the variables  $v$ ,  $v_1$ , and  $v_2$ , keeping the register pressure in the same level in both representations. However, although this problem that is not impossible to overcome, it can bring some extra difficulties to further analyses with high asymptotic complexity in function of the program size. In this case, there is a trade-off between speed of analysis and precision of results. In Section 4.4.2 we present a further discussion of this trade-off in the context of our Range Analysis.

Benchmark	# SSA	# e-SSA	% e-SSA/SSA	# u-SSA	% u-SSA/SSA
429.mcf	2633	2742	104.14%	2816	106.95%
470.lbm	3656	3718	101.70%	4058	111.00%
462.libquantum	6026	6250	103.72%	6503	107.92%
473.astar	8576	8920	104.01%	9261	107.99%
401.bzip2	17045	17845	104.69%	18865	110.68%
433.milc	23418	24355	104.00%	24942	106.51%
458.sjeng	30590	31780	103.89%	33191	108.50%
450.soplex	62865	64301	102.28%	65541	104.26%
456.hmmer	65163	68018	104.38%	70205	107.74%
444.namd	68050	71319	104.80%	73016	107.30%
471.omnetpp	92057	94377	102.52%	95507	103.75%
464.h264ref	136174	140274	103.01%	145185	106.62%
447.dealII	424250	434417	102.40%	442983	104.42%
483.xalancbmk	585212	599251	102.40%	604610	103.31%
Total	1525715	1567567	102.74%	1596683	104.65%

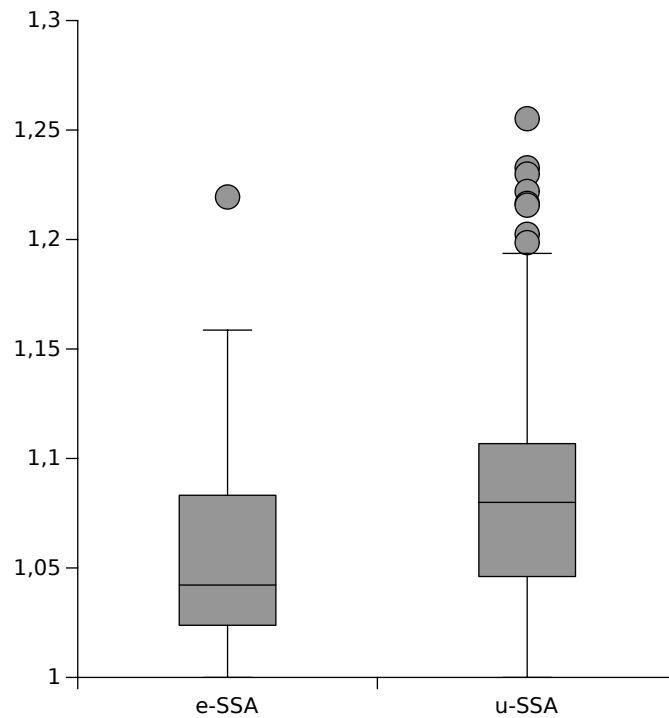
**Table 3.1.** Impact of the transformation to e-SSA and u-SSA in terms of program size. # SSA: number of instructions in the SSA form program. # e-SSA: number of instructions in the e-SSA form program. # u-SSA: number of instructions in the u-SSA form program.

## 3.2 Experiments

We have observed the actual impact of the transformation of programs to e-SSA and u-SSA forms. Table 3.1 shows the effect of the transformation in the programs of the SPEC CPU 2006 benchmarks. According to the table, we can observe that both representations lightly affect the number of instructions of the programs. From those benchmarks, we can see that e-SSA causes an average growth of 2.74% and a maximum growth of 4.8% of the program size, while u-SSA causes growths of 4.65% and 11.00% respectively. Those results, however, are not exclusive to that set of programs. We have similar results when we perform the same experiment on a set of benchmarks extracted from the LLVM test-suite infrastructure and from SPEC CPU 2006.

Figure 3.2 shows the statistic distribution of the growth of our 300 larger benchmarks. We represent our data using a Turkey box plot, as described by Frigge et al. [1989]. In this kind of chart, the box represents 50% of the samples. The horizontal line inside the box represents the median. The upper and lower whiskers each represent up to 25% of the samples. The circles represent the outliers. We can observe from the first box that in very few cases the e-SSA transformation increases the number of





**Figure 3.2.** Growth on the number of instructions in comparison with SSA representation.

instructions by more than 10% and that when this increment is higher than 15% it can be considered an outlier. The second box shows that the increment in the number of instructions is inferior to 10% in most of cases. However, in some cases it can reach up to 20%.

### 3.3 Conclusion

In this chapter, we evaluate the program representations that will be used throughout the rest of this work. First, we review e-SSA, proposed by Bodik et al. [2000], that allow us to learn information from the conditional branches of the programs. The information that e-SSA allows us to extract makes it possible to almost double the precision of our range analysis, as we discuss in Section 4.4.2. Furthermore, we propose u-SSA, a new program representation that extends e-SSA and applies to programs that are free from integer overflows. u-SSA extracts information from the uses of the variables, given that a integer overflow does not occur. Those information also increase the precision of our analyses, but it narrows the scope of application to programs that have finite precision but are guaranteed to never let an integer overflow happen. Finally, we discuss the

impact of those representations on the number of instructions of the source code.

# Chapter 4

## Range Analysis

Most of the material in this chapter has been published in the paper "Speed and Precision in Range Analysis" (Campos et al. [2012]). Here we expand that first discussion, with more examples and details.

### 4.1 Background

Following Gawlitza et al. [2009]'s notation, we shall be performing arithmetic operations over the complete lattice  $\mathcal{Z} = \mathbb{Z} \cup \{-\infty, +\infty\}$ , where the ordering is naturally given by  $-\infty < \dots < -2 < -1 < 0 < 1 < 2 < \dots < +\infty$ . For any  $x > -\infty$  we define:

$$\begin{aligned} x + \infty &= \infty, x \neq -\infty & x - \infty &= -\infty, x \neq +\infty \\ x \times \infty &= \infty \text{ if } x > 0 & x \times \infty &= -\infty \text{ if } x < 0 \\ 0 \times \infty &= 0 & (-\infty) \times \infty &= \text{not defined} \end{aligned}$$

From the lattice  $\mathcal{Z}$  we define the product lattice  $\mathcal{Z}^2$ , which is defined as follows:

$$\mathcal{Z}^2 = \{\emptyset\} \cup \{[z_1, z_2] \mid z_1, z_2 \in \mathcal{Z}, z_1 \leq z_2, -\infty < z_2\}$$

This interval lattice is partially ordered by the subset relation, which we denote by " $\sqsubseteq$ ". The meet operator " $\sqcap$ " is defined by:

$$[a_1, a_2] \sqcap [b_1, b_2] = \begin{cases} [\max(a_1, b_1), \min(a_2, b_2)], & \text{if } a_1 \leq b_1 \leq a_2 \text{ or } b_1 \leq a_1 \leq b_2 \\ \emptyset, & \text{otherwise} \end{cases}$$

The join operator, " $\sqcup$ ", is given by:

$$[a_1, a_2] \sqcup [b_1, b_2] = [\min(a_1, b_1), \max(a_2, b_2)]$$

$$\begin{array}{l}
Y = [l, u] \\
Y = \phi(X_1, X_2) \\
Y = X_1 + X_2 \\
Y = X_1 \times X_2 \\
Y = aX + b \\
Y = X \sqcap [l', u']
\end{array}
\qquad
\begin{array}{l}
e(Y) = [l, u] \\
\frac{I[X_1] = [l_1, u_1] \quad I[X_2] = [l_2, u_2]}{e(Y) = [\min(l_1, l_2), \max(u_1, u_2)]} \\
\frac{I[X_1] = [l_1, u_1] \quad I[X_2] = [l_2, u_2]}{e(Y) = [l_1 + l_2, u_1 + u_2]} \\
\frac{I[X_1] = [l_1, u_1] \quad I[X_2] = [l_2, u_2] \quad L = \{l_1l_2, l_1u_2, u_1l_2, u_1u_2\}}{e(Y) = [\min(L), \max(L)]} \\
\frac{I[X] = [l, u] \quad k_l = al + b \quad k_u = au + b}{e(Y) = [\min(k_l, k_u), \max(k_l, k_u)]} \\
\frac{I[X] = [l, u]}{e(Y) \leftarrow [\max(l, l'), \min(u, u')]}
\end{array}$$

**Figure 4.1.** A suite of constraints that produce an instance of the range analysis problem.

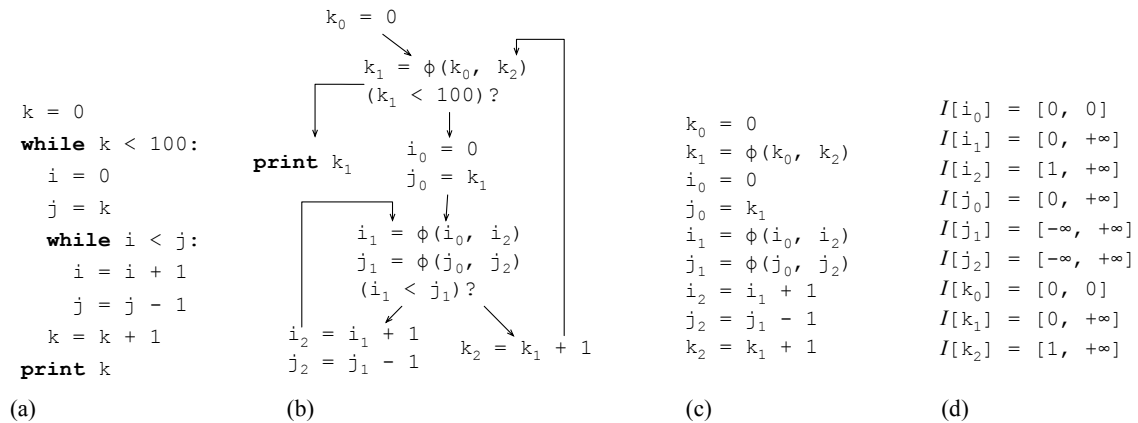
Given an interval  $\iota = [l, u]$ , we let  $\iota_\downarrow = l$ , and  $\iota_\uparrow = u$ . We let  $\mathcal{V}$  be a set of constraint variables, and  $I : \mathcal{V} \mapsto \mathcal{Z}^2$  a mapping from these variables to intervals in  $\mathcal{Z}^2$ . Our objective is to solve a constraint system  $C$ , formed by constraints such as those seen in Figure 4.1(left). We let the  $\phi$ -functions be as defined by Cytron et al. [1991]: they join different variable names into a single definition. Figure 4.1(right) defines a valuation function  $e$  on the interval domain. Armed with these concepts, we define the range analysis problem as follows:

**Definition 4.2** RANGE ANALYSIS PROBLEM

**Input:** a set  $C$  of constraints ranging over a set  $\mathcal{V}$  of variables.

**Output:** a mapping  $I$  such that, for any variable  $V \in \mathcal{V}$ ,  $e(V) = I[V]$ .

We will use the program in Figure 4.2(a) as the running example to illustrate our range analysis. Figure 4.2(b) shows the same program in SSA form (Cytron et al. [1991]), and Figure 4.2(c) outlines the constraints that we extract from this program. There is a clear correspondence between instructions and constraints. A possible solution to the range analysis problem, as obtained via the techniques that we will introduce in Section 4.3, is given in Figure 4.2(d). The SSA form, so common in modern compilers, leads to a very conservative solution. This happens because in the SSA rep-



**Figure 4.2.** (a) Example program. (b) Control Flow Graph in SSA form. (c) Constraints that we extract from the program. (d) Possible solution to the range analysis problem.

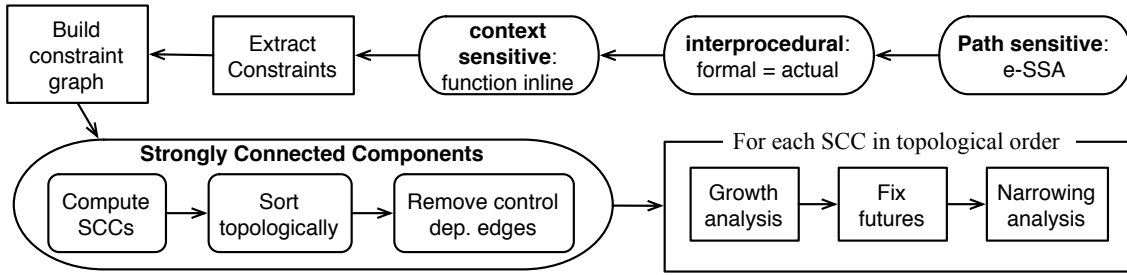
resentation we can not extract information from the conditional branches. As we will see shortly, we can improve this solution substantially by using a more sophisticated program representation – the e-SSA form – which gives us flow-sensitiveness.

Our analysis is not like classical abstract interpretation implemented in PAGAI by Henry et al. [2012], where the constraints are assigned to program points. Instead, we assign information to the variables, as a strategy to achieve sparsity in our analysis. Because SSA ensures that a variable is defined in a unique point of the program, the constraint assigned to a variable holds in every program points that the variable is alive. Thus, this association of constraints to variables is sound because we are using the SSA form and e-SSA, that also have all the properties of SSA.

## 4.3 Our Design of a Range Analysis Algorithm

In this section we explain the algorithm that we use to solve the range analysis problem. This algorithm involves a number of steps. First, we convert the program to a suitable intermediate representation that makes it easier to extract constraints. From these constraints, we build a dependence graph that allows us to do range analysis sparsely. Finally, we solve the constraints applying different fix-point iterators on this dependence graph. Figure 4.3 gives a global view of this algorithm. Some of the steps in the algorithm are optional. They improve the precision of the range analysis, at the expense of a longer running time.

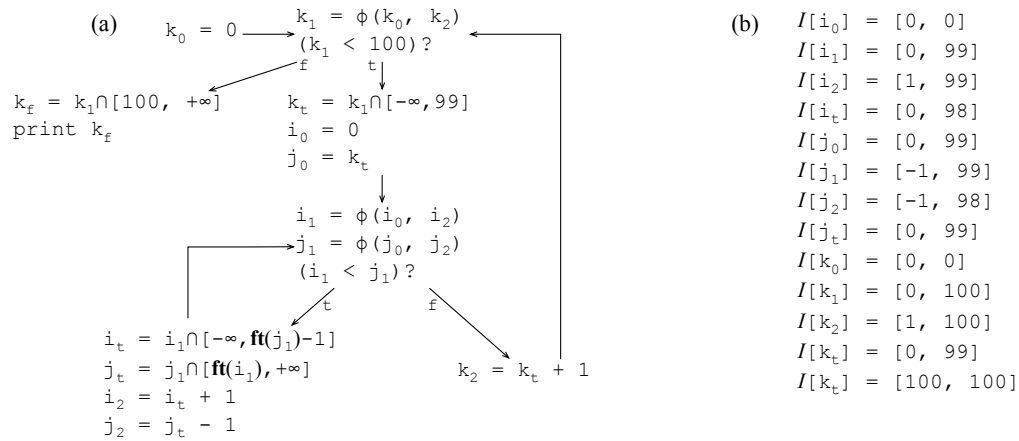
**Choosing a Program Representation.** The solution to the range analysis problem



**Figure 4.3.** Our implementation of range analysis. Rounded boxes are optional steps.

in Figure 4.2 is imprecise because we did not take conditional tests into considerations. Branches give us information about the ranges that some variables assume, but only at *specific* program points. For instance, given a test such as  $(k_1 < 100)?$  in Figure 4.2(b), we know that  $I[k_1] \sqsubseteq [-\infty, 99]$  whenever the condition is true. In order to encode this information, we might split the live range of  $k_1$  right after the branching point; thus, creating two new variables, one at the path where the condition is true, and another where it is false. There is a program representation, introduced by Bodik et al. [2000], that performs this live range splitting: the *Extended Static Single Assignment* form, or e-SSA for short.

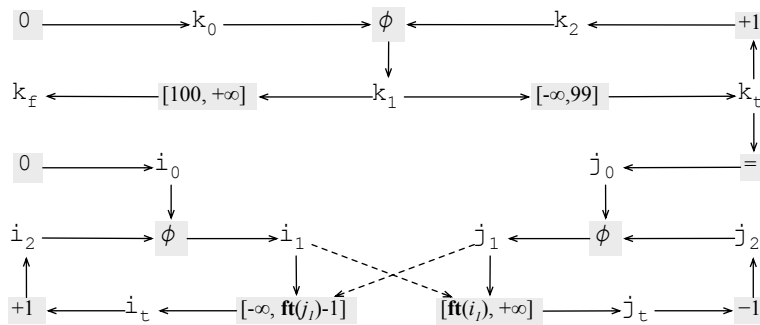
Given that the exact rules to convert a program to e-SSA form have never been explicitly stated in the literature, we describe our rules as follows. Let  $(v < c)?$  be a conditional test, and let  $l_t$  and  $l_f$  be labels in the program, such that  $l_t$  is the target of the test if the condition is true, and  $l_f$  is the target when the condition is false. We split the live range of  $v$  at any of these points if at least one of two conditions is true: (i)  $l_f$  or  $l_t$  dominate any use of  $v$ ; (ii) there exist a use of  $v$  at the dominance frontier of  $l_f$  or  $l_t$ . For the notions of dominance and dominance-frontier, see [Aho et al., 2006, p.656]. To split the live range of  $v$  at  $l_f$  we insert at this program point a copy  $v_f = v \sqcap [c, +\infty]$ , where  $v_f$  is a fresh name. We then rename every use of  $v$  that is dominated by  $l_f$  to  $v_f$ . Dually, if we must split at  $l_t$ , then we create at this point a copy  $v_t = v \sqcap [-\infty, c - 1]$ , and rename variables accordingly. If the conditional uses two variables, e.g.,  $(v_1 < v_2)?$ , then we create intersections bound to *futures*. We insert, at  $l_f$ ,  $v_{1f} = v_1 \sqcap [\mathbf{ft}(v_2), +\infty]$ , and  $v_{2f} = v_2 \sqcap [-\infty, \mathbf{ft}(v_1)]$ . Similarly, at  $l_t$  we insert  $v_{1t} = v_1 \sqcap [-\infty, \mathbf{ft}(v_2) - 1]$  and  $v_{2t} = v_2 \sqcap [\mathbf{ft}(v_1) + 1, +\infty]$ . Notice that a variable  $v$  can never be associated with a future bound to itself, e.g.,  $\mathbf{ft}(v)$ . This invariant holds because whenever the e-SSA conversion associates a variable  $u$  with  $\mathbf{ft}(v)$ , then  $u$  is a fresh name created to split the live range of  $v$ , given that  $v$  was used in a conditional.



**Figure 4.4.** (a) The control flow graph from Figure 4.2(b) converted to standard e-SSA form. (b) A solution to the range analysis problem

We use the notation  $\mathbf{ft}(v)$  to denote the *future* bounds of a variable. As we will show in Section 4.3.1, once the growth pattern of  $v$  is known, we can replace  $\mathbf{ft}(v)$  by an actual value. After splitting the live ranges according to the rules stated above, we might have to insert  $\phi$ -functions into the transformed program to re-convert it to SSA form. This last step avoids that two different names given to the same original variable be simultaneously *alive* at the program code. A variable  $v$  is alive at a program point  $p$  if the program's control flow graph contains a path from  $p$  to a site where  $v$  is used, that does not go across any re-definition of  $v$ . Figure 4.4(a) shows our running example changed into standard e-SSA form. We have not created variable names for  $i_f$  and  $j_f$ , because neither  $i_1$  nor  $j_1$ , the variables that have been renamed, are dominated by the target of the conditional's else case. In this example, new  $\phi$ -functions are not necessary: new variable names are not alive together with the original variables. The part (b) of this figure shows the solution that we get to this new program. The e-SSA form allows us to bind interval information directly to the live ranges of variables; thus, giving us the opportunity to solve range analysis sparsely. More traditional approaches, which we call *dense analyses*, bind interval information to pairs formed by variables and program points.

**Extracting Constraints.** Our implementation handles 18 different assembly instructions. The constraints in Figure 4.1 show only a few examples. Instructions that we did not show include, for instance, the multiplicative operators `div` and `modulus`, the bitwise operators `and`, `or`, `xor` and `neg`, the different types of shifts, and the logical operators `andalso`, `orelse` and `not`. Most of these instructions are sign-agnostic; that



**Figure 4.5.** The dependence graph that we build to the program in Figure 4.4.

is, given that numbers are internally represented in 2's complement, the same implementation of a constraint handles positive and negative numbers. However, there are instructions that require different constraints, depending on the input being signed or not. Examples include `modulus` and `div`. We also handle different kinds of type conversion, e.g., converting 8-bit characters to 32-bit integers and vice-versa. In addition to constraints that represent actual assembly instructions, we have constraints to represent  $\phi$ -functions, and intersections, as seen in Figure 4.1. The growth analysis that we will introduce in Section 4.3.1 require monotonic transfer functions. Many assembly operations, such as modulus or division, do not afford us this monotonicity. However, these non-monotonic instructions have conservative approximations, as shown by Warren [2002].

**The Constraint Graph.** The main data structure that we use to solve the range analysis problem is a variation of the *program dependence graph* of Ferrante et al. [1987]. For each constraint variable  $V$  we create a variable node  $N_v$ . For each constraint  $C$  we create a constraint node  $N_c$ . We add an edge from  $N_v$  to  $N_c$  if the name  $V$  is used in  $C$ . We add an edge from  $N_c$  to  $N_v$  if the constraint  $C$  defines the name  $V$ . Figure 4.5 shows the dependence graph that we build for the e-SSA form program given in Figure 4.4(a). If  $V$  is used by  $C$  as the input of a future, then the edge from  $N_v$  to  $N_c$  represents what Ferrante *et al.* call a *control dependence* [Ferrante et al., 1987, p.323]. We use dashed lines to represent these edges. All the other edges denote *data dependencies* [Ferrante et al., 1987, p.322].

**Strongly Connected Components.** In order to solve range analysis we find all the strongly connected components (SCCs) of the dependence graph with Nuutila and Soisalon-Soininen [1994]'s algorithm and collapse them into single nodes, obtaining a directed acyclic graph. We then sort the resulting DAG topologically, and apply the



analyses from Section 4.3.1 on every SCC in topological order. Once we solve the range analysis problem for a SCC, we propagate the intervals that we found to the variable nodes at the *frontier* of this SCC. A variable node  $N_v$  is said to be in the frontier of a strongly connected component  $S$  if: (i)  $N_v \notin S$ ; and (ii) there exists a variable node  $N'_v \in S$ , and a constraint node  $N_c$ , such that  $N_v \leftarrow N_c$ , and  $N_c \leftarrow N'_v$ . This propagation ensures that when analyzing a strongly connected component  $S$  any influence that  $S$  might suffer from nodes outside it has already been considered.

When finding strongly connected components, we take control dependence edges into consideration. For instance, in Figure 4.5 the nodes that correspond to the variables  $i_1, i_2, i_t, j_1, j_2$  and  $j_t$  form a single component. The dashed edges, which represent control dependencies, keep all these variables connected. In this way, we ensure that, upon stumbling upon an interval associated with future bounds, e.g.,  $\mathbf{ft}(v)$ , either variable  $v$  has been solved in a previous component, or it belongs to the current component. In the latter case, as we will see soon, we can still take  $v$ 's interval into consideration. This flexibility is possible because we first discover how each variable in a strong component grows, before resolving future bounds. As we show in Section 4.3.2, most of the strong components in actual programs are singletons. Furthermore, the composite components tend to be small. These two facts ensure that the more costly parts of our algorithm only have to handle small inputs.

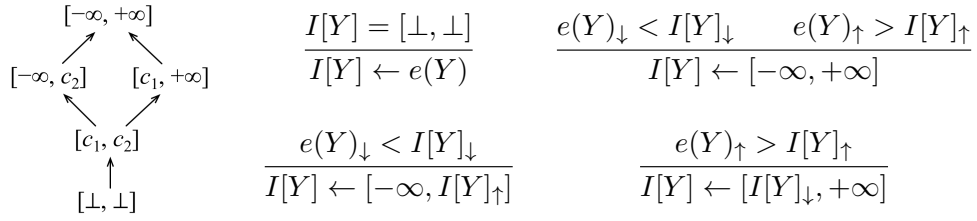
### 4.3.1 Finding Ranges in Strongly Connected Components

Given a strongly connected component of the dependence graph with  $N$  nodes, we solve the range analysis problem in three-steps:

1. Run growth analysis:  $O(N)$ .
2. Fix intersections:  $O(N)$ .
3. Apply the narrowing operator:  $O(N^2)$ .

However, before we start, we remove control dependence edges from the strongly connected component, as they have no semantics to our transfer functions.

**Growth Analysis.** The first step of our algorithm consists in determining how the interval bound to each variable grows. The possible behaviors of an interval are: (i) does not change; (ii) grows towards  $+\infty$ ; (iii) grows towards  $-\infty$ ; and (iv) grows in both directions. We ensure termination of this phase via a *widening operator*. We have experimented with four different widening strategies, which we discuss in Section 4.4.5. One of these strategies is based on the widening operator of [Cousot and Cousot,



**Figure 4.6.** (Left) The lattice of the growth analysis. (Right) Cousot and Cousot's widening operator. We evaluate the rules from left-to-right, top-to-bottom, and stop upon finding a pattern matching. Again: given an interval  $\iota = [l, u]$ , we let  $\iota_{\downarrow} = l$ , and  $\iota_{\uparrow} = u$

$$\frac{Y = X \sqcap [l, \mathbf{ft}(V) + c] \quad I[V]_{\uparrow} = u}{Y = X \sqcap [l, u + c]} \quad u, c \in \mathbb{Z} \cup \{-\infty, +\infty\}$$

$$\frac{Y = X \sqcap [\mathbf{ft}(V) + c, u] \quad I[V]_{\downarrow} = l}{Y = X \sqcap [l + c, u]} \quad l, c \in \mathbb{Z} \cup \{-\infty, +\infty\}$$

**Figure 4.7.** Rules to replace futures by actual bounds.

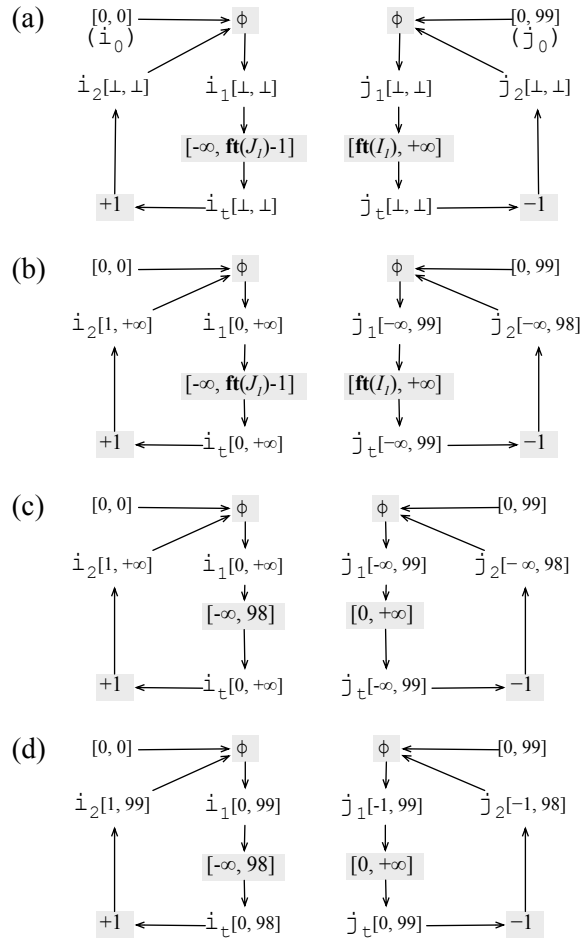
$$\frac{I[Y]_{\downarrow} = -\infty \quad e(Y)_{\downarrow} > -\infty}{I[Y] \leftarrow [e(Y)_{\downarrow}, I[Y]_{\uparrow}]} \quad \frac{I[Y]_{\downarrow} > e(Y)_{\downarrow}}{I[Y] \leftarrow [e(Y)_{\downarrow}, I[Y]_{\uparrow}]}$$

$$\frac{I[Y]_{\uparrow} = +\infty \quad e(Y)_{\uparrow} < +\infty}{I[Y] \leftarrow [I[Y]_{\downarrow}, e(Y)_{\uparrow}]} \quad \frac{I[Y]_{\uparrow} < e(Y)_{\uparrow}}{I[Y] \leftarrow [I[Y]_{\downarrow}, e(Y)_{\uparrow}]}$$

**Figure 4.8.** Cousot and Cousot's narrowing operator.

1977, p.247]. The lattice of abstract states, plus a constraint system representing this operator is given in Figure 4.6. Because the lattice has height three, the intervals bound to each variable can change at most three times.

**Fixing futures.** The ranges found by the growth analysis tells us which variables have fixed bounds, independent on the intersections in the constraint system. Thus, we can use actual limits to replace intersections bounded by futures. Figure 4.7 shows the rules to perform these substitutions. In order to correctly replace a future  $\mathbf{ft}(v)$  that limits a variable  $v'$ , we need to have already applied the growth analysis onto  $v$ . Had we considered only data dependence edges, then it would be possible that  $N_{v'}$  be analyzed before  $N_v$ . However, because of control dependence edges, this case cannot happen.



**Figure 4.9.** Four snapshots of the last SCC of Figure 4.4. (a) After removing control dependence edges. (b) After running the growth analysis. (c) After fixing the intersections bound to futures. (d) After running the narrowing analysis.

The control dependence edges ensure that any topological ordering of the constraint graph either places  $N_v$  before  $N_{v'}$ , or places these nodes in the same strongly connected component. For instance, in Figure 4.5, variables  $j_1$  and  $i_t$  are in the same SCC only because of the control dependence edges.

**Narrowing Analysis.** The growth analysis associates very conservative bounds to each variable. Thus, the last step of our algorithm consists in narrowing these intervals. We accomplish this step via the classic narrowing operator of [Cousot and Cousot, 1977, p.248], which we show in Figure 4.8.

**Example:** Continuing with our example, Figure 4.9 shows the application of our algorithm on the last strong component of Figure 4.5. We are not guaranteed to find the least fixed point of a constraint system. However, in this example we did it.

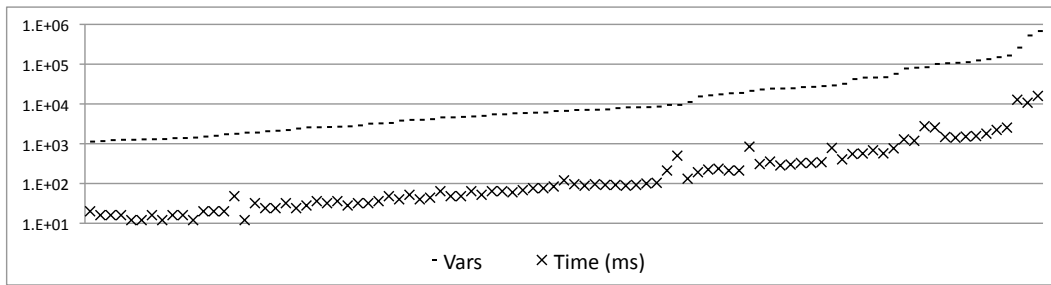
Figure 4.4(b) shows our final solution for this example. This solution is very precise, in the sense that it is the least fixed point of the constraint system given in Figure 4.1. However, the solution is still an over approximation of the dynamic behavior of the program in Figure 4.2(a). For instance, we have found that variable  $i$  could reach the upper value of 99. In any actual run of the program,  $i$  could be at most 50. Analyses on relational lattices, such as the polyhedron (Cousot and Halbwachs [1978]) or the Octagon (Miné [2006]) domains, can infer such tighter bounds, as shown by Lakhdar-Chaouch et al. [2011]. However, analyses on these higher dimensional domains are much more computationally expensive than analyses on the interval domain, as shown by Oh et al. [2012].

We emphasize that finding this tight solution was only possible because of the topological ordering of the constraint graph in Figure 4.5. Upon meeting the constraint graph’s last SCC, shown in Figure 4.9, we had already determined that the interval  $[0, 0]$  is bound to  $i_0$  and that the interval  $[0, 99]$  is bound to  $j_0$ , as we show in Figure 4.9(a). Had we applied the widening operator onto the whole graph, then we would have found out that variable  $j_1$  is bound to  $[-\infty, +\infty]$ . This imprecision happens because, on one hand  $j_1$ ’s interval is influenced by  $k_t$ ’s, which is upper bounded by  $+\infty$ . On the other hand  $j_1$  is part of a decreasing cycle of dependencies formed by variables  $j_t$  and  $j_2$  in addition to itself. Therefore, if we had not computed the strongly connected components and, consequently, had applied the widening phase over the entire program followed by a global narrowing phase, then we would not be able to recover some of widening’s precision loss. However, because in this example we only analyze  $j$ ’s SCC after we have analyzed  $k$ ’s,  $k$  only contribute the constant range  $[0, 99]$  to  $j_0$ .

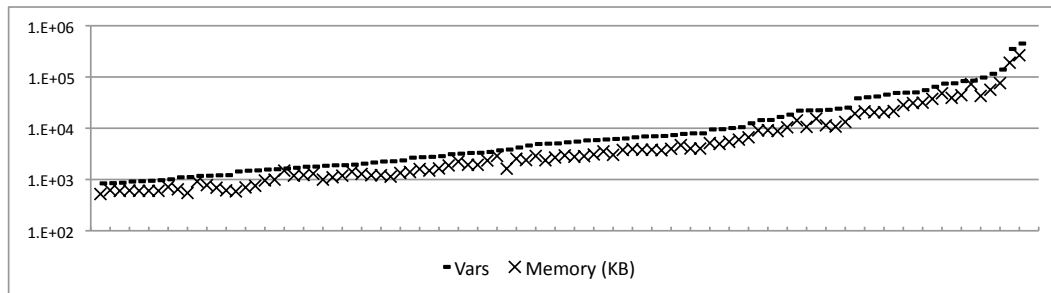
### 4.3.2 Experiments

The objective of this section is to show, via experimental numbers, that our implementation of range analysis is efficient and effective. We have used it to analyze a test suite with 2.72 million lines of C code, which includes, in addition to all the benchmarks distributed with LLVM, the programs in the SPEC CPU 2006 collection.

**Time and Memory Complexity.** Figure 4.10 provides a visual comparison between the time to run our algorithm and the size of the input programs. We show data for the 100 largest benchmarks in our test suite, in number of variable nodes in the constraint graph. We perform function inlining before running our analysis. Each point in the X line corresponds to a benchmark. We analyze the smallest benchmark in this set, `Prolangs-C/deriv2`, which has 1,131 variable nodes in the constraint graph, in 20ms. We take 15.91 sec to analyze our largest benchmark, `403.gcc`, which, after



**Figure 4.10.** Correlation between program size (number of var nodes in constraint graphs after inlining) and analysis runtime (ms). Coefficient of determination = 0.967.

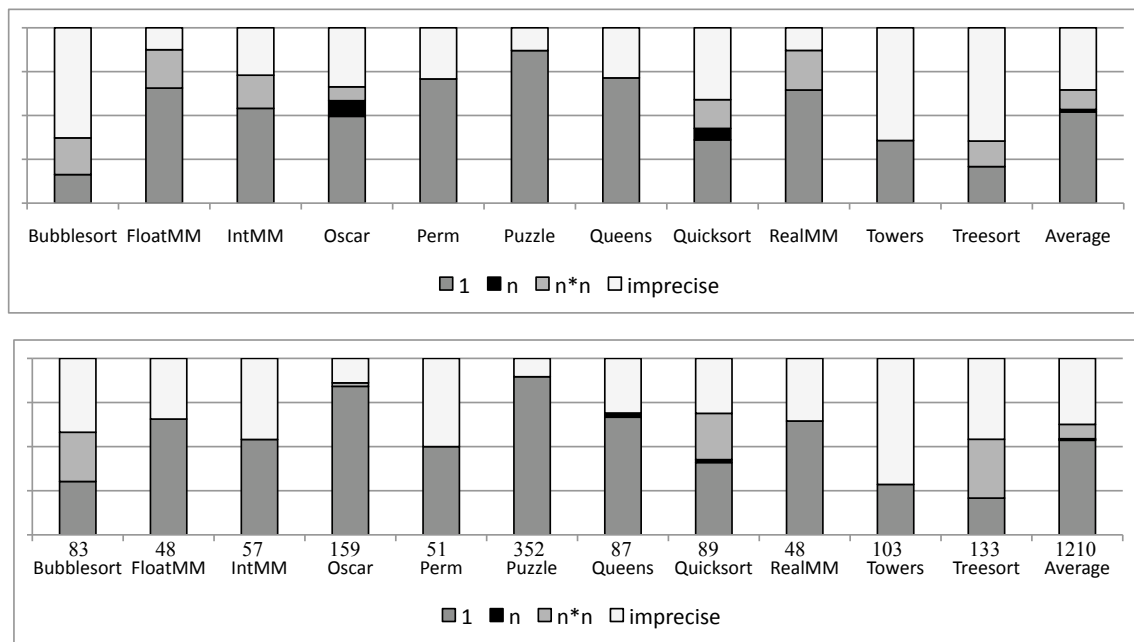


**Figure 4.11.** Comparison between program size (number of var nodes in constraint graphs) and memory consumption (KB). Coefficient of determination = 0.9947.

function inlining, has 1,266,273 assembly instructions, and a constraint graph with 679,652 variable nodes. For this data set, the coefficient of determination ( $R^2$ ) is 0.967, which provides very strong evidence about the linear asymptotic complexity of our implementation.

The experiments also reveal that the memory consumption of our implementation is linear with the program size. Figure 4.11 plots these two quantities together. The linear correlation, in this case, is even stronger than that found in Figure 4.10, which compares runtime and program size: the coefficient of determination is 0.9947. The figure only shows our 100 largest benchmarks. Again, SPEC 403.gcc is the heaviest benchmark, requiring 265,588KB to run. Memory includes stack, heap and the executable program code.

**Precision.** Our implementation of range analysis is remarkably precise, considering its runtime. The relational analysis of Lakhdar-Chaouch et al. [2011], for instance, takes about 25 minutes to go over a program with almost 900 basic blocks. We analyze programs of similar size in less than one second. We do not claim our approach is



**Figure 4.12.** (Upper) Comparison between static range analysis and dynamic profiler for upper bounds. (Lower) Comparison between static range analysis and dynamic profiler for lower bounds. The numbers above the benchmark names give the number of variables in each program.

as precise as such algorithms, even though we are able to find exact bounds to 4/5 of the examples presented in Lakhdar-Chaouch et al. [2011]. On the contrary, this paper presents a compromise between precision and speed that scales to very large programs. Nevertheless, our results are far from being trivial. We have implemented a dynamic profiler that measures, for each variable, its upper and lower limits, given an execution of the target program. Figure 4.12 compares our results with those measured dynamically for the Stanford benchmark suite, which is publicly available<sup>1</sup>. We chose Stanford because these benchmarks do not read data from external files; hence, imprecisions are due to library functions that we cannot analyze.

We have classified the bounds estimated by the static analysis into four categories. The first category, which we call 1, contains those bounds that are tight: during the execution of the program, the variable has been assigned an upper, or lower limit, that equals the limit inferred statically. The second category, which we call  $n$ , contains the bounds that are within twice the value inferred statically. For instance, if the range analysis estimates that a variable  $v$  is in the range  $[0, 100]$ , and during the execution the dynamic profiler finds that its maximum value is 51, then  $v$  falls into this category.

<sup>1</sup><http://classes.engineering.wustl.edu/cse465/docs/BCCExamples/stanford.c>

The third category,  $n^2$ , contains variables whose actual value is within a quadratic factor from the estimated value. In our example,  $v$ 's upper bound would have to be at most 10 for it to be in this category. Finally, the fourth category contains variables whose estimated value lays outside a quadratic factor of the actual value. We call this category *imprecise*, and it contains mostly the limits that our static analysis has marked as either  $+\infty$  or  $-\infty$ .

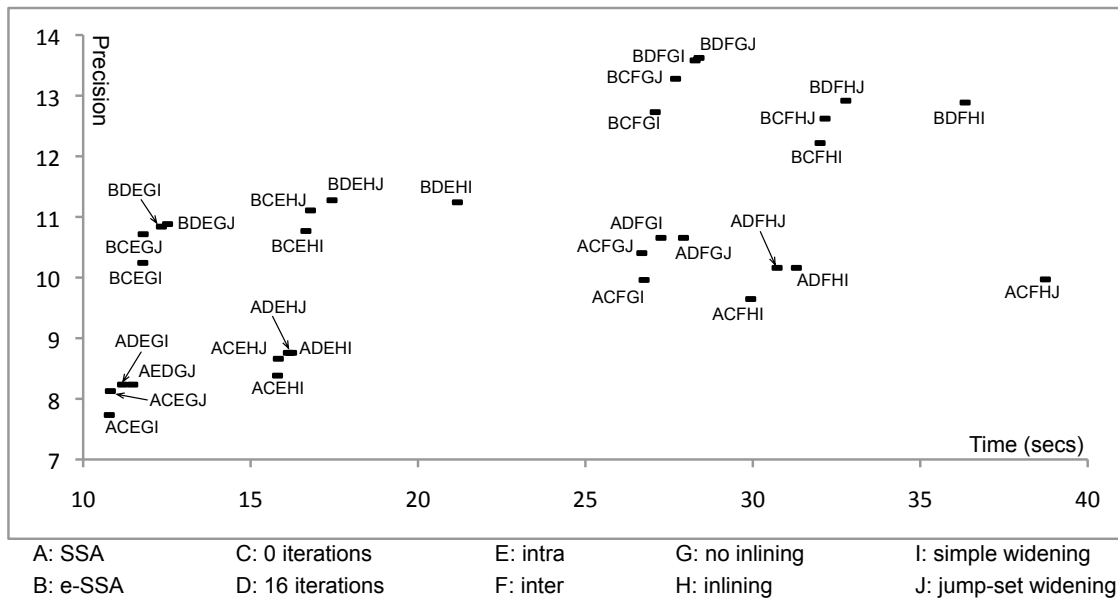
As we see in Figure 4.12, 54.11% of the lower limits that we have estimated statically are exact. Similarly, 51.99% of our upper bounds are also tight. The figure also shows that, on average, 37.39% of our lower limits are imprecise, and 35.40% of our upper limits are imprecise. This result is on par with those obtained by more costly analysis, such as that of Stephenson et al. [2000]. However, whereas that approach has only been tested on single functions, we have been able to deal with remarkably larger programs.

## 4.4 Design Space

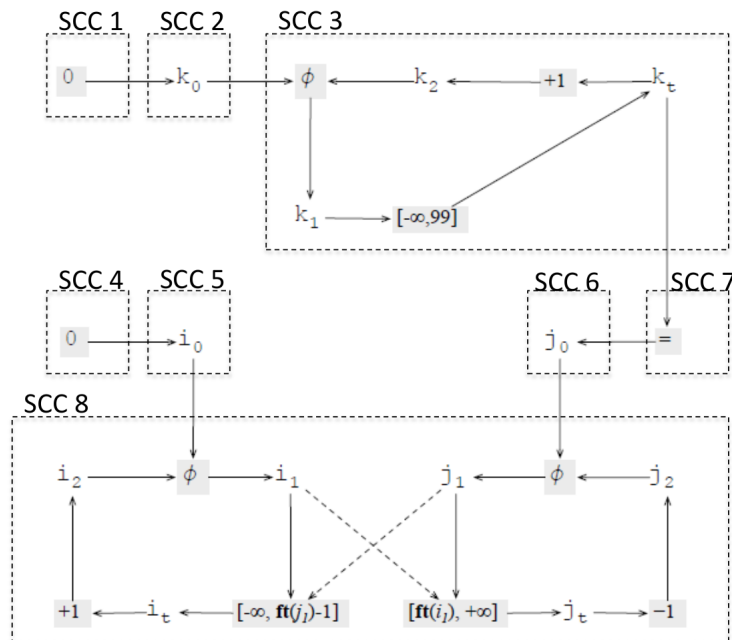
As we see from a cursory glance at Figure 4.3, our range analysis algorithm has many optional modules. These modules give the user the chance to choose between more precise results, or a faster analysis. Given the number of options, the design space of a range analysis algorithm is vast. In this section we try to cover some of the most important trade-offs. Figure 4.13 plots, for the integer programs in the SPEC CPU 2006 benchmark suite, precision versus speed for different configurations of our implementation. Our initial goal when developing this analysis was to support a bitwidth-aware register allocator. Thus, we measure precision by the average number of bits that our analysis allows us to save per program variable. It is very important to notice that we do not consider constants in our statistics of precision. In other words, we only measure bitwidth reduction in variables that a constant propagation step could not remove.

### 4.4.1 Strongly Connected Components

The greatest source of improvement in our implementation is the use of strongly connected components (SCCs). Figure 4.14 shows the SCCs extracted from our motivating example. In order to propagate ranges across the constraint graph, we fragment it into strongly connected components, collapse each of these components into single nodes, and sort the resulting directed acyclic graph topologically. We then solve the range analysis problem for each component individually. Once we have solved a component,

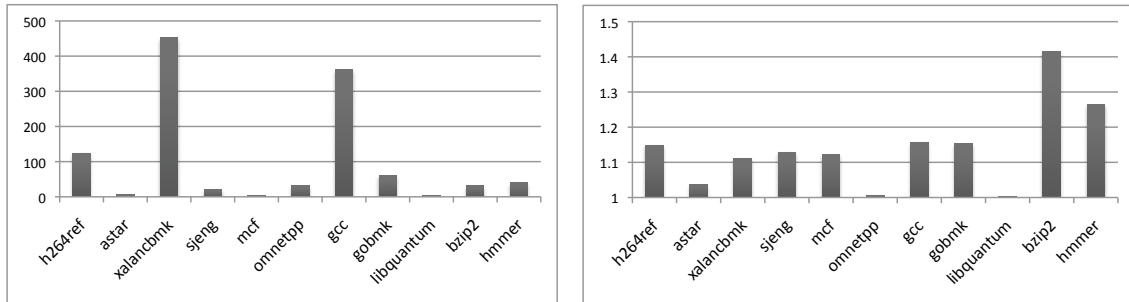


**Figure 4.13.** Design space exploration: precision (percentage of bitwidth reduction) versus speed (secs) for different configurations of our algorithm analyzing the SPEC CPU 2006 integer benchmarks.



**Figure 4.14.** Strongly Connected Components extracted from our example program.





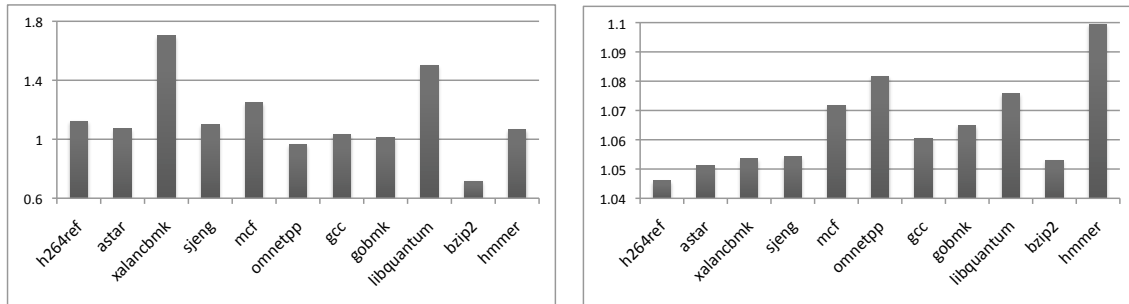
**Figure 4.15.** (Left) Time to run our analysis without building strong components divided by time to run the analysis on strongly connected components. (Right) Precision, in bitwidth reduction, that we obtain with strong components, divided by the precision that we obtain without them.

we propagate its ranges to the next components, and repeat the process until we walk over the entire constraint graph. It is well-known that this technique is essential to speedup constraint solving algorithms [Nielson et al., 1999, Sec 6.3]. In our case, the results are significant, mostly in terms of speed, but also in terms of precision. Figure 4.15 shows the speedup that we gain by using strong components. We show results for the integer programs in the SPEC CPU 2006 benchmark suite. In some cases, as in `xalancbmk` the analysis on strong components is 450x faster.

The strong components improve the precision of our growth analysis. According to Figure 4.15, in some cases, as in `bzip2`, strong components increase our precision by 40%. The gains in precision happen because, by completely resolving a component, we are able to propagate constant intervals to the next components, instead of propagating intervals that can grow in both directions. As an example, in Figure 4.9 we pass the range  $[0, 99]$  from variable  $k$  to the component that contains variable  $j$ . Had we run the analysis in the entire constraint graph, by the time we applied the growth analysis on  $j$  we would still find  $k$  bound to  $0, +\infty$ .

#### 4.4.2 The Choice of a Program Representation

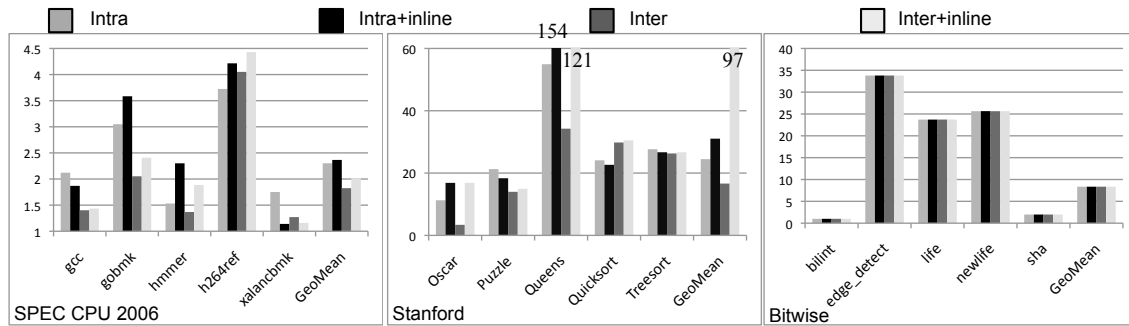
If strong components account for the largest gains in speed, the choice of a suitable program representation is responsible for the largest gains in precision. However, here we no longer have a win-win condition: a more expressive program representation decreases our speed, because it increases the size of the target program. We have tried our analysis in two different program representations: the Static Single Assignment (SSA) Form (Cytron et al. [1991]), and the Extended Static Single Assignment (e-



**Figure 4.16.** (Left) Bars give the time to run analysis on e-SSA form programs divided by the time to run analysis on SSA form programs. (Right) Bars give the size of the e-SSA form program, in number of assembly instructions, divided by the size of the SSA form program.

SSA) form (Bodik et al. [2000]). The SSA form gives us a faster, albeit more imprecise, analysis. Any program in e-SSA form has also the SSA core property: any variable name has at most one definition site. The contrary is not true: SSA form programs do not have the core e-SSA property: any use site of a variable that appears in a conditional test post-dominates its definition. The program in Figure 4.2(b) is in e-SSA form. The live ranges of variables  $i_1$  and  $j_1$  have been split right after the conditional test via the assertions that creates variables  $i_t$  and  $j_t$ . The e-SSA format serves well analyses that extract information from definition sites and conditional tests, and propagate this information forwardly. Examples include, in addition to range analysis, tainted flow analysis (Rimsa et al. [2011]) and array bounds checks elimination (Bodik et al. [2000]).

Figure 4.16 compares these two program representations in terms of runtime. As we see in Figure 4.16(Left), the e-SSA form slows down our analysis. In some cases, as in `xalancbmk`, this slowdown increases execution time by 71%. Runtime increases for two reasons. Firstly, the e-SSA form programs are larger than the SSA form programs, as we show in Figure 4.16(Right). However, this growth is small: in none of the integer programs in SPEC CPU 2006 we verified an increase in code size of more than 9%. Secondly, the e-SSA form program has futures; hence requiring the future resolution phase of our algorithm, which is not necessary in SSA form programs. Nevertheless, if the e-SSA form slows down the analysis runtime, its gains in precision are remarkable, as seen in Figure 4.17. These gains happen because the e-SSA format lets the analysis to use the results of conditional tests to narrow the ranges of variables.



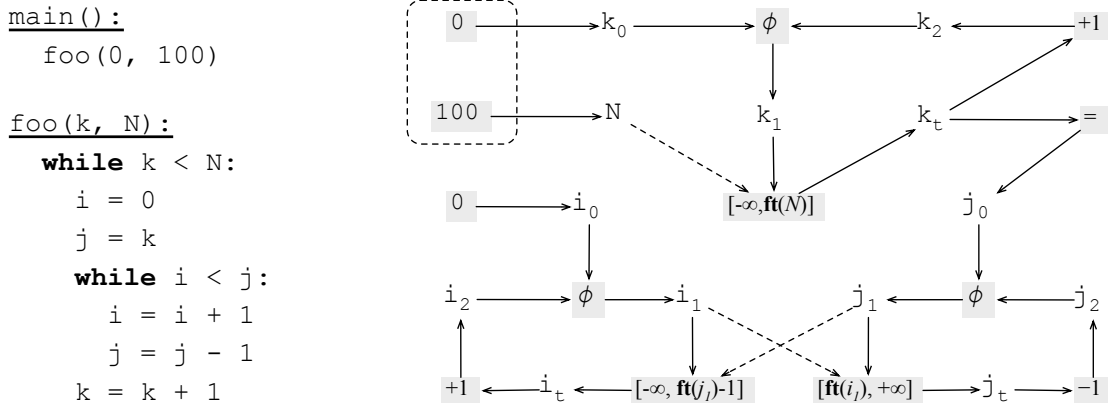
**Figure 4.17.** The impact of the e-SSA transformation on precision for three different benchmark suites. Bars give the ratio of precision (in bitwidth reduction), obtained with e-SSA form conversion divided by precision without e-SSA form conversion.

### 4.4.3 Intra versus Inter-procedural Analysis

A naive implementation of range analysis would be intra-procedural; that is, would solve the range analysis problem once per each function. However, we can gain in precision by performing it inter-procedurally. An inter-procedural implementation allows the results found for a function  $f$  to flow into other functions that  $f$  calls. Figure 4.18 illustrates the inter-procedural analysis for the program seen in Figure 4.2(a). The trivial way to produce an inter-procedural implementation is to insert into the constraint system assignments from the actual parameter names to the formal parameter names. In our example of Figure 4.18, our constraint graph contains a flow of information from 0, the actual parameter, to  $k_0$ , the formal parameter of function `foo`.

Figure 4.20 compares the precision of the intra and inter-procedural analyses for the five largest programs in three different categories of benchmarks: SPEC CPU 2006, the Stanford Suite <sup>2</sup> and Bitwise (Stephenson et al. [2000]). Our results for the SPEC programs were disappointing: on the average for the five largest programs, the intra-procedural version of our analysis saves 5.23% of bits per variable. The inter-procedural version increases this number to 8.89%. A manual inspection of the SPEC programs reveals that this result is expected: these programs manipulate files, and their source codes do not provide enough explicit constants to power our analysis up. However, with numerical benchmarks we fare much better. On the average our inter-procedural algorithm reduces the bitwidth of the Stanford benchmarks by 36.24%. For Bitwise we obtain a bitwidth reduction of 12.27%. However, this average is lowered by two outliers: `edge_detect` and `sha`, which have been purposely engineered to be resilient against

<sup>2</sup><http://classes.engineering.wustl.edu/cse465/docs/BCCEXamples/stanford.c>



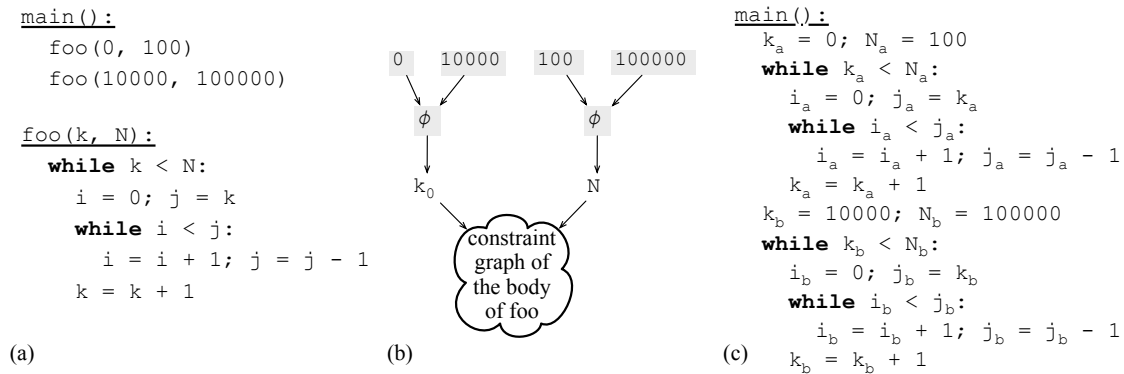
**Figure 4.18.** Example where an intra-procedural implementation would lead to imprecise results.

range analyses (Stephenson et al. [2000]). The bitwise benchmarks were implemented by Stephenson et al. [2000] to validate their intra-procedural bitwidth analysis. Our results are on par with those found by the original authors. The bitwise programs contain only the `main` function; thus, different versions of our algorithm find the same results when applied onto these programs.

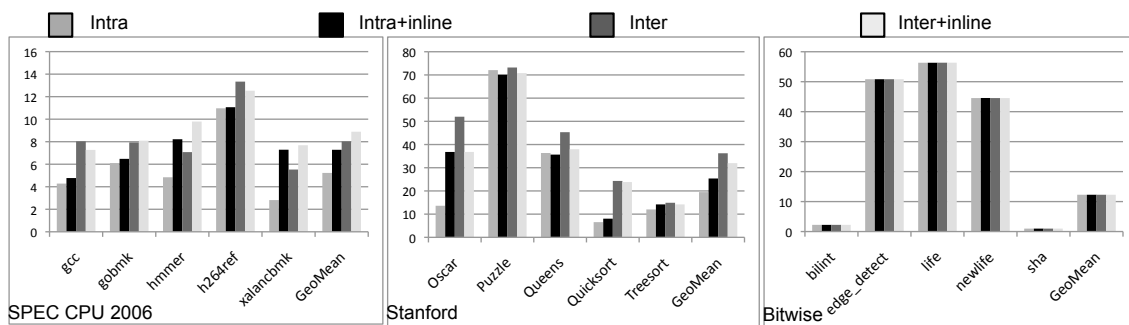
#### 4.4.4 Achieving Partial Context-Sensitiveness via Function Inlining

Another way to increase the precision of range analysis is via a context-sensitive implementation. Context-sensitiveness allows us to distinguish different calling sites of the same function. Figure 4.19 shows why the ability to make this distinction is important for precision. In Figure 4.19(a) we have two different calls of function `foo`. If we apply the trivial inter-procedural approach of Section 4.4.3, then we get the graph shown in Figure 4.19(b). In other words, if a function is called more than once, then its formal parameters will receive information from many actual parameters. We use  $\phi$ -functions to bind this information together into a single flow. However, in this case the multiple assignment of values to parameters makes the ranges of these parameters very large, whereas in reality they are not. A way to circumvent this source of imprecision is via function inlining, as we show in Figure 4.19(c). The results that we can derive for the transformed program are more precise, as each input parameter is assigned a single value.

Figure 4.20 also shows how function inlining modifies the precision of our results.



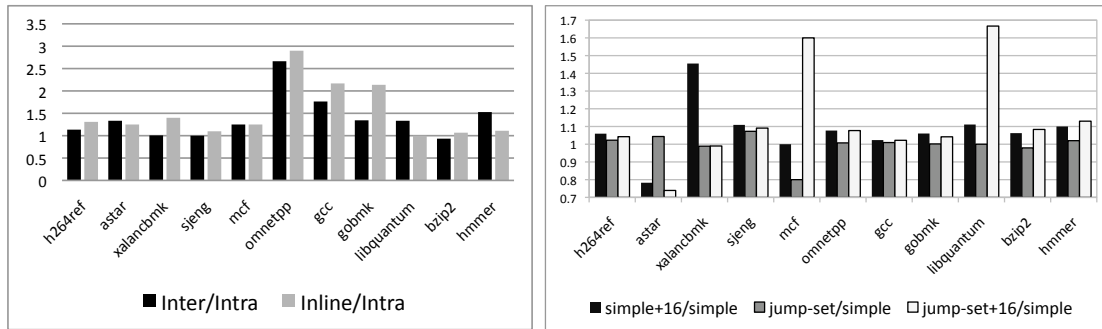
**Figure 4.19.** Example where a context-sensitive implementation improves the results of range analysis.



**Figure 4.20.** The impact of inter-procedural analysis on precision. Each bar shows precision in %bitwidth reduction.

It is difficult to find an adequate way to compare the precision of our analysis with, and without inlining. This difficulty stems from the fact that this transformation tends to change the target program too much. In absolute numbers, we always reduce the bitwidth of more variables after function inlining. However, proportionally function inlining leads to a smaller percentage of bitwidth reduction for many benchmarks. In the Stanford Collection, for instance, where most of the functions are called in only one location, inlining leads to worse precision results. On the other hand, for the SPEC programs, inlining, even in terms of percentage of reduction, tends to increase our measure of precision.

**Intra vs Inter-procedural runtimes.** Figure 4.21(Left) compares three different execution modes. Bars are normalized to the time to run the intra-procedural analysis without inlining. On the average, the intra-procedural mode is 28.92% faster than



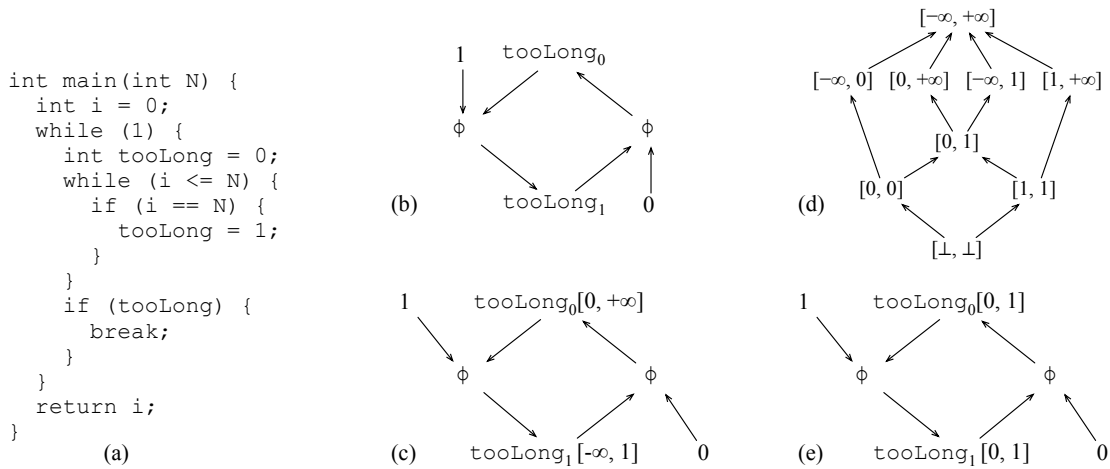
**Figure 4.21.** (Left) Runtime comparison between intra, inter and inter+inline versions of our algorithm. (Right) Runtime comparison between different widening operators. The bars are normalized to the time to run the intra-procedural analysis.

the inter-procedural mode. If we perform function inlining, then this difference is 45.87%. These numbers are close because our runtime is bound to the size of the strong components. Although function inlining can increase the number of strongly connected components in the constraint graph, it cannot increase the size of the largest component, when compared to the simple inter-procedural analysis of Section 4.4.3.

#### 4.4.5 Choosing a Widening Strategy

We have implemented the widening operator used in the growth analysis in two different ways. The first way, which we call *simple*, is based on the original widening operator of Cousot and Cousot [1977], and we have shown it in Figure 4.6(Right). The second widening strategy, which we call *jump-set widening* consists in using the constants that appear in the program text, in sorted order, as the next limits of each interval after widening is applied. The *jump-set widening* can be seen as kind of *Widening "up to"* (Halbwachs et al. [1997]), because it also delays the widening operation. This operator is common in implementations of range analysis [Nielson et al., 1999, p.228]. Jump-set widening never produces worse results than the simple operator, and sometimes it does better. Figure 4.22 shows an example taken from the code of SPEC CPU *bzip2*. Part of the constraint graph of the program in Figure 4.22(a) is given in Figure 4.22(b). The result of applying the simple operator is shown in Figure 4.22(c). Jump-set widening would use the lattice in Figure 4.22(d), instead of the lattice in Figure 4.6(Left). This lattice yields the result given in Figure 4.22(e), which is more precise.

Another way to improve the precision of growth analysis is to perform a few rounds of abstract interpretation on the constraint graph, and, in case the process



**Figure 4.22.** An example where jump-set widening is more precise.

Benchmark	Size	0 + Simple	16 + Simple	0 + Jump	16 + Jump
hmmer	38,409	9.98	11.40 (12.45)	10.98 (9.11)	11.40 (12.45)
gobmk	84,846	8.15	9.93 (17.92)	9.02 (9.64)	10.13 (19.54)
h264ref	97,494	12.58	13.11 (4.04)	13.00 (3.23)	13.11 (4.04)
xalancbmk	352,423	7.71	7.98 (3.38)	7.95 (3.02)	7.98 (3.38)
gcc	449,442	16.09	16.63 (3.25)	16.41 (1.95)	16.64 (3.31)

**Table 4.1.** Variation in the precision of our analysis given the widening strategy. The size of each benchmark is given in number of variable nodes in the constraint graph. Precision is given in percentage of bitwidth reduction. Numbers in parenthesis are percentage of gain over 0 + Simple.

does not reach a fixed point, only then to apply the widening operator. Each round of abstract interpretation consists in evaluating all the constraints, and then updating the intervals that change from one evaluation to the other. For instance, in Figure 4.22 one round of abstract interpretation, coupled with the simple widening operator, would be enough to reach the fixed point of that constraint system. We have experimented with 0 and 16 iterations before doing widening, and the overall result, for the programs in the SPEC CPU 2006 suite is given in Figure 4.13. Table 4.1 shows some of these results in more details for the five largest benchmarks in this collection. In general jump-set widening improves the precision of our results in non-trivial ways. Nevertheless, the simple widening operator preceded by 16 rounds of abstract interpretation in general is more precise than jump-set widening without any cycle of pre-evaluation, as we see in Table 4.1.

By combining the different widening operators – simple or jump-set – with the different number of pre-evaluations – in our case 0 or 16, we have four different widening strategies. Figure 4.21(Right) compares the runtime of all these strategies for the integer programs in the SPEC CPU 2006 collection. We have observed no measurable difference between the jump-set and the simple operator: the latter is 0.93% faster, e.g., 1.0093x faster. The strategy that precedes the simple operator with 16 rounds of pre-evaluation is 7% slower than the strategy that does not do any pre-evaluation. Finally, the combination of 16 rounds of pre-evaluation, plus jump-set widening is 13% slower than the simple widening strategy. We observe an anomalous behavior in `astar` and `mcf`: the simple strategy results in a small slowdown. These benchmarks have the two fastest runtimes in the benchmark suite; e.g., we analyze `mcf` in 0.02 seconds. Thus, we believe that the unexpected behavior is due to runtime noise that outlives a sequence of 100 executions of each benchmark.

## 4.5 Conclusion

In this chapter, we have presented our range analysis algorithm. We described the data structures that we have used as well as the algorithm that we propose in this work. We introduced the notion of future values, as the key insight to handle comparisons between variables. Furthermore, we discussed the trade-off of each optional step of our algorithm, giving a clear idea that our users have some control of the speed and precision of our analysis. To the best of our knowledge, this work does the most extensive evaluation in the literature of different settings of range analysis. Figure 4.13, for instance, compares 32 different configurations in terms of precision and analysis time.



# Chapter 5

## Integer Overflows

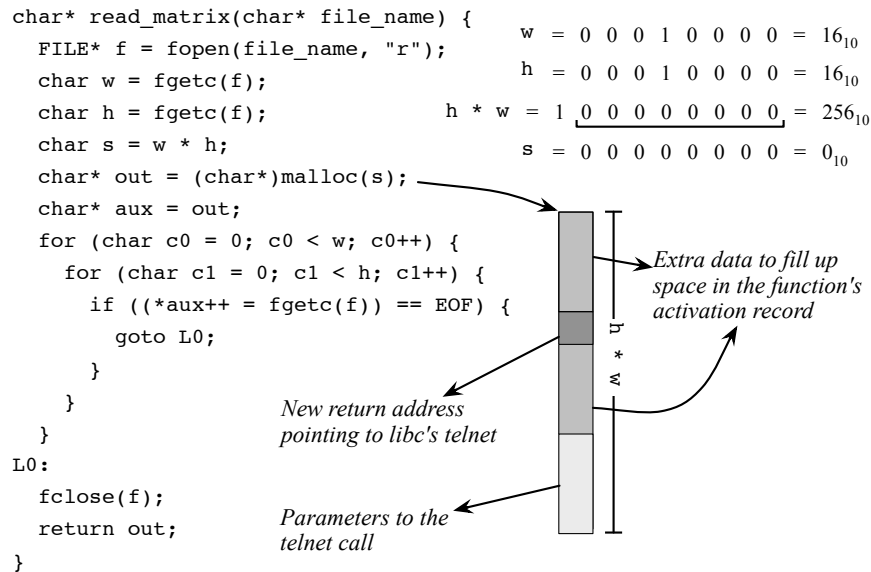
In addition to being a source of several bugs that are hard to identify, integer overflows can represent a serious security risk. As pointed by Dietz et al. [2012], integer overflows are the sources of vulnerabilities in widely used software, such as OpenSSH <sup>1</sup> and Mozilla Firefox <sup>2</sup>. Those vulnerabilities could give an adversary the opportunity to execute arbitrary code in the compromised application. We will demonstrate this with an example. The  $n$ -bits two's complement arithmetics wraps around at multiples of  $2^n$ . As an example the representation of  $16 \times 16$  in 8-bits two's complement is 0, because  $0 \equiv (16 \times 16) \pmod{2^8}$ . Programmers can use this well-defined behavior in benign ways. For instance, the wrapping arithmetics provides a cheap  $\pmod{2^n}$  operator, which is used in the implementation of hash-functions or random number generators.

Figure 5.1 illustrates an example of integer overflow vulnerability. This vulnerability allows an adversary to perform a buffer overrun attack on a program that is apparently guarded against this type of exploit. For simplicity, we will use the C datatype `char`, which represents 8-bit long numbers. Examples of actual vulnerabilities, with larger primitive types, are given by Brumley et al. [2007], Dietz et al. [2012], and Zhang et al. [2010]. The function `read_matrix` reads a matrix of bytes from a file. This matrix is stored as a linear sequence of bytes in a vector `out`. The first and second bytes in the file, e.g., `w` and `h`, determine the number of rows and columns in the matrix. If the `char` datatype could represent arbitrarily long numbers, than this function would be safe against buffer overruns, because it would only write data on allocated memory. However, if, for instance, we have that `w= 16` and `h= 16`, then `s= 0`, as pointed before. In this case, a buffer of length zero would be allocated, and all the data stored in the file would overwrite memory in the stack of activation records. By

---

<sup>1</sup><http://www.openssh.com/>

<sup>2</sup><http://www.mozilla.org/firefox/>



**Figure 5.1.** An example of an exploitable integer overflow vulnerability.

carefully crafting an input file, and adversary can, in this way, overwrite the return address of `read_matrix`, diverting program's execution to one of `libc`'s function, such as `telnet`, for instance.

## 5.1 The Dynamic Instrumentation Library

We have implemented our instrumentation library as an LLVM transformation pass; thus, we work at the level of the compiler's intermediate representation<sup>3</sup>. This is in contrast to previous works, which either transforms the source code Dietz et al. [2012] or the machine dependent code Brumley et al. [2007]. We work at the intermediate representation level to be able to couple our library with static analyses, such as the algorithm that we described in Chapter 4. Our instrumentation works by identifying the instructions that may lead to an overflow, and inserting assertions after those instructions. The LLVM IR has five instructions that may result in arithmetic overflow: `ADD`, `SUB`, `MUL`, `TRUNC` (also bit-casts) and `SHL` (left shift). Figure 5.2 shows the dynamic tests that we perform to detect overflows.

The instrumentation that we insert is mostly straightforward. We take the values of the operands and the resulting value of the checked operation to verify if an overflow occurred. We discuss in the rest of this section a few interesting cases. When dealing

<sup>3</sup><http://llvm.org/docs/LangRef.html>

Instruction	Dynamic Check
$x = o_1 +_s o_2$	$(o_1 > 0 \wedge o_2 > 0 \wedge x < 0) \vee$ $(o_1 < 0 \wedge o_2 < 0 \wedge x > 0)$
$x = o_1 +_u o_2$	$x < o_1 \vee x < o_2$
$x = o_1 -_s o_2$	$(o_1 < 0 \vee o_2 > 0 \vee x > 0) \vee$ $(o_1 > 0 \vee o_2 < 0 \vee x < 0)$
$x = o_1 -_u o_2$	$o_1 < o_2$
$x = o_1 \times_{u/s} o_2$	$x \neq 0 \Rightarrow x \div o_1 \neq o_2$
$x = o_1 \ll n$	$(o_1 > 0 \wedge x < o_1) \vee (o_1 < 0 \wedge n \neq 0)$
$x = \downarrow_n o_1$	$\text{cast}(x, \text{type}(o_1)) \neq o_1$

**Figure 5.2.** Overflow checks. We use  $\downarrow_n$  for the operation that truncates to  $n$  bits. The subscript  $s$  indicates a signed instruction; the subscript  $u$  indicate an unsigned operation.

with an unsigned SUB instruction, e.g.  $x = o_1 -_u o_2$ , then a single check is enough to detect the bad behavior:  $o_1 < o_2$ . If  $o_2$  is greater than  $o_1$ , then we assume that it is a bug to try to represent a negative number in unsigned arithmetics. Regarding multiplication, e.g.,  $x = o_1 \times o_2$ , if  $o_1 = 0$ , then this operation can never cause an overflow. This test is necessary, because we check integer overflows in multiplication via the inverse operation, e.g., integer division. Thus, the test prevents a division by zero from happening. The TRUNC instruction, e.g.,  $x = \downarrow_n o_1$  assigns to  $x$  the  $n$  least significant bits of  $o_1$ . The dynamic check, in this case, consists in expanding  $x$  to the datatype of  $o_1$  and comparing the expanded value with  $o_1$ . The LLVM IR provides instructions to perform these type expansions. Note that our instrumentation catches any truncation that might result in data loss, even if this loss is benign. To make the dynamic checks more liberal, we give users the possibility of disabling tests over truncations.

**Practical Considerations.** Our instrumentation library inserts new instructions into the target program. Although the dynamic check depends on the instruction that is instrumented, the general modus operandi is the same. Dynamic tests check for overflows after they happen. The code that we insert to detect the overflow diverts

	ADD	SUB	MUL	SHL	TRUNC
signed	12	12	6	8	3
unsigned	4	2	6	2	3

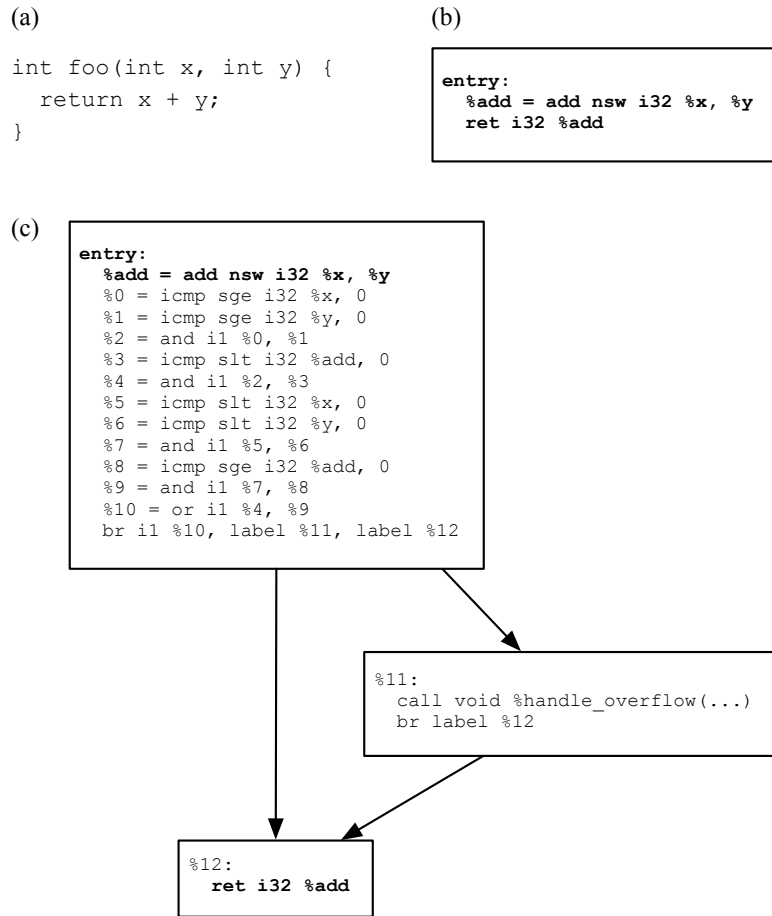
**Figure 5.3.** Number of instructions used in each check.

the program flow in case such an event takes place. Figure 5.4 shows an actual control flow graph, before and after the instrumentation. Clearly the instrumented program will be larger than the original code. Figure 5.3 shows how many LLVM instructions are necessary to instrument each arithmetic operation. These numbers do not include the instructions necessary to handle the overflow itself, e.g., block %11 in Figure 5.4, as this code is not in the program’s main path. Nevertheless, as we show empirically, this growth is small when compared to the total size of our benchmarks, because most of the instructions in these programs do not demand instrumentation. Furthermore, none of the instructions used to instrument integer arithmetics access memory. Therefore, the overall slowdown that the instrumentation causes is usually small, and the experiments in Section 5.2 confirm this observation.

Which actions are performed once the overflow is detected depends on the user, who has the option to overwrite the `handle_overflow` function in Figure 5.4. Our library gives the user three options to handle overflows. First option: no-op. This option allows us to verify the slowdown produced by the new instructions. Second option: logging. This is the standard option, and it preserves the behavior of the instrumented program. Whenever an overflow is detected, we print "Overflow detected in FileName.cpp, line X." in the standard error stream. Third option: abort. This option terminates the program once an overflow is detected. Thus, it disallows undefined behavior due to integer overflows, and gives us the opportunity to use the u-SSA form to get extra precision.

**Using the static analysis to avoid some overflow checks.** Our library can, optionally, use the range analysis to avoid having to insert some overflow checks into the instrumented program. We give the user the possibility of calling the range analysis with either the e-SSA or the u-SSA live range splitting strategies. Our static analysis classifies variables into four categories, depending on their bounds:

- **Safe:** a variable is safe if its bounds are fully contained inside its declared type. For instance, if  $x$  is declared as an unsigned 8-bits integer, then  $x$  is safe if its bounds are within the interval  $[0, 255]$ .



**Figure 5.4.** (a) A simple C function. (b) The same function converted to the LLVM intermediate representation. (c) The instrumented code. The boldface lines were part of the original program.

- **Suspicious:** we say that a variable is suspicious if its bounds go beyond the interval of its declared type, but the intersection between these two ranges is non-empty. For instance, the same variable  $x$  would be suspicious if  $I[x] = [0, 257]$ , as  $I[x]_{\uparrow} > \text{uint8}_{\uparrow}$ .
- **Uncertain:** we classify a variable as uncertain if at least one of its limits is unbounded. Our variable  $x$  would be uncertain if  $I[x] = [0, \infty]$ . We distinguish suspicious from uncertain variables because we speculate that actual overflows are more common among elements in the former category.
- **Buggy:** a variable is buggy if the intersection between its inferred range and the range of its declared type is empty. This is a definitive case of an overflow.

Continuing with our example,  $x$  would be buggy if, for instance,  $I[x] = [257, \infty]$ , given that  $[257, \infty] \cap [0, 255] = \emptyset$ .

Independent on the arithmetic instruction that is being analyzed, the instrumentation library performs the same test: if the result  $x$  of an arithmetic instruction such as  $x = o_1 +_s o_2$  is safe, then the overflow check is not necessary, otherwise it must be created.

## 5.2 Experimental Results

We have implemented our integer overflow check algorithm in LLVM 3.3, and have run experiments on a Intel quad core CPU with a 2.40GHz clock, and 3.6GB of RAM. Each core has a 4,096KB L1 cache. We have used Linux Ubuntu 10.04.4. We have executed the instrumented programs of the integer benchmarks of SPEC 2006 CPU to probe the overhead imposed by our instrumentation. These programs have been executed with their "test" input sets. We have not been able to run the binary that LLVM produces for SPEC's gcc in our environment, even without any of our transformations, due to an incompatible ctype.h header. In addition, we have not been able to collect the statistics about the overflows that occurred in SPEC's bzip2, because the log file was too large. We verified more than 3,000,000,000 overflows in this program. Table 5.1 shows the percentage of instructions that we instrument, without the intervention of the range analysis. The number of instrumented instructions is relatively low, compared to the total number of instructions, because we only instrument six different LLVM bitcodes, in a set of 57 opcodes, not counting intrinsics. Table 5.1 also shows how many instructions have caused overflows. On the average, 4.90% of the instrumented sites have caused integer overflows.

Table 5.2 shows how many checks our range analysis avoids. Some results are expressive: the range analysis avoids 1,138 out of 1,142 checks in 470.1bm. In other benchmarks, such as in 429.mcf, we have been able to avoid only 1 out of 165 tests. In general we fare better in programs that bound input sizes via conditional tests, as 1bm does. Using u-SSA, instead of e-SSA, adds a negligible improvement onto our results. We speculate that this improvement is small because variables tend to be used a small number of times. Boissinot et al. [2008] have demonstrated that the vast majority of all the program variables are used less than five times in the program code. The u-SSA form only helps to avoid checks upon variables that are used more than once.

Table 5.3 shows how our range analysis classifies instructions. Out of all the 102,790 instructions that we have instrumented in SPEC, 3.92% are suspicious, 17.19%

Benchmark	#I	#II	#II/#I	#O
470.lbm	13,724	1,142	8.32%	0
433.milc	44,236	1,602	3.62%	11
444.namd	100,276	3,234	3.23%	12
447.dealII	1,381,408	36,157	2.62%	50
450.soplex	136,367	3,158	2.32%	13
464.h264ref	271,627	13,846	5.10%	167
473.astar	19,243	857	4.45%	0
458.sjeng	54,051	2,504	4.63%	68
429.mcf	4,725	165	3.49%	8
471.omnetpp	203,201	1,972	0.97%	2
403.gcc	1,419,456	18,669	1.32%	N/A
445.gobmk	308,475	14,129	4.58%	4
462.libquantum	16,297	928	5.69%	7
401.bzip2	38,831	2,158	5.56%	N/A
456.hmmer	114,136	4,001	3.51%	0
Total (Average)	275,070	6,968	3.96%	

**Table 5.1.** Instrumentation without support of range analysis. #I: number of LLVM bitcode instructions in the original program. #II: number of instructions that have been instrumented. #O: number of instructions that actually overflowed in the dynamic tests.

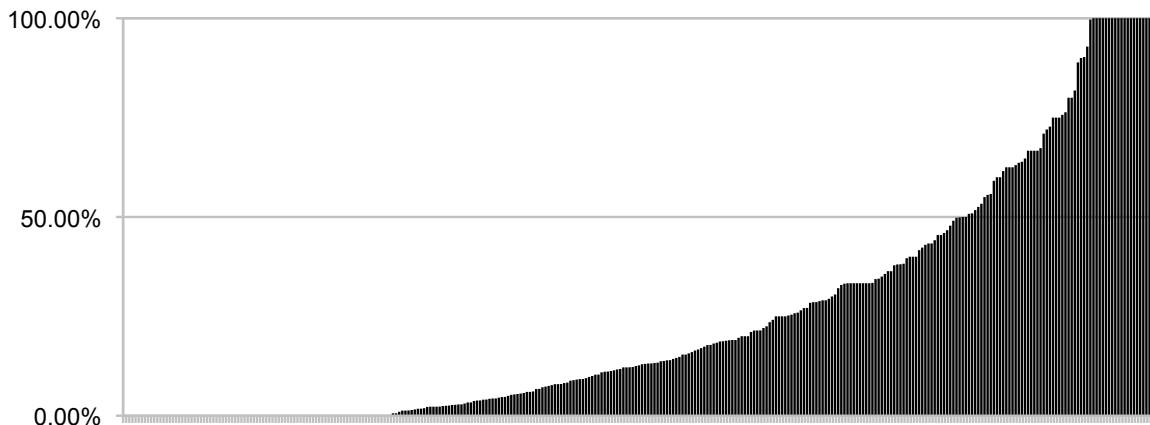
are safe, and 78.89% are uncertain. This means that we found precise bounds to  $3.92 + 17.19 = 21.11\%$  of all the program variables, and that 78.98% of them are bound to intervals with at least one unknown limit. We had, at first, speculated that overflows would be more common among suspicious instructions, as their bounds, inferred statically, go beyond the limits of their declared types. However, our experiments did not let us confirm this hypothesis. To check the correctness of our approach, we have instrumented the safe instructions, but have not observed any overflow caused by them.

Figure 5.5 shows, for the entire LLVM test suite, the percentage of overflow checks that our range analysis, with the e-SSA intermediate representation, could avoid. Each bar refers to a specific benchmark in the test suite. We only consider applications that had at least one instrumented instruction; the total number of benchmarks that meet this requirement is 333. On the average, our range analysis avoids 24.93% of the overflow checks. Considering the benchmarks in SPEC 2006 only, this number is 20.57%.

Figure 5.6 shows the impact of our instrumentation in the runtime of the SPEC benchmarks. We ran each benchmark 20 times. The largest slowdown that we have

Benchmark	#II	#E	%(II, E)	#U	%(II, U)
lbm	1,142	4	99.65%	4	99.65%
milc	1,602	1,070	33.21%	1,065	33.52%
namd	3,234	2,900	10.33%	2,900	10.33%
dealII	36,157	29,870	17.39%	28,779	20.41%
soplex	3,158	2,927	7.31%	2,897	8.26%
h264ref	13,846	11,342	18.38%	11,301	18.08%
astar	857	808	5.72%	806	5.95%
sjeng	2,504	2,354	5.99%	2,190	12.54%
mcf	165	164	0.61%	164	0.61%
omnetpp	1,972	1,313	33.42%	1,313	33.42%
gcc	18,669	15,282	18.14%	15,110	19.06%
gobmk	14,129	12,563	11.08%	12,478	11.69%
libquantum	928	820	11.64%	817	11.96%
bzip2	2,158	1,966	8.90%	1,966	8.90%
hmmmer	4,001	3,346	16.37%	3,304	17.42%
Total	104,522	86,688		85,135	

**Table 5.2.** Instrumentation library with support of static range analysis. #II: number of instructions that have been instrumented without range analysis. #E: number of instructions instrumented in the e-SSA form program. #U: number of instructions instrumented in the u-SSA form program.



**Figure 5.5.** Percentage of overflow checks that our range analysis removes. Each bar is a benchmark in the LLVM test suite. Benchmarks have been ordered by the effectiveness of the range analysis. On average, we have eliminated 24.93% of the checks (geomean).

observed, 11.83%, happened in `h264ref`, the benchmark that presented the largest number of distinct sites where overflows happened dynamically. On the average, the



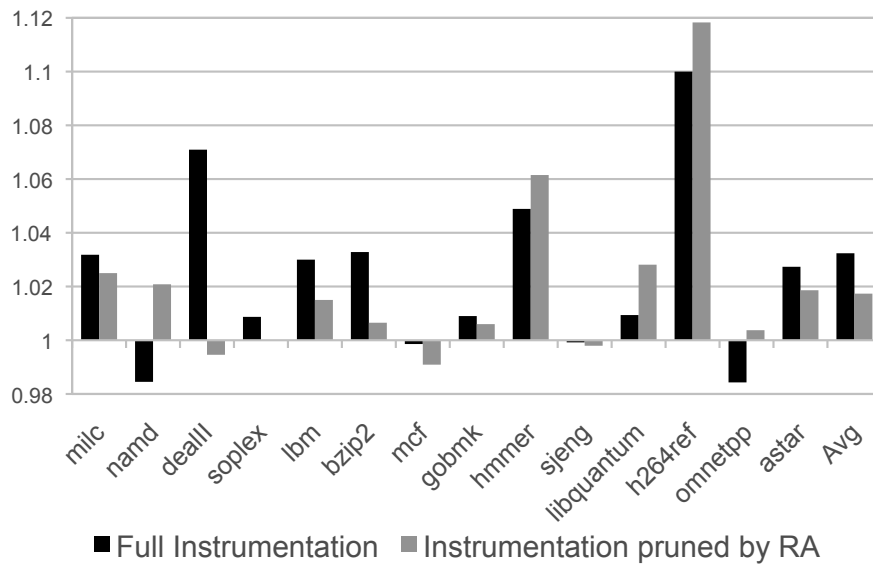
Bench	#Sf	#S	#U	#SO	#SO/#S	#UO	#UO/#U
lbm	1138	0	4	0	0,00%	0	0,00%
milc	536	17	1048	0	0,00%	11	1,05%
namd	334	480	2420	0	0,00%	12	0,50%
dealII	6188	39	28740	0	0,00%	50	0,17%
soplex	229	16	2881	0	0,00%	13	0,45%
h264ref	2539	1195	10147	7	0,59%	160	1,58%
astar	48	11	795	0	0,00%	0	0,00%
sjeng	150	213	1977	0	0,00%	68	3,44%
mcf	1	0	164	0	0,00%	8	4,88%
omnetpp	659	25	1288	1	4,00%	1	0,07%
gcc	3365	1045	14065	N/A	N/A	N/A	N/A
gobmk	1509	742	11736	0	0,00%	4	0,03%
libqtum	104	12	805	0	0,00%	7	0,87%
bzip2	192	40	1926	N/A	N/A	N/A	N/A
hmmer	663	222	3082	0	0,00%	0	0,00%

**Table 5.3.** How the range analysis classified arithmetic instructions in the u-SSA form programs. #Sf: safe. #S: suspicious. #U: uncertain. #SO: number of suspicious instructions that overflowed. #UO: number of uncertain instructions that overflowed.

instrumented programs are 3.24% slower than the original benchmarks. If we use the range analysis to eliminate overflow checks, this slowdown falls to 1.73%. The range analysis, in this case, reduces the instrumentation overhead by 46.60%. This improvement is larger than the percentage of overflow checks that we avoid, e.g., 20.57%. We believe that this difference is due to the fact that we are able to eliminate checks on induction variables, as our range analysis can rely on the loop boundaries to achieve this end. We have not noticed any runtime difference between programs converted to e-SSA form or u-SSA form. Surprisingly, some of the instrumented programs run faster than the original code. This behavior has also been observed by Dietz et al. [2012].

## 5.3 Conclusion

In this chapter we have presented a technique to secure programs against integer overflows. Our strategy consists in detecting the integer overflows right after it has occurred, by evaluating an expression containing only the operands and the result of the checked instruction. Surprisingly, this simple transformation have produced less overhead in the instrumented programs than any previous approach. Furthermore, we



**Figure 5.6.** Comparison between execution times with and without pruning, normalized by the original program’s execution time.

have also demonstrated that it is possible to reduce even more the slowdown of the checked programs by pruning unnecessary checks. We have used our range analysis to avoid inserting overflow checks in instructions classified as safe. Our experiments have shown that we were able to avoid inserting 24.93% of the checks, reducing the runtime overhead by 46.60%.

# Chapter 6

## Trip count prediction

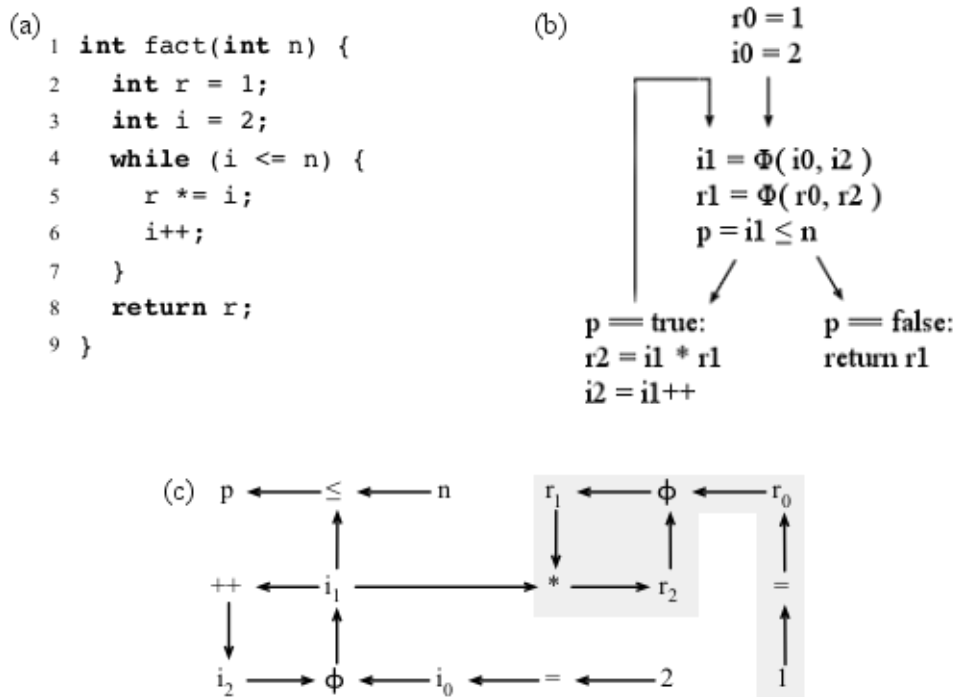
In this chapter, we present a heuristic that extracts patterns of the updates of variables' values and estimates the trip count of loops with symbolic expressions. Those expressions might, then, be evaluated at runtime and allow the compiler to decide dynamically what code to execute depending on the actual expected number of iterations. Here we demonstrate that most of the loops of a large benchmark with more than 400 programs have a structure that is easy to analyze. Furthermore, we show that our heuristic was able to estimate precisely the trip count for more than 70% of such loops, using a simple technique that might be applied in Just-In-Time compilers.

### 6.1 Background

Our analysis combines information contained in the Control Flow Graph (CFG) and in the Data Dependence Graph of the program. From the CFG we can extract information about the structure of the analyzed program, like the points of the program where the loops start and stop, and which variables and instructions directly affect the execution flow. From the dependence graph we can extract information about the way that the information flow among the variables. Those information allow us to generate symbolic expressions that estimate the trip count of the loops of the program.

The dependence graph used here was defined by Ferrante et al. [1987] and is defined in the following way: For each program variable  $v$ , we create a node  $n_v$ , and for each instruction  $i$  in the program we create a node  $n_i$ . For each instruction  $i : v = f(\dots, u, \dots)$  that defines a variable  $v$  and uses a variable  $u$  we create two edges:  $n_u \rightarrow n_i$  and  $n_i \rightarrow n_v$ .

In a program, many variables do not affect the predicates that represent the loops' stop conditions. Thus, we do not need to consider those variables in our analy-



**Figure 6.1.** (a) Example program. (b) CFG of the program, after conversion to SSA form. (c) Dependence graph highlighting nodes that do not affect the loop predicate, after converting the original program into .

sis, because they do not have any impact on the number of iterations of those loops. Therefore, despite of working with a slice of the program that eliminates those instructions, the result of our analysis remains the same. Figure 6.1 shows a dependence graph for the factorial function and highlights the variables that we can prune before doing our analysis.

### 6.1.1 Natural Loops

According to [Appel and Palsberg, 2002, p.376], a natural loop is a set of nodes  $S$  of the control flow graph (CFG) of a program, including a header node  $H$ , with the following properties:

- from any node in  $S$  there is a path that reaches  $H$ ;
- there is a path from  $H$  to any node that belongs to  $S$ ;
- any path from a node outside  $S$  to a node inside  $S$  contains  $H$ .

A node  $PH$  of the CFG is a pre-header of a natural loop if and only if  $PH$  has  $H$  as an immediate successor. In this work we normalize the CFG, so that every natural loop have one unique pre-header  $PH$ , that is executed immediately before the first iteration of the natural loop that succeeds it. The *stop condition* of a loop is a boolean expression  $E = f(e_1, e_2, \dots, e_n)$ , where each  $e_j, 1 \leq j \leq n$  is a value that contributes to the computation of  $E$ .

Depending on the stop condition of a natural loop we classify it in one of the following categories:

- **Interval Loops** The *stop condition* is an integer comparison instruction that receives two operands  $e_1$  and  $e_2$  and compares them with an inequality ( $<$ ,  $\leq$ ,  $>$ , or  $\geq$ ).
- **Equality Loops** The *stop condition* is an integer comparison instruction that receives two operands  $e_1$  and  $e_2$  and compares them with an equality ( $==$  or  $!=$ ).
- **Other Loops** Any natural loop that neither is an *Interval Loop* nor is an *Equality Loop*.

### 6.1.2 Strongly Connected Components

Variables that are redefined during the execution of a loop of the program belong to cycles in the dependence graph. That means that the value contained in a variable is used to compute the future value that the same variable might assume. Those redefinition cycles can be identified by computing the Strongly Connected Components (SCCs) on the dependence graph by using Tarjan [1972]'s or Nuutila and Soisalon-Soininen [1994]'s algorithm. This helps us to group the different nodes of the graph that belong to the same redefinition sequence.

After we have computed the SCCs of the dependence graph, the SCCs can be classified in the following way:

- **Single-node SCC** - SCCs composed by only one node.
- **Multi-node SCC** - SCCs composed by more than one node. SCCs of this class represent cycles in the dependence graph.

Multi-node SCCs can be divided in two categories:

- **Single-path SCC** - From any node in the SCC there is only one path that starts and ends in the same node and passes through the edges of the SCC at most once.

- **Multi-path SCC** - There is at least one node in the SCC for which there are two or more paths that start and end in the same node and pass through the edges of the SCC at most once.

Single-path SCCs are unconditional sequences of redefinitions of a variable. This pattern of redefinition is the easiest to analyze, because it is possible to compute the effect of one iteration in the program loop over the SCC variables. Multi-path SCCs are conditional sequences of redefinitions of a variable. This means that there are conditional branches inside the CFG loop. The number of branches makes the total number of possible paths to grow exponentially. Thus, this class of SCCs is harder to analyze.

Furthermore, Multi-path SCCs can be classified into two different categories:

- **Single-loop SCC** - The SCC has branches that does not constitute nested loops.
- **Nested-loop SCC** - There are inner loops inside the SCC.

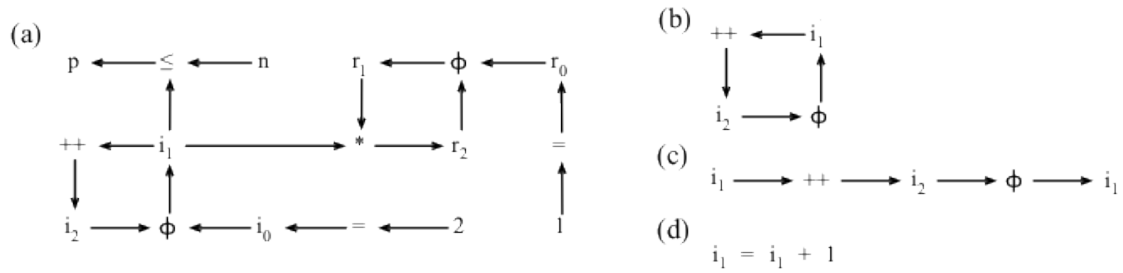
Although we have the possibility of an exponential number of paths in Multi-path SCCs, the number of paths in Single-loop SCCs is finite, so we can enumerate all the possible paths and do our analysis. Section 6.4 we show that just 3.1% of all the Multi-node SCCs have more than 1000 paths. Therefore, even if we set a limit in the number of paths to avoid exponential behavior, we will still be able to analyze more than 95% of the SCCs of the programs. On the other hand, Nested-loop SCCs have infinite possible paths, what would make our current approach to not stop. As a consequence, we exclude Nested-loop SCCs from our analysis. In Section 6.4 we show that this kind of SCC represents just 7.99% of the Multi-path SCCs and we can avoid analyze them for now.

### 6.1.3 Sequences of Redefinitions of Variables

A sequence of redefinitions (SR) is a path in the SCC that starts and ends in the same node and does not repeat any edge. By construction, our dependence graph does not admit self loops, so SRs are only extracted from Multi-node SCCs. A SR can be interpreted to generate the effect of one iteration of the program on a given variable. Figure 6.2 shows an example of SR for one induction variable.

Considering infinite precision, a SR have one of the following classifications:

- **Constant** - after one iteration through this path, the value of the variable remains the same.



**Figure 6.2.** (a)Dependence graph. (b)Multi-node SCC of the variable  $i_1$ . (c)Sequence of redefinitions of the variable  $i_1$ . (d)Effect of one iteration on the variable  $i_1$

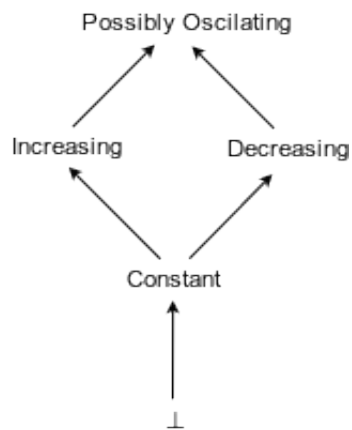
- **Increasing** - after one iteration through this path, the value of the variable is always larger than the initial value.
- **Decreasing** - after one iteration through this path, the value of the variable is always smaller than the initial value.
- **Possibly Oscillating** - after one iteration, we are not able to prove neither an increasing nor decreasing behavior.

We classify the SRs by interpreting them and comparing the final value of the variable with its starting value.

SRs that are classified as *Possibly Oscillating* are placed in this category because at least one of the following reasons is true:

- There is a call instruction in the SR. Currently our analysis is not able to analyze interprocedural SRs.
- There is an operation in the SR that receives an operand  $X$ , where the range analysis of  $X$  states that  $X_{\downarrow} < 0$  and  $X_{\uparrow} > 0$ .
- The SR depends on SCCs that have been classified as *Possibly Oscillating*.

We can classify a Multi-node SCC within the same categories of a single SR. For that, the classification of all the the SRs of the SCC must be combined using a meet operation in the lattice shown in Figure 6.3. Therefore, the classification of a SCC is the least upper bound of the classifications of the SRs that the SCC contains.



**Figure 6.3.** Lattice of SR classifications.

### 6.1.4 Vectors

In order to achieve good precision, without sacrificing efficiency, we use an abstraction called vectors to predict trip count. We place each numeric variable  $v$  of the analyzed program on the real number line in the point corresponding to the value stored by  $v$ . Thus, whenever the value that  $v$  stores is changed, we move  $v$  to another point of the real line, corresponding to its new value. By doing that, we have observed that some variables have a well defined behavior along loop iterations, that we can translate into patterns of movement. The vectors are, then, the structures that help us to understand those patterns of movement.

A vector is the step given by a variable  $v$  after one complete iteration through a SR  $p$ . Before the execution of  $p$ , we have  $v_0$  stored in  $v$ . After the execution of  $p$ ,  $v$  will be redefined with a new value,  $v_p$ . We can understand this redefinition of  $v$  as a move on the real number line. The step given by  $v$  in the line is  $v_p - v_0$ . Thus, a vector of a variable  $v$  extracted from a SR  $p$  is  $\Delta v_p$ .

The sign of  $\Delta v_p$  indicates the direction of the vector (i.e. the direction to where we are moving  $v$ ). Vectors may be defined by symbolic expressions involving other variables of the program. This characteristic generates a chain of dependencies that brings the need to process the SCCs in topological order. If the SCC of a variable  $n$  is classified as *Possibly Oscillating*, then the vectors that depend on  $n$  are unknown.

### 6.1.5 Patterns of movement

When variables have their values updated always by vectors with the same characteristics, some patterns of movement are noticeable:



- **Stationary** - Occurs when the variables are updated by vectors with modules equal to zero.
- **Constant Speed** - Occurs when the variable is updated by a vector with constant module. In this case, in each iteration, the variable is moved a constant distance from its previous location, creating a linear behavior.
- **Constant Acceleration** - Occurs when the variable is updated by a vector that has a linearly increasing module. This kind of vector is generated by a linear expression involving a *Constant Speed* variable, creating a quadratic behavior.
- **Constantly Increasing Acceleration** Occurs when the variable is updated by a vector that is generated by a linear expression involving a *Constant Acceleration* variable, creating a cubic behavior.
- **More Than Cubic** Occurs when the variable is updated by a vector that is generated by a linear expression involving a *Constantly Increasing Acceleration* or *More Than Cubic* variable, creating a more-than-cubic behavior.
- **Unknown** - Occurs when:
  - Variables are updated with vectors that depend on variables with *Unknown* movement patterns.
  - Variables are updated with vectors that have their modules decreasing in each iteration.

## 6.2 A Trip Count Algorithm Based on Vectors

In order to estimate the trip count of loops, the classic static analysis techniques based on abstract interpretation can not achieve the precision we want to have. That is true because given that most of the loops have their number of iterations controlled by the data that comes from the input, a purely static analysis will never be precise enough to solve this problem. We propose, then, a hybrid solution involving a static and a dynamic step: we statically analyze the program and generate symbolic expressions that represent the estimated trip counts of its loops. Dynamically, during its execution, the instrumented program evaluates those expressions with a  $O(1)$  complexity. Other optimizations may use the result of those expressions to make decisions at runtime, depending on the expected trip count. Any of those expressions that remain unused at the end of the compilation process can be trivially removed by a dead code elimination procedure.

---

**Algorithm 1** Trip Count Instrumentation Based on Vectors

---

**Input:** Program  $P$ **Output:** Program  $P$  with new instructions that estimate the maximum trip count of the loops

```

1: for all Loop  $l \in P$  do
2:   if not isOtherLoops( $l$ ) then
3:     Variable  $op_1 = \text{getFirstOperand}(l.\text{getStopCondition}())$ 
4:     Variable  $op_2 = \text{getSecondOperand}(l.\text{getStopCondition}())$ 
5:     Expression  $step = \text{estimateMinimumStep}(op_1, op_2)$ 
6:     if  $\exists step$  then
7:       Insert instruction  $|op_1 - op_2|/step$  before the first iteration of  $l$ .
8:     end if
9:   end if
10: end for

```

---

Algorithm 1 presents the static analysis needed to generate the trip count expressions using vectors. Our heuristic only covers *Interval Loops* and *Equality Loops*, although it is possible to extend this code in order to handle some of the *Other Loops*. For sake of simplicity, we have chosen to ignore *Other Loops*, as they represent a small part of the loops we have analyzed. *Interval Loops* and *Equality Loops* guaranteedly have two operands. Once we have collected both operands of the stop condition of a loop  $l$ , we have to estimate the minimum step of approximation of the two variables in the real numbers line. If there is a well defined behavior of update of both variables, then  $\text{estimateMinimumStep}(op_1, op_2)$  will return a valid step and we can estimate the trip count. On the contrary, the program will not receive new instructions to estimate the trip count of  $l$ .

---

**Algorithm 2**  $\text{estimateMinimumStep}$ : Estimate the minimum step of approximation of variables in the real line.

---

**Input:** Pair of Variables  $op_1, op_2$ **Output:** Expression  $step$  with the minimum step.

```

1: Vector  $v_1 = \text{getMinVector}(op_1)$ 
2: Vector  $v_2 = \text{getMinVector}(op_2)$ 
3: if  $\exists v_1$  and  $\exists v_2$  then
4:   return  $|v_1 - v_2|$ 
5: else
6:   return null
7: end if

```

---

Algorithm 2 shows how we estimate the minimum step given the minimum vectors of the two variables that control the stop condition of the loop. The variables will only have a minimum vector if they have a monotonic behavior i.e. whenever the variables move in the real line, they move in the same direction. As we have implemented only *Stationary* and *Constant Speed* vectors, the calculation of the step is given by the subtraction of both vectors. When we implement more complex vectors, the step calculation will turn into a more complicated method, eventually involving the calculation of an integral.

---

**Algorithm 3** getMinVector: Generate the minimum vector of a given monotonic variable

---

**Input:** Variable  $v$   
**Output:** Vector  $\vec{V}$  with the minimum length.

- 1: Vector  $\vec{V} = \perp$
- 2: **for all** *RedefinitionSequence*  $rs$  of  $v$  **do**
- 3:   Vector  $\vec{Tmp} = evaluateDelta(rs)$
- 4:    $\vec{V} = joinVectors_{min}(\vec{V}, \vec{Tmp})$
- 5:   **if**  $\vec{V} == unknown$  **then**
- 6:     **break**
- 7:   **end if**
- 8: **end for**
- 9: **return**  $\vec{V}$

---

Algorithm 3 generates the minimum vector for a given variable. In order to generate such vector, we have to symbolically evaluate every *Sequence of Redefinition* of that variable and join the results in a vector. The join operation has two steps: first we check the direction of both vectors. If the vectors have opposite directions or one of the vectors is *unknown*, the result of  $joinVectors_{min}(\vec{V}, \vec{Tmp})$  is *unknown*. Else, we take the vector with the minimum length as the result of the join. If during the processing the vector  $\vec{V}$  assumes the value *unknown*, then we stop the processing routine, because that means that we have completely lost our precision.

## 6.3 A Simplified Trip Count Algorithm Based on Vectors for JIT compilers

With the massive increase of the usage of the World-Wide-Web and the introduction of many new architectures that must run the same programs, it is essential to have portable programs. Code interpreting provides easy portability of programs, because just the interpreter must be translated into the different architectures, instead of any program of a given language. However, code interpreting is slow and excessively consumes resources. In this context, Just-In-Time (JIT) compilers are used to overcome the inefficiencies that code interpreters inherently have (Plezbert and Cytron [1997]).

JIT compilers work by compiling pieces of code right before they are executed. Whenever the controller thread tries to execute some function that has no native code available, the controller thread calls the compiler before executing the function. That means that the execution stops while the JIT compiler is generating native code. Therefore, the JIT compiler must generate the most optimized code possible in the minimum time, because the total time (compiling + execution) must be lower than the interpreting time, or else there is no point in compiling those programs. Because of that, JIT

compilers must use extremely lightweight algorithms to keep the compiling time as low as possible.

Here we present a simplification in our trip count prediction heuristic in order to be able to apply it in JIT compilers. As we have observed in section 6.4, 90% of the natural loops are either Interval Loops or Equality Loops. Moreover, most of our vectors are constant speed vectors with length equal to one. From those facts, in the simple heuristic we assume that in every Interval or Equality loops the minimum step of approximation of  $op_1$  and  $op_2$  is equal to one. Thus, the estimated trip count is  $|op_2 - op_1|$  and we avoid calling *EstimateMinimumStep*( $op_1$ ,  $op_2$ ). Our heuristic generates the expression that estimates the trip count with  $O(1)$  complexity. The complexity of the analysis of the whole program is  $O(n)$ , where  $n$  is the number of natural loops of the program.

## 6.4 Experimental Results

We have implemented the heuristics presented in this chapter and an analysis that observes the structure of the loops of programs with regards to the taxonomy that we have defined here. We have implemented our analyses in the LLVM compiler, version 3.3, and have used it to analyze more than 500 programs, including the benchmarks of the LLVM test-suite and the benchmarks of SPEC 2006 CPU. In this session we will focus the discussion in the results obtained with the analysis of the benchmarks of SPEC 2006 CPU. We have chosen that set of benchmarks, because they reflect well a subset of general real world programs. Moreover, SPEC benchmarks are accepted by the scientific community and are widely used as a base to compare different algorithms.

Most of the loops of the programs have a simple structure, and that means that the analysis does not need to be complicated in order to cover almost all loops of a program. Table 6.1 analyzes the structure of natural loops of programs. According to Ferrante et al. [1987], natural loops are *single-entry* regions. We have observed that 65.92% of the loops have just one instruction that decides when the loop must stop or not. Those loops are, then, *single-entry* and *single-exit* regions, what is an interesting property and might allow some aggressive optimization to be applied. However, 39.87% of the loops are nested inside other loops. Those numbers tell us that despite of the simplicity of most loops, a considerable amount of them is nested, so loop analyses that does not support nested loops leave a large number of loops uncovered.

We have also identified a pattern in the stop conditions of the loops. Table 6.2 shows that approximately 85% of the natural loops have a single integer comparison

Program	L	NL	% NL/L	SEL	% SEL/L
433.milc	426	211	49.53%	399	93.66%
444.namd	623	418	67.09%	593	95.18%
447.dealII	6526	2695	41.30%	3412	52.28%
450.soplex	742	181	24.39%	554	74.66%
470.lbm	23	10	43.48%	23	100.00%
401.bzip2	238	85	35.71%	150	63.03%
403.gcc	4614	1357	29.41%	3202	69.40%
429.mcf	50	9	18.00%	39	78.00%
445.gobmk	1288	482	37.42%	913	70.89%
456.hmmmer	881	245	27.81%	740	84.00%
458.sjeng	267	62	23.22%	201	75.28%
462.libquantum	98	13	13.27%	90	91.84%
464.h264ref	1870	1008	53.90%	1784	95.40%
471.omnetpp	465	66	14.19%	249	53.55%
473.astar	119	37	31.09%	104	87.39%
483.xalancbmk	3106	259	8.34%	1611	51.87%
Total	21336	7138	33.46%	14064	65.92%

**Table 6.1.** Natural Loops in the Control Flow Graph. L: number of natural loops. NL: number of nested loops. SEL: number of loops that have a single exit point.

Program	L	IL	% IL/L	EL	% EL/L	OL	% OL/L
433.milc	426	417	97.89%	5	1.17%	4	0.94%
444.namd	623	494	79.29%	7	1.12%	122	19.58%
447.dealII	6526	4597	70.44%	604	9.26%	1325	20.30%
450.soplex	742	572	77.09%	101	13.61%	69	9.30%
470.lbm	23	23	100.00%	0	0.00%	0	0.00%
401.bzip2	238	201	84.45%	29	12.18%	8	3.36%
403.gcc	4614	2103	45.58%	1954	42.35%	557	12.07%
429.mcf	50	17	34.00%	28	56.00%	5	10.00%
445.gobmk	1288	1098	85.25%	131	10.17%	59	4.58%
456.hmmmer	881	697	79.11%	109	12.37%	75	8.51%
458.sjeng	267	117	43.82%	128	47.94%	22	8.24%
462.libquantum	98	88	89.80%	6	6.12%	4	4.08%
464.h264ref	1870	1789	95.67%	19	1.02%	62	3.32%
471.omnetpp	465	283	60.86%	82	17.63%	100	21.51%
473.astar	119	108	90.76%	1	0.84%	10	8.40%
483.xalancbmk	3106	1687	54.31%	752	24.21%	667	21.47%
Total	21336	14291	66.98%	3956	18.54%	3089	14.48%

**Table 6.2.** Classification of Natural Loops according to their stop conditions. L: number of natural loops. IL: number of *Interval Loops*. EL: number of *Equality Loops*. OL: number of *Other Loops*.

as the stop condition. Moreover, the vast majority of those loops are interval loops, the easiest kind of loop to analyze. We have also observed similar proportions while analyzing the rest of our benchmarks. Those numbers are favorable to our heuristics, because we take advantage of the simplicity of the loops to produce precise results with simple algorithms.

Program	SN	MN	SP	% SP/MN	MP	SL	% SL/MP	NL	% NL/MN
433.milc	2507	426	409	96.01%	17	11	64.71%	6	1.41%
444.namd	5879	781	604	77.34%	177	6	3.39%	171	21.90%
447.dealII	79169	7249	6077	83.83%	1172	505	43.09%	667	9.20%
450.soplex	13032	807	683	84.63%	124	53	42.74%	71	8.80%
470.lbm	94	24	23	95.83%	1	1	100.00%	0	0.00%
401.bzip2	3610	214	171	79.91%	43	16	37.21%	27	12.62%
403.gcc	123775	5121	4513	88.13%	608	276	45.39%	332	6.48%
429.mcf	1022	54	40	74.07%	14	7	50.00%	7	12.96%
445.gobmk	17675	1555	1283	82.51%	272	163	59.93%	109	7.01%
456.hmmmer	12215	946	825	87.21%	121	67	55.37%	54	5.71%
458.sjeng	3337	276	221	80.07%	55	38	69.09%	17	6.16%
462.libquantum	1439	123	100	81.30%	23	8	34.78%	15	12.20%
464.h264ref	21502	1946	1841	94.60%	105	27	25.71%	78	4.01%
471.omnetpp	12383	470	379	80.64%	91	39	42.86%	52	11.06%
473.astar	2591	138	118	85.51%	20	7	35.00%	13	9.42%
483.xalancbmk	57181	3024	2486	82.21%	538	373	69.33%	165	5.46%
Total	357411	23154	19773	85.40%	3381	1597	47.23%	1784	7.70%

**Table 6.3.** Classification of Strongly Connected Components in the Dependence Graph. SN: number of *Single-Node* SCCs. MN: number of *Multi-Node* SCCs. SP: number of *Single-Path* SCCs. MP: number of *Multi-Path* SCCs. SL: number of *Single-Loop* SCCs. NL: number of *Nested-Loop* SCCs.

The dependence graph of the programs has been inspected during our research. In order to avoid interferences caused by variables that do not affect the stop conditions of the loops, we have pruned the graph before analyzing it. We did backward depth-first searches starting from the stop conditions and discarded the nodes that were not visited at least once. Table 6.3 shows the statistics collected while analyzing the dependence graphs of the programs. 85.40% of the Multi-Node SCCs are Single-Path SCCs. That means that there is only one SR for the variables of such SCCs. Moreover, just 7.70% of the Multi-Node SCCs have nested cycles and do not fulfill the requirements of our analysis. All the presented data confirms that the programs have a structure that that is suitable for our heuristics to produce accurate results.

In order to estimate the trip count of a loop, our prototype must be able to infer the values that  $Op_1$  and  $Op_2$  store before the first iteration.  $Op_1$  and  $Op_2$  are the operands of the stop condition of the loop. This information is not always possible to be inferred, because sometimes one of the operands is the result of a call to other function. In cases like this, we do not know the value of the operand before the loop starts and, consequently, are not able to estimate its trip count. Moreover, both operands must be integer variables. Table 6.4 shows the number of loops of which we are able to infer the trip count. For instance, we were able to estimate the trip count of 85.50% of the interval loops of the benchmark 403.gcc, while we were able to instrument just 9.83% of its equality loops. We have investigated this and we observed that most of

Program	# IL	# IIL	% IIL/IL	# EL	# IEL	% IEL/EL
433.milc	417	391	93.76%	5	3	60.00%
444.namd	494	469	94.94%	7	1	14.29%
447.deallI	4597	3535	76.90%	604	77	12.75%
450.soplex	572	422	73.78%	101	48	47.52%
470.lbm	23	23	100.00%	0	0	-
401.bzip2	201	186	92.54%	29	7	24.14%
403.gcc	2103	1798	85.50%	1954	192	9.83%
429.mcf	17	7	41.18%	28	1	3.57%
445.gobmk	1098	1040	94.72%	131	56	42.75%
456.hammer	697	664	95.27%	109	39	35.78%
458.sjeng	117	106	90.60%	128	17	13.28%
462.libquantum	88	77	87.50%	6	1	16.67%
464.h264ref	1789	1411	78.87%	19	8	42.11%
471.omnetpp	283	238	84.10%	82	29	35.37%
473.astar	108	80	74.07%	1	1	100.00%
483.xalancbmk	1687	1403	83.17%	752	108	14.36%
Total	14291	11850	82.92%	3956	588	14.86%

**Table 6.4.** Trip Count Instrumentation. IL: interval loops. IIL: instrumented interval loops. EL: equality loops. IEL: instrumented equality loops.

the 403.gcc’s equality loops are bounded by comparisons between pointers. The same was observed in other programs. Because of that, we should focus only in the interval loops.

We have developed a profiler that collects the estimated trip count and the real trip count during an actual execution of the benchmarks. The result of our profiler let us to observe how accurate are our heuristics. We have split our accuracy results into seven categories according to the actual number  $N$  of iterations:

- $[0, \sqrt{N}]$ : Occurs when the estimated trip count is less or equal the square root of the actual trip count. For example, if we estimate that a loop will iterate 2 times and it iterates 10 times during its execution, this loop will be classified into this category.
- $[\sqrt{N}, N/2]$ : Occurs when the estimated trip count is greater than the square root of the actual trip count but is less or equal its half. For example, if we estimate that a loop will iterate 4 times and it iterates 10 times during its execution, this loop will be classified into this category.
- $[N/2, N]$ : Occurs when the estimated trip count is greater than the half of the actual trip count but is less than the trip count. For example, if we estimate that a loop will iterate 8 times and it iterates 10 times during its execution, this loop will be classified into this category.

- $[N, N]$ : Occurs when the estimated trip count equals the actual trip count. For example, if we estimate that a loop will iterate 10 times and it iterates 10 times during its execution, this loop will fall into this category.
- $]N, 2 * N]$ : Occurs when the estimated trip count is greater than the actual trip count, but is less or equal to two times the actual trip count. For example, if we estimate that a loop will iterate 16 times and it iterates 10 times during its execution, this loop will be classified into this category.
- $]2 * N, N^2]$ : Occurs when the estimated trip count is greater than two times the actual trip count, but is less or equal to the power of two of the actual trip count. For example, if we estimate that a loop will iterate 32 times and it iterates 10 times during its execution, this loop will be classified into this category.
- $]N^2, +\infty]$ : Occurs when the estimated trip count is greater than the power of two of the actual trip count. For example, if we estimate that a loop will iterate 128 times and it iterates 10 times during its execution, this loop will be classified into this category.

Table 6.5 shows the comparison between the estimated trip count and the actual trip count that we have collected with our profiler. The subtotal lines contain only the SPEC CPU benchmarks, while the total lines also include more than 300 benchmarks distributed with LLVM. While running the programs, each time a loop stops, we collect the actual trip count and compare it with the estimated trip count. Thus, the numbers that we presented is the number of *instances* of loops, instead of the number of natural loops. We did this because we may predict correctly the trip count for some instances and may predict wrongly for other instances of the same CFG loop. Table 6.6 shows information about the programs that had their trip counts estimated using the simplified heuristic, following the same rules used to build table 6.5.

By analyzing table 6.5 we can observe that our heuristic is very precise. 87.25% of the trip counts that we have predicted were the same as the actual trip count. Moreover, we have observed that more than 99% of the estimated trip counts were over-approximations of the actual trip counts. This is an interesting fact, because although we developed an heuristic, our results are still conservative. When we analyze the results obtained with the simplified heuristic, we also find some impressive numbers. As expected, the vector heuristic has better results than the simplified heuristic, but the difference was small. Our predictions were exact in 84.46% of the cases, despite of the extreme simplicity of the algorithm. Furthermore, we also have observed the same over-approximation that we have noticed with the complete vector heuristic.



Program	$[0, \sqrt{N}]$	$[\sqrt{N}, N/2]$	$[N/2, N]$	$[N, N]$	$[N, 2 * N]$	$[2 * N, N^2]$	$[N^2, +\infty]$
433.milc	14	0	0	435,514,912	38,360	9,984	1,032,930
444.namd	0	0	0	21,602,695	8,064	3,168	0
450.soplex	1,851	367	122	186,943	12,782	10,219	43,338
470.lbm	0	0	0	53,397	0	0	0
401.bzip2	8,616,650	2	311,724	13,204,855	14,195,603	1,128,948	28,939,274
403.gcc	433,588	17	326	17,240,735	1,851,284	278,164	336,422
429.mcf	96,576	87	42	555	2,643,736	634,623	1,705,369
445.gobmk	8,392	20	400	651,081	70,492	117	20,141
456.hammer	0	0	0	31,551,408	8,512,797	3,893,744	3,273,429
458.sjeng	0	620	2,565,378	41,787,788	3,423,766	7,917	1,038,075
462.libquantum	0	0	0	8,182,095	0	1	0
464.h264ref	6,749,850	0	0	311,274,945	13,427,840	57,300	228,711
473.astar	7,147	0	0	74,614,711	602,244	2,550	609,708
Subtotal	15,914,068	1,113	2,877,992	955,866,120	44,786,968	6,026,735	37,227,397
Subtotal (%)	1.50%	0.00%	0.27%	89.95%	4.21%	0.57%	3.50%
Total	25,525,142	2,078	2,922,080	4,134,074,825	163,974,403	11,363,892	400,209,181
Total (%)	0.54%	0.00%	0.06%	87.25%	3.46%	0.24%	8.45%

**Table 6.5.** Trip Count Profiler - Trip count estimated using vectors.

Program	$[0, \sqrt{N}]$	$[\sqrt{N}, N/2]$	$[N/2, N]$	$[N, N]$	$[N, 2 * N]$	$[2 * N, N^2]$	$[N^2, +\infty]$
433.milc	14	0	0	435,514,912	38,360	9,984	1,032,930
444.namd	0	0	0	21,602,688	8,065	3,174	0
450.soplex	1,851	367	112	186,939	12,784	10,231	43,338
470.lbm	0	0	0	53,333	0	64	0
401.bzip2	5,270,006	2	311,724	14,386,219	15,987,072	1,502,759	28,939,274
403.gcc	420,390	17	326	17,252,944	1,841,701	283,373	343,054
429.mcf	96,576	87	42	555	2,643,736	634,623	1,705,369
445.gobmk	8,392	20	400	651,081	70,492	117	20,141
456.hammer	0	0	0	31,551,408	8,512,797	3,893,744	3,273,429
458.sjeng	0	620	2,565,378	41,787,788	3,423,766	7,917	1,038,075
462.libquantum	0	0	0	8,182,095	0	1	0
464.h264ref	367,010	0	0	302,394,768	12,636,622	5,636,387	10,703,859
473.astar	7,147	0	0	74,614,711	602,244	2,550	609,708
Subtotal	6,171,386	1,113	2,877,982	948,179,441	45,777,639	11,984,924	47,709,177
Subtotal (%)	0.58%	0.00%	0.27%	89.22%	4.31%	1.13%	4.49%
Total	10,762,387	2,094	2,882,136	3,996,856,652	227,506,781	53,384,130	441,012,280
Total (%)	0.23%	0.00%	0.06%	84.46%	4.81%	1.13%	9.32%

**Table 6.6.** Trip Count Profiler - Trip count estimated using simplified heuristic.

## 6.5 Conclusion

In this chapter we have discussed the prediction of the number of iterations of loops. We have indicated the usefulness of such information to decide at runtime which code to execute based on the estimated trip count of loops. We also have classified the loops into a taxonomy proposed by us, which allowed us to better understand where the most promising optimizing opportunities are. In addition, we proposed the Vectors,

an abstraction inspired by physics to represent patterns of updates of variables on the real line. Furthermore, we have proposed two heuristics to estimate the trip count of loops, based on our Vectors. Finally, we have evaluated the precision of our heuristics, observing 87.25% of accuracy while analysing interval loops.

# Chapter 7

## Final considerations

In this chapter we finish this work. First, we point possible directions that we can further explore, in order to increase either the precision or the applicability of our analyses. Finally, we conclude the two-year long effort that culminates with this dissertation. We briefly visit again our contributions discussing the main results we have achieved.

### 7.1 Future Works

The contributions presented in this work can be used to achieve further goals in many research areas; some of them are already being pursued by our compiler research team. First, we can use our Range Analysis in further analyses and optimizations, like we did with our Integer Overflow Instrumentation. Second, we can extend our Range Analysis to other types of variables (e.g. `float`). Third, we can use our trip count prediction methods to create optimizations according to the estimated trip count. Finally, we can develop a Worst-Case Computational Complexity (WCCC) estimation method based on our trip count estimation.

Our Range Analysis extracts valuable information about the integer variables. We can use that information to enhance existing analyses and optimizations, such as dead code elimination or branch prediction. We can also implement a true context-sensitive analysis, instead of using function inlining, in order to try to achieve more precision without the drawback of the growth of the program size. Furthermore, our symbolic implementation of range analysis makes possible to use the Range Analysis together with algorithms based on Symbolic Execution. In addition, we can extend our Range Analysis to other numerical types (e.g. `float`) and have it compatible with a larger number of analyses.

Our trip count prediction heuristics also open new opportunities for future works. First, our simpler heuristic allows us to perform more aggressive optimizations in JIT compilers, when we estimate that we will have a large number of iterations in a given loop. Second, our vector heuristic allows us to perform optimizations that depend on the number of iterations of a given loop. For instance, some architectures like ccNUMA are very sensitive to the location of the memory read by a program running in a given core. Thus, we can choose to either move or not to move pages in the memory according to the number of iterations. Similarly, we can use the symbolic trip count of loops to derive the WCCC of a given function and decide whether it is advantageous or not to run the same function in a GPGPU (General Purpose Graphics Processing Unit).

## 7.2 Conclusions

This dissertation summarizes a two-year long effort of research on compiler analyses and optimizations. One of the more significant findings to emerge from this study is our Range Analysis algorithm. The resolution of future values – the key contribution of our research – allows our analysis to be more precise without resorting to complex techniques. Instead, we have demonstrated that in the average case, our analysis has linear time complexity in function of the program size.

Our second contribution is a technique to dynamically check integer overflows. Our approach to tackle this problem is simple, but is effective and has produced the least overhead in the programs, when compared with previous works existing in the literature. Furthermore, we were able to show an example of how other analysis or optimizations can benefit from the information provided by our range analysis to give better results to their users. Moreover, we have also extracted some properties from overflow-free programs. Those properties were used to develop the u-SSA representation, that were able to increase the precision of our Range Analysis.

Our third major finding was an heuristic to estimate the trip count of loops. Our insight of using vectors to represent the abstraction of patterns of redefinition of variables have allowed us to estimate precise trip counts for 75% of the instrumented loops. Additionally, we have developed a simplified version of our heuristic, aiming JIT compilation. Despite of the simplicity of the heuristic, we have been able to infer correctly the trip counts of 66% of the instrumented loops.

# Bibliography

- Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley.
- Ananian, S. (1999). The static single information form. Master's thesis, MIT.
- Appel, A. W. and Palsberg, J. (2002). *Modern Compiler Implementation in Java*. Cambridge University Press, 2nd edition.
- Barik, R., Grothoff, C., Gupta, R., Pandit, V., and Udupa, R. (2006). Optimal bitwise register allocation using integer linear programming. In *LCPC*, volume 4382 of *Lecture Notes in Computer Science*, pages 267–282. Springer.
- Bertrane, J., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., and Rival, X. (2010). Static analysis and verification of aerospace software by abstract interpretation. In *I@A*, pages 1--38. AIAA.
- Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., and Rival, X. (2003). A static analyzer for large safety-critical software. In *PLDI*, pages 196--207. ACM.
- Bodik, R., Gupta, R., and Sarkar, V. (2000). ABCD: eliminating array bounds checks on demand. In *PLDI*, pages 321–333. ACM.
- Boissinot, B., Darte, A., Rastello, F., de Dinechin, B. D., and Guillon, C. (2009). Revisiting out-of-SSA translation for correctness, code quality, and efficiency. In *CGO*, pages 114–125. IEEE.
- Boissinot, B., Hack, S., Grund, D., de Dinechin, B. D., and Rastello, F. (2008). Fast liveness checking for SSA-form programs. In *CGO*, pages 35–44. IEEE.
- Brumley, D., Song, D. X., cker Chiueh, T., Johnson, R., and Lin, H. (2007). RICH: Automatically protecting against integer-based vulnerabilities. In *NDSS*. USENIX.

- Campos, V. H. S., Rodrigues, R. E., de Assis Costa, I. R., and Pereira, F. M. Q. (2012). Speed and precision in range analysis. In *Programming Languages*, pages 42--56. Springer.
- Chen, P., Wang, Y., Xin, Z., Mao, B., and Xie, L. (2009). BRICK: A binary tool for run-time detecting and locating integer-based vulnerability. In *ARES*, pages 208--215.
- Chinchani, R., Iyer, A., Jayaraman, B., and Upadhyaya, S. (2004). ARCHERR: Run-time environment driven program safety. In *European Symposium on Research in Computer Security*. Springer.
- Choi, J.-D., Cytron, R., and Ferrante, J. (1991). Automatic construction of sparse data flow evaluation graphs. In *POPL*, pages 55--66.
- Clarke, E. M., Grumberg, O., and Long, D. E. (1994). Model checking and abstraction. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1512--1542.
- Cong, J., Fan, Y., Han, G., Lin, Y., Xu, J., Zhang, Z., and Cheng, X. (18-21 Jan. 2005). Bitwidth-aware scheduling and binding in high-level synthesis. *Design Automation Conference, 2005. Proceedings of the ASP-DAC 2005. Asia and South Pacific*, 2:856--861.
- Costan, A., Gaubert, S., Goubault, E., Martel, M., and Putot, S. (2005). A policy iteration algorithm for computing fixed points in static analysis of programs. In *CAV*, pages 462--475.
- Cousot, P. and Cousot, R. (1977). Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238--252. ACM.
- Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., and Rival, X. (2009). Why does astrée scale up? *Form. Methods Syst. Des.*, 35(3):229--264.
- Cousot, P. and Halbwachs, N. (1978). Automatic discovery of linear restraints among variables of a program. In *POPL*, pages 84--96. ACM.
- Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. (1991). Efficiently computing static single assignment form and the control dependence graph. *TOPLAS*, 13(4):451--490.

- Dietz, W., Li, P., Regehr, J., and Adve, V. (2012). Understanding integer overflow in `c/c++`. In *ICSE*, pages 760--770. IEEE.
- Dowson, M. (1997). The ariane 5 software failure. *SIGSOFT Softw. Eng. Notes*, 22(2):84--.
- Ermedahl, A. and Gustafsson, J. (1997). Deriving annotations for tight calculation of execution time. In *Euro-Par'97 Parallel Processing*, pages 1298--1307. Springer.
- Ferrante, J., Ottenstein, K., and Warren, J. (1987). The program dependence graph and its use in optimization. *TOPLAS*, 9(3):319--349.
- Flanagan, C., Leino, K. R. M., Lillibridge, M., Nelson, G., Saxe, J. B., and Stata, R. (2002). Extended static checking for java. In *ACM Sigplan Notices*, volume 37, pages 234--245. ACM.
- Frigge, M., Hoaglin, D. C., and Iglewicz, B. (1989). Some implementations of the boxplot. *The American Statistician*, 43(1):50--54.
- Gampe, A., von Ronne, J., Niedzielski, D., Vasek, J., and Psarris, L. (2011). Safe, multiphase bounds check elimination in java. *Software: Practice and Experience*, 41:753--788.
- Gawlitza, T., Leroux, J., Reineke, J., Seidl, H., Sutre, G., and Wilhelm, R. (2009). Polynomial precise interval analysis revisited. *Efficient Algorithms*, 1:422 -- 437.
- George, L. and Matthias, B. (2003). Taming the ixp network processor. In *PLDI*, pages 26--37. ACM.
- Gough, J. and Klaeren, H. (1994). Eliminating range checks using static single assignment form. Technical report, Queensland University of Technology.
- Gulavani, B. S. and Gulwani, S. (2008). A numerical abstract domain based on expression abstraction and max operator with application in timing analysis. In *Computer Aided Verification*, pages 370--384. Springer.
- Gulwani, S., Jain, S., and Koskinen, E. (2009a). Control-flow refinement and progress invariants for bound analysis. In *ACM Sigplan Notices*, volume 44, pages 375--385. ACM.
- Gulwani, S., Mehra, K. K., and Chilimbi, T. (2009b). Speed: precise and efficient static estimation of program computational complexity. In *ACM SIGPLAN Notices*, volume 44, pages 127--139. ACM.

- Halbwachs, N., Proy, Y.-E., and Roumanoff, P. (1997). Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2):157–185.
- Henry, J., Monniaux, D., and Moy, M. (2012). Pagai: a path sensitive static analyser. *Electronic Notes in Theoretical Computer Science*, 289:15–25.
- Johnson, R., Pearson, D., and Pingali, K. (1994). The program tree structure. In *PLDI*, pages 171–185. ACM.
- Johnson, R. and Pingali, K. (1993). Dependence-based program analysis. In *PLDI*, pages 78–89. ACM.
- Jung, Y., Kim, J., Shin, J., and Yi, K. (2005). Taming false alarms from a domain-unaware c analyzer by a bayesian statistical post analysis. In *SAS*, pages 203–217.
- Kennedy, K. and Allen, R. (2001). *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann.
- Kong, T. and Wilken, K. D. (1998). Precise register allocation for irregular architectures. In *MICRO*, pages 297–307. IEEE.
- Lakhdar-Chaouch, L., Jeannet, B., and Girault, A. (2011). Widening with thresholds for programs with complex control graphs. In *ATVA*, pages 492–502. Springer-Verlag.
- Lattner, C. and Adve, V. S. (2004). LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88. IEEE.
- Lhairech-Lebreton, G., Coussy, P., Heller, D., and Martin, E. (2010). Bitwidth-aware high-level synthesis for designing low-power dsp applications. In *ICECS*, pages 531–534. IEEE.
- Liu, Y. A. and Gomez, G. (1998). Automatic accurate time-bound analysis for high-level languages. In *Languages, Compilers, and Tools for Embedded Systems*, pages 31–40. Springer.
- Lo, R., Chow, F., Kennedy, R., Liu, S.-M., and Tu, P. (1998). Register promotion by sparse partial redundancy elimination of loads and stores. In *PLDI*, pages 26–37. ACM.
- Lokuciejewski, P., Cordes, D., Falk, H., and Marwedel, P. (2009). A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models. In *CGO*, pages 136–146. IEEE.



- Lundqvist, T. and Stenström, P. (1998). Integrating path and timing analysis using instruction-level simulation techniques. In *Languages, Compilers, and Tools for Embedded Systems*, pages 1--15. Springer.
- Mahlke, S., Ravindran, R., Schlansker, M., Schreiber, R., and Sherwood, T. (2001). Bitwidth cognizant architecture synthesis of custom hardware accelerators. *Computer-Aided Design of Integrated Circuits and Systems*, 20(11):1355–1371.
- Miné, A. (2006). The octagon abstract domain. *Higher Order Symbol. Comput.*, 19:31–100. ISSN 1388-3690.
- Molnar, D., Li, X. C., and Wagner, D. A. (2009). Dynamic test generation to find integer bugs in x86 binary linux programs. In *SSYM*, pages 67--82. USENIX.
- Nethercote, N. and Seward, J. (2007). Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*, pages 89--100. ACM.
- Nielson, F., Nielson, H. R., and Hankin, C. (1999). *Principles of Program Analysis*. Springer.
- Nuutila, E. and Soisalon-Soininen, E. (1994). On finding the strongly connected components in a directed graph. *Inf. Process. Lett.*, 49(1):9–14.
- Oh, H., Heo, K., Lee, W., Lee, W., and Yi, K. (2012). Design and implementation of sparse global analyses for C-like languages. In *PLDI*, pages 229--238. ACM.
- Ottenstein, K. J., Ballance, R. A., and MacCabe, A. B. (1990). The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *PLDI*, pages 257--271. ACM.
- Patterson, J. R. C. (1995). Accurate static branch prediction by value range propagation. In *PLDI*, pages 67--78. ACM.
- Pereira, F. M. Q. and Palsberg, J. (2008). Register allocation by puzzle solving. In *PLDI*, pages 216–226. ACM.
- Pingali, K. and Bilardi, G. (1995). APT: A data structure for optimal control dependence computation. In *PLDI*, pages 211–222. ACM.
- Pingali, K. and Bilardi, G. (1997). Optimal control dependence computation and the roman chariots problem. In *TOPLAS*, pages 462--491. ACM.

- Plevyak, J. B. (1996). *Optimization of Object-Oriented and Concurrent Programs*. PhD thesis, University of Illinois at Urbana-Champaign.
- Plezbart, M. P. and Cytron, R. K. (1997). Does “just in time”=“better late than never”? In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 120--131. ACM.
- Ramalingam, G. (2002). On sparse evaluation representations. *Theoretical Computer Science*, 277(1-2):119--147.
- Rice, H. G. (1953). Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358--366.
- Rimsa, A. A., D’Amorim, M., and Pereira, F. M. Q. (2011). Tainted flow analysis on e-SSA-form programs. In *CC*, pages 124--143. Springer.
- Rodrigues, R. E. and Pereira, F. M. Q. (2013). Prevenção de ataques de nao-terminação baseados em estouros de precisão. In *Anais do XVII Simpósio Brasileiro de Linguagens de Programação*, pages 32--46. SBC.
- Rodrigues, R. E., Sperle Campos, V. H., and Quintao Pereira, F. M. (2013). A fast and low-overhead technique to secure programs against integer overflows. In *Code Generation and Optimization (CGO), 2013 IEEE/ACM International Symposium on*, pages 1--11. IEEE.
- Scholz, B. and Eckstein, E. (2002). Register allocation for irregular architectures. In *LCTES/SCOPE5*, pages 139--148. ACM.
- Schwartzbach, M. I. (2008). Lecture notes on static analysis. *Basic Research in Computer Science, University of Aarhus, Denmark*.
- Simon, A. (2008). *Value-Range Analysis of C Programs: Towards Proving the Absence of Buffer Overflow Vulnerabilities*. Springer, 1th edition.
- Singer, J. (2006). *Static Program Analysis Based on Virtual Register Renaming*. PhD thesis, University of Cambridge.
- Stephenson, M., Babb, J., and Amarasinghe, S. (2000). Bitwidth analysis with application to silicon compilation. In *PLDI*, pages 108--120. ACM.
- Su, Z. and Wagner, D. (2004). A class of polynomially solvable range constraints for interval analysis without widenings and narrowings. In *TACAS*, pages 280--295.

- Su, Z. and Wagner, D. (2005). A class of polynomially solvable range constraints for interval analysis without widenings. *Theoretical Computer Science*, 345(1):122–138.
- Tallam, S. and Gupta, R. (2003). Bitwidth aware global register allocation. In *POPL*, pages 85–96. ACM.
- Tarjan, R. E. (1972). Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160.
- Tavares, A. L. C., Boissinot, B., Bigonha, M. A. S., Bigonha, R., Pereira, F. M. Q., and Rastello, F. (201X). A program representation for sparse dataflow analyses. *Science of Computer Programming*, X(X):2–25. Invited paper with publication expected for 2012.
- Tu, P. and Padua, D. (1995). Efficient building and placing of gating functions. In *PLDI*, pages 47–55. ACM.
- Venet, A. and Brat, G. (2004). Precise and efficient static array bound checking for large embedded c programs. *SIGPLAN Not.*, 39:231–242.
- Wagner, D., Foster, J. S., Brewer, E. A., and Aiken, A. (2000). A first step towards automated detection of buffer overrun vulnerabilities. In *NDSS*, pages 3–17. ACM.
- Wang, T., Wei, T., Lin, Z., and Zou, W. (2009). Intscope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In *NDSS*. Internet Society.
- Warren, H. S. (2002). *Hacker’s Delight*. Addison-Wesley Longman Publishing Co., Inc.
- Wolfe, M. J., Shanklin, C., and Ortega, L. (1995). *High performance compilers for parallel computing*. Addison-Wesley Longman Publishing Co., Inc.
- Yang, Y., Yan, H., Shao, Z., and Guo, M. (2011). Compiler-assisted dynamic scratch-pad memory management with space overlapping for embedded systems. *Software: Practice and Experience*, 41(7):737–752.
- Zhang, C., Wang, T., Wei, T., Chen, Y., and Zou, W. (2010). Intpatch: automatically fix integer-overflow-to-buffer-overflow vulnerability at compile-time. In *ESORICS*, pages 71–86. Springer-Verlag.