

RESTRITIFICAÇÃO DE ARGUMENTOS DE FUNÇÃO

VICTOR HUGO SPERLE CAMPOS

RESTRITIFICAÇÃO DE ARGUMENTOS DE FUNÇÃO

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

Agosto de 2016

VICTOR HUGO SPERLE CAMPOS

RESTRICTIFICATION OF FUNCTION ARGUMENTS

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

August 2016

Ficha catalográfica elaborada pela Biblioteca do ICEx - UFMG

Campos, Victor Hugo Sperle.

C198r Restrictification of function arguments. / Victor Hugo
Sperle Campos. – Belo Horizonte, 2016.
xxiv, 53 f.: il.; 29 cm.

Dissertações (mestrado) - Universidade Federal de
Minas Gerais – Departamento de Ciência da Computação.

Orientador: Fernando Magno Quintão Pereira.

1. Computação- Teses. 2. Compiladores (Computadores).
3. Análise estática. 4. Otimização de código. I. Orientador.
II. Título.

CDU 519.6*32(043)



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

FOLHA DE APROVAÇÃO

Restrictification of function arguments

VICTOR HUGO SPERLE CAMPOS

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

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

PROFA. KÉCIA ALINE MARQUES FERREIRA
Departamento de Computação - CEFET-MG

PROF. SANDRO RIGO
Instituto de Computação - UNICAMP

Belo Horizonte, 26 de agosto de 2016.

*To my parents, to my sister, to my friends from UFMG and elsewhere, I shall
dedicate this work.*

Acknowledgments

First and foremost, I thank my family for all the support throughout these 25 years I have lived. My father, Ruy Cabral de Campos; my mother, Alzira Sperle Campos; and my sister, Dayanne Sperle Campos. Thanks for being the best relatives of all time and for loving me despite my many untold flaws.

My deepest appreciations to my advisors, whose tips and directions have been always more worth it than I ever realized: Fernando Pereira and Nelson Amaral. The former that had all the patience in the world with me for five and a half years, and never ever hesitated in lifting my confidence and referring me to great opportunities. I will be always thankful to you, Fernando. And Nelson, for welcoming me in the cold Edmonton and introducing me to research in cooperation with the industry.

A warm thank you to Michael Frank. He was one of the earliest that trusted my capabilities and brought funding to this project. Also, he is among the very few people in the industry that is actually kind and cares about people.

I sincerely thank all of my mates from graduation class, 091, the one and the best. Among them, my best friends: Diogo Rennó, Anderson Freitas, Hugo Vellozo, Alexandre Guelman and Eduardo Gomes. All the studying we did before classes, discussing ideas for the assignments, and more specially, all the laughs we shared amidst these challenges. I also thank my more recent friends from the Master's period: Lucas Silva, Luís Pena and Tony Gonçalves. My friends from the soccer teams I have been in the Department's Soccer League: Stupid Freshmen, Segmentation Fanta and Inimigos do TP. Two people that made my 2015 one of the best years of my life and changed my life in more than one way: Vitor Paisante and Laís German.

The best teacher that has passed through my life is also worth mentioning: Crepalde, the best Portuguese teacher to walk on Earth, who taught me how to study and how to tackle exams. A lot of what I am today with regards to study I owe to him.

The people from Programming Languages Laboratory, who were more than colleagues to me: Douglas Teixeira, Péricles Alves, Demontiê Júnior, Rubens Emílio and Henrique Santos. I never could have had a better work environment. I hope someday

to work again with all you together.

I also appreciate the presence in my life of my friends from abroad: Edward Barter, Robbie Haelzle and Joël Boisvert from Ottawa, who embraced me in times of need. The Brazilians that have grown a wonderful friendship with me in Canada, among which I shall always mention RightCast: Claudio Nuernberg, Daniel Pereira, Thiago Melo and Rafael Toledo. Also the gang that has enjoyed with me the Canadian summer to the bone: Mateus Vieira, Filipe Gomes, Tiago Jaime and Henrique Falcão. Let's do some appreciation for Edmonton, of course! My friends Gustavo Coutinho, Antônio Cavalcante, Camila Ferreira, Rodolfo Wottrich, and Marjorie Vale.

Last, but by far the most special, thanks to friends that endured the test of time and remain my dearest: Gabriel Chaves, Tâmara Souza, Rebeca Passos and Maurício Souza. I love you.

“Do not meddle in the affairs of wizards, for they are subtle and quick to anger.”
(Gandalf the Grey – J. R. R. Tolkien)

Resumo

Aliasing de ponteiros ainda prejudica as otimizações de compiladores apesar de anos de pesquisa em desambiguação de ponteiros. Por causa da dificuldade em realizar a desambiguação, diversas linguagens de programação oferecem aos programadores mecanismos para distinguir referências à memória, como a palavra-chave `restrict` na linguagem C. Entretanto, o uso de tais mecanismos é sujeito a erros humanos. Nesta dissertação, apresentamos um conjunto de técnicas automáticas para mitigar o problema. Chamamos esse conjunto de Restritificação. Nós desenvolvemos e testamos três diferentes formas de desambiguar ponteiros passados como argumentos de funções. Nossas técnicas combinam análises estáticas para inferir limites simbólicos de regiões de memória e versionamento de código. Geramos um clone para cada função cujos argumentos podem ser desambiguados e o otimizamos assumindo que os parâmetros formais da função não são alias. Em tempo de execução, usamos os resultados de comparações simbólicas para decidir qual versão da função será executada: a original, não otimizada, ou o clone “restritificado” (otimizado). A implementação dos nossos métodos, feita sobre o LLVM, mostra que podemos vetorizar até 63% mais operações do que o nível mais alto de otimização do compilador, -O3. A aplicação da técnica sobre uma biblioteca de uso bastante difundido, a OpenCV, resultou em melhorias no tempo de execução de até 40%.

Palavras-chave: Compiladores, Aliasing, Análise estática, Clonagem de função, Otimização de código.

Abstract

Pointer aliasing still hinders compiler optimizations, in spite of years of research on pointer disambiguation. Because the automatic disambiguation of pointers is a difficult endeavor, several programming languages offer programmers mechanisms to distinguish memory references, such as the “restrict” keyword in C. However, the use of such mechanisms is prone to human mistakes. In this paper we present a suite of automatic techniques that mitigate this problem. We call this suite Restrictification. We have designed, implemented and tested three different ways to disambiguate pointers passed as arguments of functions. Our techniques combine static analyses to infer symbolic bounds of memory regions and code versioning. We generate a clone for each function whose arguments we can disambiguate and optimize it assuming the absence of aliasing among formal parameters. At runtime, we use the results of the symbolic interval tests to decide which version of a function we should call: the original one or the “restricted” (optimized) clone, whenever we can prove that no aliasing can occur at runtime. An implementation of our restrictification methods in LLVM shows that we can vectorize up to 63% more operations than what could be accomplished using the -O3 optimization level of said compiler. When applying the optimization on OpenCV benchmarks, we have observed speedups as great as 40%.

Palavras-chave: Compiler, Aliasing, Static analysis, Function cloning, Optimization.

List of Figures

2.1	Simplified compilation process.	7
2.2	Abstract Syntax Tree.	8
2.3	Fibonacci function in the LLVM Intermediate Representation.	9
2.4	Control flow graph of Fibonacci function.	10
2.5	Example of the usage of restrict. Taken directly from the ISO/IEC 9899:1999 standard.	12
3.1	Our different analyses let us disambiguate all the pointer arguments passed to the two <code>sumTill*</code> functions.	18
3.2	Optimized versions of the functions seen in Figure 3.1. Notice that these functions are only equivalent to their original selves in the absence of aliasing.	19
3.3	Invocation sites seen earlier in Figure 3.1 rewritten to accommodate our optimization. Our approaches are named as follows: PSA - purely static approach (Sec. 4.1); FRI - forward region inference (Sec. 4.2.1); BRI - backward region inference (Sec. 4.2.2). If the approach is able to restrictify function arguments at a given call site, then we mark it with <i>ok</i> , otherwise we mark it with <i>no</i>	20
3.4	Version of <code>sumTillSize</code> optimized by our backward analysis of Section 4.2.2.	21
4.1	A comparison between the different pointer disambiguation approaches that we use in this dissertation. WPI : Whole-program Information affects precision. Ch.Call : involves inserting computation at call sites. Ch.Fun : involves inserting computation in the target function. Dyn : determines, based on runtime values, when it is safe to call optimized code.	23
4.2	Snippet of function <code>main</code> , seen in Figure 3.1, rewritten after the replacement phase used in purely static restrictification.	26
4.3	Forward Region Inference Analysis.	29

4.4	The result of applying the forward region inference analysis on the program seen in Figure 3.1.	30
4.5	Example of code produced by our backward region inference analysis. Memory accesses are marked grey. Arrows indicate the constraints produced for matrix A	31
5.1	Execution time ratio of each benchmark suite for our three techniques (lower is better).	40
5.2	Execution time ratio for each benchmark in Shootout-C++, using our three methods (lower is better).	40
5.3	Execution time ratio for the core and imgproc test modules of OpenCV, using the PSA and BRI methods. Each point along the horizontal axis represents a different test. Lower is better. The vertical line in each chart marks the point where ratios become greater than or equal to 1.	42
5.4	(a) Example of loop found in OpenCV’s core benchmarks. (b) Version of the loop that LLVM -O3 produces.	44

List of Tables

5.1	Benchmark descriptions: size (number of LLVM Intermediate Representation instructions), number of functions overall, number of functions eligible for cloning (take two or more pointer arguments), number of call sites where eligible functions are invoked.	36
5.2	Reach of our different disambiguation methods. PSA : number of call sites optimized by the Purely Static Approach. FRI : number of call sites instrumented when using Forward Region Inference. BRI : number of functions guarded with dynamic checks by the Backward Region Inference.	36
5.3	Relative improvement of code optimizations when combined with our different disambiguation techniques. PSA : purely static approach. FRI : forward region inference. BRI : backward region inference.	38
5.4	Relative increase in executable’s size when using each disambiguation method. PSA : purely static. FRI : forward region inference. BRI : backward region inference.	39

Contents

Acknowledgments	xi
Resumo	xv
Abstract	xvii
List of Figures	xix
List of Tables	xxi
1 Introduction	1
1.1 Publications	3
1.2 Earlier Publications	4
1.3 Structure of the Document	4
2 Literature Review	7
2.1 Compilation Process	7
2.2 Control Flow Graph	9
2.3 Pointers	11
2.3.1 Pointer Aliasing and the Restrict Keyword	11
2.4 Pointer Disambiguation	12
2.5 Applications	14
2.6 Final Thoughts	15
3 Overview	17
3.1 Final Thoughts	21
4 Restrictification of Function Arguments	23
4.1 Static Restrictification	24
4.1.1 Dynamic Linkage	25

4.2	Hybrid Restrictification	27
4.2.1	Forward Region Inference	28
4.2.2	Backward Region Inference	30
4.3	Code Generation	32
4.4	Final Thoughts	33
5	Evaluation	35
5.1	The Reach of Restrictification	36
5.2	Strengthening Code Optimizations	37
5.3	Code Size Expansion	39
5.4	Speedup	39
5.5	Testing Our Approach on Larger Programs	41
5.6	Final Thoughts	44
6	Conclusion	45
6.1	Future Work	46
6.2	Final Thoughts	47
	Bibliography	49

Chapter 1

Introduction

Compilers need to deal with pointers to carry out effective optimizations. A staple feature in imperative programming languages, pointers pose several challenges to code analyses and transformations [Hind, 2001]. These challenges stem from the possibility of aliasing, which increases the number of dependences among data; hence, forcing optimizations to be more conservative [Appel and Palsberg, 2002, Ch 17.5]. Aliasing has compelled the programming languages community to design, implement, and test a very large number of different techniques to disambiguate pointers. Nevertheless, the classic memory disambiguation approaches are either too costly [Jones and Muchnick, 1982; Whaley and Lam, 2004] or too imprecise [Steensgaard, 1996] to be practical.

To mitigate the problem posed by pointers, some programming languages furnish developers with high-level constructs that they can use to tell compilers that variables do not alias each other. Examples of such techniques include the several systems of ownership types [Noble et al., 1998], which, today, find concrete use in keywords like Rust’s `mut` and Scala’s `@unique`. In this direction, the C programming language, since the C99 Standard, features the `restrict` keyword, which programmers use to indicate that a memory position can only be accessed through a pointer p , or through values derived from p , such as $p + 1$. Yet, the task of annotating type declarations with `restrict` or similar modifiers still belongs to the programmer, and contrary to Rust’s or Scala’s type system, C’s does not perform any verification, be it statically or dynamically, to ensure that annotations are not misplaced.

In this dissertation, we call the act of disambiguating two pointers “Restrictification”, a new term conceived by us. We attack the aliasing problem by proposing three different and complementary techniques to restrictify pointers used as arguments of functions. Our goal, as we explain in Section 3, is to provide compilers with information that enables them to perform more extensive optimizations. Restrictification is not

always possible: it depends on the calling site where a function is invoked, and it depends on the arguments passed to that function. To maximize the number of times that we can apply it, we resort to code cloning: for each function that is “restrictifiable”, we create a clone of it. In this clone, the compiler is free to assume absence of aliasing between the function’s arguments. Even though, in principle, we could already benefit from partial restrictification, i.e., by disambiguating only some of the function’s arguments, for simplicity, we only use a restricted clone when we can determine that none of its arguments alias each other. The three restrictification approaches that we introduce in this paper can be used separately, or combined, as the application of one does not prevent the use of the other. Another advantage of our approach is that these three techniques rely on infrastructure already available in mainstream compilers. Thus, their implementation is feasible; and as we will show, very effective. They are described as follows:

- In Section 4.1, we discuss a *purely static* way to disambiguate function arguments. We use off-the-shelf static analyses to decide, for each call site, when the restricted clone can be called. As we explain in Section 4.1.1, we are able to apply this approach even when we compile the modules that constitute a program separately.
- In Section 4.2.1, we discuss a hybrid restrictification approach, which performs a *forward* analysis to infer the size of memory regions and uses this information to create statically, at each call site, checks that decide, dynamically, if it is possible to invoke the restricted function instead of its original version.
- In Section 4.2.2, we discuss a second hybrid approach that uses a *backward* analysis to determine the bounds of memory regions. With this information, we also statically produce checks to determine, dynamically, when it is safe to branch to code that is aliasing-free.

To validate the ideas discussed in this paper, we have implemented them in the LLVM compilation infrastructure [Lattner and Adve, 2004], version 3.7. As we show in Section 5, our different pointer disambiguation techniques let this compiler produce non-trivial speedups in several different benchmarks. For instance, the extra aliasing information that we provide lets LLVM produce code 8% faster for MiBench and Stanford, two of the benchmark suites that were selected for our experimental section. The code size expansion caused by function cloning is affordable: we have observed an increase of less than 10% in most programs. Furthermore, our technique can bring benefits that would not be possible with inlining, our most direct competitor. One of such benefits is the possibility to apply it onto library code, without the need

to recompile client programs. To support this last statement, we have applied our optimizations onto OpenCV, a broadly used computer vision library. In this experiment, we were able to speedup 39% of the benchmarks.

Contributions. This work is the first to automatically insert the `restrict` keyword in pointer arguments. Before our techniques, programmers had the burden of annotating arguments with the keyword. This task is tedious, time consuming (for legacy code to be annotated can be large), and error prone (given that a mistakenly annotated function can severely impact the program’s correctness). Of the three techniques described in this dissertation, we developed from scratch the first one, the Purely Static Method, including the extension for dynamic linkage. We are clients of the other two methods, the hybrid ones. Nonetheless, the work of deployment, integration, experiments, and empirical analysis of the three methods was done entirely by us.

1.1 Publications

This project has resulted in three publications:

- The first, *Restritificação* [Campos et al., 2015b], presents the first implementation of two (out of three) techniques we discuss in this dissertation: the static method (Section 4.1) and the backward inference method (Section 4.2.2). We limited ourselves to testing our methods on benchmarks.
- Alongside our first publication, we also published *Restrictifier: a tool to disambiguate pointers at function call sites* [Campos et al., 2015a]. This is a tool based on our static method that can either be downloaded or be used via a Web interface.
- Our most relevant text, *Restrictification of function arguments* [Sperle Campos et al., 2016], is the final description of our project. On top of what had been accomplished by Campos et al. [2015b], we augmented our implementation with the forward inference method (Section 4.2.1). The static method was also improved by the addition of the dynamic linkage extension (Section 4.1.1). Lastly, we evaluated our methods not only on benchmarks but also on a real world, widely used computer vision library, OpenCV.

We were awarded the title of *Best Paper* in the Brazilian Symposium on Programming Languages 2012, and also the title of *Best Tool* in the Congresso Brasileiro de Software - Sessão de Ferramentas 2012.

1.2 Earlier Publications

Previous to this project, we had two papers published that are indirectly related to the current subject.

- *Speed and precision in range analysis* [Campos et al., 2012] is the first description of our integer range analysis algorithm. This implementation finds, for every integer variable in the program, the range of values it may assume during the program’s execution. Such information can be used for a number of applications, like improving security of programs by detecting buffer overflows. More related to the current project, this information can be employed in the pointer range comparisons used to disambiguate pointers in a function.
- In 2013, we had *A fast and low-overhead technique to secure programs against integer overflows* [Rodrigues et al., 2013] which extends Campos et al. [2012] with an automatic tool to insert runtime checks in programs in order to protect them against integer overflows. The range analysis framework is used to prove that some of the checks are unnecessary because the integer operation never overflows. In our experiments, 25% of the checks were proven unnecessary, which led to 50% of runtime improvements in secured programs.

Even though Restrictification does not use this integer range analysis algorithm, the symbolic range calculation used here is based on those earlier designs. Therefore those two initial works were of great importance to the current project and to this dissertation.

1.3 Structure of the Document

The rest of this document is organized as follows. Chapter 2 presents a literature review of compilers, optimizations, and all the work regarding pointer aliasing and disambiguation. In Chapter 3, a background about the three techniques is lectured alongside a motivating example. The core of our proposed solution is in Chapter 4, where we explain in depth our three implemented methods: the purely static approach, the forward region inference, and the backward region inference. Chapter 5 contains all the description about the experiments we performed to assess the quality of Restrictification. At the end, Chapter 6 wraps things up with our final remarks.

The next chapter reviews a few key concepts that we shall be using throughout this text. We start by recalling basic notions related to the process of compiling programs.

From this point, we funnel down towards more specific concepts related to this work. In particular, we re-state the problem of pointer disambiguation, and explain why it is important.

Chapter 2

Literature Review

2.1 Compilation Process

Compiler is a program that translates source code from a programming language into code in another language [Aho et al., 2006]. This latter language is generally binary, which makes the output binary code, ready to be executed after linking. This translation process is divided in several steps, which are shown in Figure 2.1.

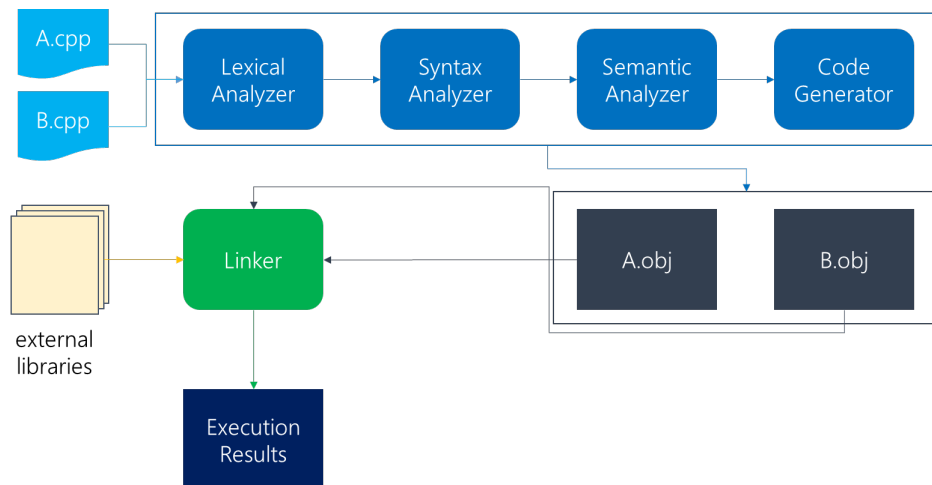


Figure 2.1. Simplified compilation process.

Lexical analysis is the process of analyzing a stream of individual characters (normally arranged as lines) into a sequence of lexical tokens to feed into the parser (or syntactic analyzer). Roughly the equivalent of splitting ordinary text written in a natural language (e.g. English) into a sequence of words and punctuation symbols. Lexical analysis is often done with tools such as `lex` [Lesk and Schmidt, 1975], `flex` and `jflex`.

Syntactic analysis (or parsing) takes the input from a lexical analyzer in the form of token streams. The parser analyzes the source code (token stream) against the production rules to detect any errors in the code. The output of this phase is either a parse tree or an abstract syntax tree. The difference between these two lies in the fact that, in ASTs, not every detail in representation is mapped. For instance, grouping parenthesis and branch conditions don't need to be explicitly represented.

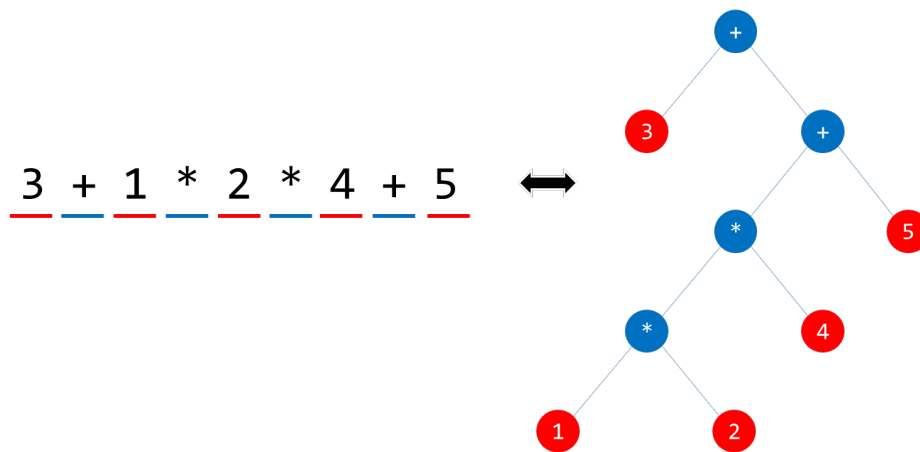


Figure 2.2. Abstract Syntax Tree.

An example of Abstract Syntax Tree is in Figure 2.2. The sequence of tokens in the left, output by a lexical analyzer, is transformed into the tree in the right. During such procedure, the source code is checked for syntactic correctness. This correctness is described as a set of rules known as a grammar. A programming language's grammar is a set of production rules that described all possible strings that can take place in the language.

Semantic analysis is the task of ensuring that the declarations and statements of a program are semantically correct, i.e. that their meaning is clear and consistent with the way in which control structures and data types are supposed to be used. This stage of the compilation process is typically responsible for three things: (i) type checking, by which data types ought to be used in a way consistent with their definition; (ii) label checking, which confirms that all labels in the code do exist; and (iii) flow control checks, which verifies if control flow constructs are correct.

Code generation is the process by which a compiler's code generator converts some intermediate representation of source code into a form (e.g. machine code) that

can be readily executed by a machine¹. This process is composed by instruction selection (which instructions to use), instruction scheduling (order of the instructions) and register allocation (assigning variables to processor registers).

The steps from lexical analysis up until semantic analysis are done in the *compiler frontend*, whereas the code generation is done in the *compiler backend*. There is also another module that partakes in the compilation pipeline: the code optimization, performed by the *compiler optimizer*. This step takes place between the semantic analysis and the code generation. It concentrates most of the analyses and transformations that aim to make programs faster and safer (some of them work in the code generation though). In general, the analysis and optimization stage works on an intermediate representation of the source program, output by the compiler frontend. Such representation is employed so that programs can be depicted regardless of their original language. In the LLVM Compiler Infrastructure [Lattner and Adve, 2004], compiler that we use in this project, the LLVM Intermediate Representation² is the implemented language.

```
define i32 @fibonacci(i32 %n) nounwind readnone {
    %1 = icmp ult i32 %n, 2
    br i1 %1, label %.loopexit, label %.lr.ph

.lr.ph:
    %i.04 = phi i32 [ %3, %.lr.ph ], [ 2, %0 ]
    %prev2.03 = phi i32 [ %fib.02, %.lr.ph ], [ 0, %0 ]
    %fib.02 = phi i32 [ %2, %.lr.ph ], [ 1, %0 ]
    %2 = add i32 %prev2.03, %fib.02
    %3 = add i32 %i.04, 1
    %4 = icmp ugt i32 %3, %n
    br i1 %4, label %.loopexit, label %.lr.ph

.loopexit:
    %.0 = phi i32 [ %n, %0 ], [ %2, %.lr.ph ]
    ret i32 %.0
}
```

Figure 2.3. Fibonacci function in the LLVM Intermediate Representation.

2.2 Control Flow Graph

The Figure 2.3 represents a Fibonacci Function in the LLVM IR. It contains not only instructions in the LLVM representation's language, but also control flow information.

¹[https://en.wikipedia.org/wiki/Code_generation_\(compiler\)](https://en.wikipedia.org/wiki/Code_generation_(compiler))

²<http://llvm.org/docs/LangRef.html>

Just before each blank line, there is an *end of block* instruction. This instruction is generally a branch instruction, which takes the control flow to another location in the program's code.

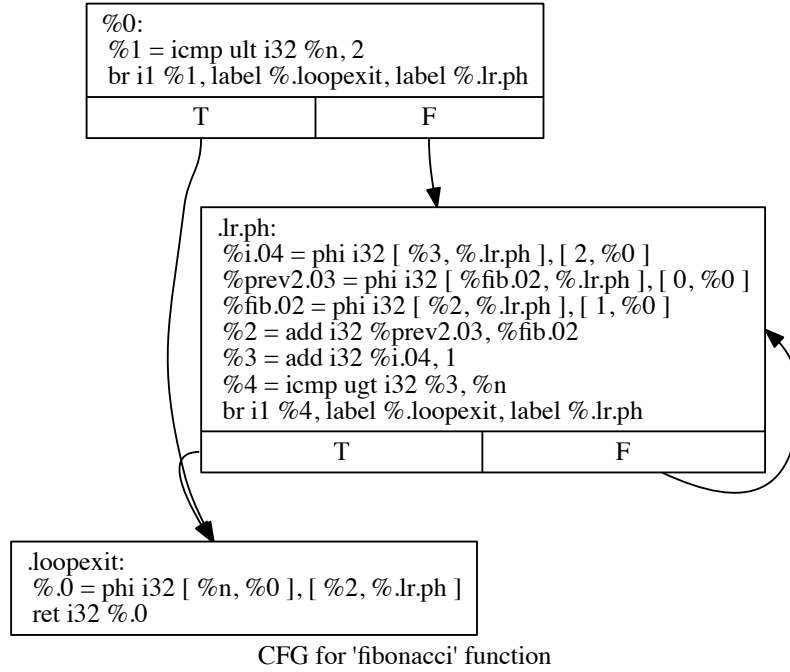


Figure 2.4. Control flow graph of Fibonacci function.

Figure 2.4 shows the same Fibonacci function in a graph representation that depicts such control flow relations. This is called the *Control Flow Graph* of the function. It contains blocks of instructions, called *basic blocks*, and branch relations between these, called *block edges*.

Basic blocks are code sequences with no branches in the middle. There are branches only to the entry and at the exit of the sequence. Under such circumstances, whenever the first instruction is executed, the rest of the instructions in the sequence are necessarily executed, in that exact order.

Block edges are the control flow relations between basic blocks. If there is a branch from a basic block $B1$ to a block $B2$, whether it being conditional or not, then a block edge from $B1$ to $B2$ exists.

Control flow graphs exist for all functions in a program. They are of great importance to perform analyses and optimizations on programs because they enable

compilers to have a complete overview of the relations between the instructions in a function.

2.3 Pointers

A pointer is a programming language object (or variable) that holds as value the memory address of another variable, through which one can access its contents. In this sense, a pointer is said to *refer* one location in memory, and the act of accessing that location via the pointer is *dereferencing*.

In general terms, a pointer is just another name for a machine language address, which can be said to “point” to a quantity of interest. In this sense, a pointer can point to only a word or a byte, depending on the relevant computer architecture. But more specifically, now, a pointer is a high-level language construct that can point to more complex data structures such as a record or a node of a graph or linked list. (Reilly [2003])

Pointers are singularly important for the purpose of the compiler optimizations discussed in this dissertation because it is through them that data structures are passed into functions. When programmers want to pass data structures to be read and/or written by functions, the most common way to achieve this is via pointers.

```
void saxpy(int n, float a, float *x, float *y) {
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}
```

Listing 2.1. Example function `saxpy` that takes pointers as arguments.

2.3.1 Pointer Aliasing and the Restrict Keyword

As first introduced in Chapter 1, the performance of the function `saxpy` in Listing 2.1 is severely diminished if there is any possibility that pointers x and y refer to the same memory location. Pointers referencing the same memory position is called *aliasing*. As will be explained in depth in Chapter 3, the presence of aliasing turns more aggressive optimizations flat out unsound, such as vectorization or multithreading parallelism.

To assist compilers in knowing when pointers do not alias, programming languages offer specialized keywords in their standard so that coders can inform compilers about

the property. For instance, the C language, since ISO/IEC 9899:1999³, contains the `restrict` qualifier for function pointer arguments. From the document:

An object that is accessed through a restrict-qualified pointer has a special association with that pointer. This association, defined in 6.7.3.1 below (*not reproduced in this dissertation*), requires that all accesses to that object use, directly or indirectly, the value of that particular pointer) The intended use of the `restrict` qualifier (like the `register` storage class) is to promote optimization, and deleting all instances of the qualifier from all preprocessing translation units composing a conforming program does not change its meaning (i.e., observable behavior). (ISO/IEC 9899:1999, p. 109)

When pointer arguments are tagged with the `restrict` keyword, compilers can assume the lack of aliasing among these pointers. This opens opportunities for more aggressive and generally more profitable optimizations, which we demonstrate in Chapter 5.

EXAMPLE 2 The function parameter declarations in the following example

```
void f(int n, int * restrict p, int * restrict q)
{
    while (n-- > 0)
        *p++ = *q++;
}
```

assert that, during each execution of the function, if an object is accessed through one of the pointer parameters, then it is not also accessed through the other.

Figure 2.5. Example of the usage of `restrict`. Taken directly from the ISO/IEC 9899:1999 standard.

2.4 Pointer Disambiguation

This dissertation describes three different techniques to disambiguate pointers. Pointer disambiguation is an important and well researched problem in computer science. Since Andersen’s work [Andersen, 1994], which represented one of the most important forays in this field, much has been accomplished. Currently, points-to analyses can be implemented very efficiently [Hardekopf and Lin, 2007; Pereira and Berlin, 2009] and very precisely [Whaley and Lam, 2004; Hardekopf and Lin, 2011]. The purely static approach that we have described in Section 4.1 can use any of these types of alias analyses to disambiguate pointers. Thus, we are a client, not a challenger, of these

³<http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf>

previous works. In the same direction, there are several works describing algorithms that are clients of alias analyses. For instance, Chen et al. [2005] have shown that finer granularity of pointer analysis can bring up to 15% of performance improvement to a typical compiler. Nevertheless, we are not aware of previous literature that tries to use static alias analyses to disambiguate pointers used as arguments of functions, as we do.

Compiler-related literature describes several different algorithms that use forward region inference to eliminate bound checks in memory accesses. As an example, Bodík et al. [2000] have used a variation of forward region inference to remove guards around memory accesses in Java. Logozzo and Fähndrich [2010] have defined *Pentagons*, a lattice that they have used to implement forward region inference. Nazaré et al. [2014] have improved *Pentagons* with symbolic ranges, which is yet another way to implement forward region inference.

In this project we use the algorithm of Nazaré et al. [2014]. The work of Nazaré et al. [2014] implements a symbolic range analysis that computes the range of values each integer variable may assume during the execution. This information is then used to infer the maximum addressable offset of pointers in the program, which is particularly useful to discover memory accesses that are guaranteed to be inbounds. Tools such as AddressSanitizer [Serebryany et al., 2012] instrument every single memory access in C programs to capture out-of-bounds dereferences during execution. Nazaré et al. [2014] are able to remove the instrumentation corresponding to memory accesses that are always safe, i.e. inbounds.

The publicly available implementation of Nazaré *et al.* has been the starting point of the algorithm that we described in Section 4.2.1. It is important to clarify that, to the best of our knowledge, no previous work has used a forward region inference analysis to disambiguate pointers passed as the arguments of functions.

The body of work most closely related to ours concerns backward region inference, because this technique has, indeed, been used to generate checks that disambiguate pointers at runtime, albeit not pointers used as function arguments. To this effect, backward region inference has been used locally, at the level of basic blocks and at the level of loops. Nicolau [1989], Gallagher et al. [1994], and [Guo et al., 2006] have proposed ways to disambiguate addresses used in load/store operations within basic blocks. Rus et al. [2003] and Alves et al. [2015] have designed backward techniques that disambiguate pointers used within nests of loops. Today, compilers such as gcc and LLVM use backward analysis to disambiguate affine memory accesses within innermost loops as a way to enable vectorization. Contrary to these previous works, our backward region inference performs this disambiguation at the function level. Algorithmically speaking, we use a global data-flow analysis that propagates information backwardly,

and we apply a meet operation (maximum) at branches, i.e., program points that span multiple paths in the control flow graph. Previous literature uses local approaches, either restricted to the analysis of straight-line code or to the analysis of induction variables of loops. Due to this global scope, our technique requires, for instance, the use of escape analysis to ensure that pointers are not updated outside the function to be restricted.

2.5 Applications

These different scopes enable different compiler optimizations. For instance, let's assume that we were using pointer disambiguation to enable code parallelization, like Rus et al. [2003] have done. In this case, Gallagher's local disambiguation would allow better instruction scheduling; hence, they would improve instruction level parallelism (ILP). Rus *et al.*'s technique would help the compiler to prove that different iterations of a loop are independent, thus creating more opportunities for the compiler to find data parallelism, in addition to enabling more aggressive ILP. Finally, by disambiguating pointers in the entire scope of a function, we can, for instance, give the compiler the opportunity to say that different - non-nested - loops could be executed in parallel. Following the parallelization jargon, we say that we are helping the compiler to identify more task parallelism, in addition of giving it more opportunities to find data and instruction-level parallelism.

Quite an effort has been spent in computation offload to GPUs [Andión et al., 2016]. Given their enormous potential for parallel computing, it is of great interest to automatically adapt current CPU-focused programs to massively parallel programs that can run on the GPU, among several other parallel architectures. Some automatic parallelization techniques include the OpenMP [Dagum and Menon, 1998; Bertolli et al., 2014] and OpenHMPP [Andión et al., 2016] standards. The task of automatically adapting programs, however, depends heavily on the independence of memory regions used by the relevant functions, which are candidates for automatic parallelization. Our techniques can be applied to disambiguate function pointer arguments of relevant functions. For such functions, if we can prove pointer independence, then we could transform them into kernels – functions directly executed by the GPU.

2.6 Final Thoughts

In this chapter we have reviewed the literature that is related to compiler optimizations in general, and points-to analysis in particular. In this work we shall augment this field, e.g., points-to analysis, with a new element, the Restrictification technique.

In the next chapter we show an example of the pointer disambiguation problem and how we tackle it using three different methods. First, two functions with pointer arguments are presented, and we discuss how aliasing can affect performance and correctness of their optimized versions. Then, the Restrictification methods are introduced: the purely static approach, and the two methods based on region analysis.

Chapter 3

Overview

To explain our contributions, we shall use two functions seen in Figure 3.1: `sumTillZero` and `sumTillSize`. Before we proceed in our dissertation, we would like the reader to consider the following question: under which circumstances can we transform these functions into the optimized versions seen in Figure 3.2? The optimization that we use in this example is *scalar promotion*: we are replacing the store operations on lines 12 and 18 of Figure 3.1 by register assignments.

It's not always possible to do this optimization, because, even though this transformation is safe for the three calling contexts seen on lines 27-30 of Figure 3.1, these functions can still be invoked from outside the program. For instance, functions `sumTillZero` (Fig. 3.1) and `sumTillZeroOpt` (Fig. 3.2) produce different results when they receive the following actual parameters: $r = \{1, 2, 3, 0\}$ and $w = r + 1$. The former will yield $*w = 9$, whereas the latter will give us $*w = 8$. In other words, functions `sumTillZero` and `sumTillZeroOpt` are only equivalent if their arguments do not reference overlapping memory regions. Notice that this limitation would apply to any purely static pointer disambiguation approach, including inter-procedural points-to analysis combined with function specialization and cloning¹.

Purely Static Restrictification. Henceforth, we shall say that for each function of interest, we have its *original* and its *restricted* version. A function's version is deemed "restricted" when we assume that its arguments do not reference overlapping memory regions. We shall call the act of replacing a function by its restricted version *restrictification*. If we had access to both versions of each function, then we could call

¹This combination of inter-procedural points-to analysis plus cloning is used, for instance, by the `-qipa` option of the XLC compiler (see http://www-01.ibm.com/support/knowledgecenter/SSPSQF_9.0.0/com.ibm.xlcpp111.linux.doc/proguide/qipa_opts.html).

```

1 char* get_new_array(int size, char init) {
2     char* A = malloc(size);
3     for (int i = 0; i < size - 1; i++) {
4         A[i] = init;
5     }
6     A[size - 1] = 0;
7     return A;
8 }
9
10 void sumTillZero(char* r, char* w) {
11     while (*r) {
12         *w += *(r++);
13     }
14 }
15
16 void sumTillSize(char* r, char* w, int N) {
17     for (int i = 0; i < N; i++) {
18         *w += r[i];
19     }
20 }
21
22 int main(int argc, char** argv) {
23     char A1[argc], A2 = 0;
24     char* A3 = get_new_array(argc, 0);
25     char* A4 = get_new_array(1, 0);
26     char* A5 = get_new_array(1 + argc, 1);
27     sumTillZero(A1, &A2);
28     sumTillZero(A3, A4);
29     sumTillSize(A3, A4, argc);
30     sumTillSize(A5, A5 + argc, argc);
31 }

```

Figure 3.1. Our different analyses let us disambiguate all the pointer arguments passed to the two `sumTill*` functions.

the restricted code in every call site in which we can prove that aliasing does not happen among actual parameters. However, disambiguating pointers through standard static analysis is not always feasible. For instance, LLVM’s alias analyses, currently based on five different algorithms, can only disambiguate the arguments of `sumTillZero` in line 27 of Figure 3.1. These purely static approaches, currently available in LLVM, are not precise enough to disambiguate the arguments of the other functions, invoked at lines 28 and 30. Thus, in this example, we would be able to restrictify only the calling site at line 27 of Figure 3.1 using static alias analyses. Figure 3.3, line 8, shows the original `main` function after we rewrite the calling site to invoke the improved version of `sumTillZero`.

Restrictification via Forward Region Analysis. Alias analyses with no context sensitivity will not be able to determine that arrays `A3` and `A4` do not overlap each other.

```

1 void sumTillZeroOpt(char* r, char* w) {
2     int tmp = *w;
3     while (*r) {
4         tmp += *(r++);
5     }
6     *w = tmp;
7 }
8
9 void sumTillSizeOpt(char* r, char* w, int N) {
10    int tmp = *w;
11    for (int i = 0; i < N; i++) {
12        tmp += r[i];
13    }
14    *w = tmp;
15 }

```

Figure 3.2. Optimized versions of the functions seen in Figure 3.1. Notice that these functions are only equivalent to their original selves in the absence of aliasing.

This imprecision happens because these pointers share the same allocation site: line 2 of Figure 3.1. The static alias analyses available in mainstream compilers such as LLVM or gcc are context-insensitive; hence, they suffer from this limitation. However, it is still possible to invoke the optimized function `sumTillZeroOpt` at line 28 of Figure 3.1 – provided that we can test for aliasing at the calling site. Arrays `A3` and `A4` will not alias if the region that can be addressed from these two base pointers do not overlap. If we let the size of `A3` be S_3 , and the size of `A4` be S_4 , then we have an absence of aliasing if $A3 + S_3 \leq A4 \vee A4 + S_4 \leq A3$. There are several ways to infer the values of S_3 and S_4 . In this dissertation, we experiment with a forward region analysis which produces results similar to those produced by the techniques introduced by Bodík et al. [2000] and Nazaré et al. [2014]. Figure 3.3, lines 10-14, shows the result of our optimization, enabled by forward region analysis. As we see in said figure, a runtime check (line 11) determines when it is safe to invoke the optimized version of `sumTillZero`.

Restrictification via Backward Region Analysis. The previous technique, based on a forward region inference analysis, is not able to disambiguate the pointers passed as parameters to `sumTillSize` at line 30 of Figure 3.1. On the other hand, by inspecting the body of said function (Fig. 3.1, lines 16-20), we know that aliasing is never possible. This inspection, be it performed automatically or manually, uses a *backward analysis*: we check how the array is used and assume that its size is given by the maximum range of any variable used to index it. In our example, a backward analysis reveals that the size of `r` is N and the size of `w` is 1 byte. Therefore, as long as $r + N \leq w \vee w + 1 \leq r$, no aliasing is possible within the body of `sumTillSize`. The technique that we shall

```

1 int main(int argc, char** argv) {
2   char A1[argc], A2 = 0;
3   char* A3 = get_new_array(argc, 0);
4   char* A4 = get_new_array(1, 0);
5   char* A5 = get_new_array(1 + argc, 1);
6
7   // PSA:ok, FRI:ok, BRI:no Section 3.1
8   sumTillZeroOpt(A1, &A2);
9
10  // PSA:no, FRI:ok, BRI:no Section 3.2.1
11  if (A3 + argc ≤ A4 || A4 + argc ≤ A3)
12    sumTillZeroOpt(A3, A4);
13  else
14    sumTillZero(A3, A4);
15
16  // PSA:no, FRI:ok, BRI:ok Section 3.2.2
17  sumTillSizeBri(A3, A4, argc);
18
19  // PSA:no, FRI:no, BRI:ok
20  sumTillSizeBri(A5, A5 + argc, argc);
21 }

```

Figure 3.3. Invocation sites seen earlier in Figure 3.1 rewritten to accommodate our optimization. Our approaches are named as follows: **PSA** - purely static approach (Sec. 4.1); **FRI** - forward region inference (Sec. 4.2.1); **BRI** - backward region inference (Sec. 4.2.2). If the approach is able to restrictify function arguments at a given call site, then we mark it with *ok*, otherwise we mark it with *no*.

describe in Section 4.2.2 performs this kind of analysis and transformation. In this example, that approach produces a new version of `sumTillSize`, which we show in Figure 3.4. Backward analysis has one advantage: it does not require whole-program analysis. Instead, it obtains information exclusively from syntax available within the target function. Its disadvantage is the fact that it can only analyze data structures accessed by variables whose integer ranges are bounded, such as `i` in Figure 3.1, lines 17-18.

All the three techniques are implemented at the Intermediate Representation of the program. As discussed in Chapter 2, this is the representation output by the compiler frontend. Using our methods at such stage enables us to profit from the multitude of analyses and optimizations that LLVM’s optimizer contains, which would not be available at the other stages, i.e. Abstract Syntax Tree or Code Generation.

Also, with regards to code correctness, it’s important to state that our methods do not fix or circumvent eventual programming mistakes made by a human. In that sense, Restrictification is a program optimizer, not a tool to find bugs in code. As long as the input program is correct in the first place, the optimized version will remain

```

1 void sumTillSizeBri(char* r, char* w, int N) {
2     if (r + N ≤ w || w + 1 ≤ r) {
3         // See Figure 2, lines 9-15
4         sumTillSizeOpt(r, w, N);
5     } else {
6         // See Figure 1, lines 16-20
7         sumTillSize(r, w, N);
8     }
9 }

```

Figure 3.4. Version of `sumTillSize` optimized by our backward analysis of Section 4.2.2.

correct after being restrictified.

3.1 Final Thoughts

This chapter introduced our three techniques to disambiguate pointer arguments of functions. We showed how the absence of aliasing enables function optimizations that would otherwise make the program incorrect.

The next chapter fully describes the three Restrictification methods we developed. We start with the purely static approach, which relies on static alias analyses present in the LLVM compiler to disambiguate pointers. The dynamic linkage extension, which enables this static method to work on shared libraries, is also discussed. Then we move on to the two hybrid techniques, which run static analyses on the code to then produce runtime disambiguation checks. First, we depict the forward region inference, which uses the sizes of allocated memory regions to insert runtime checks. Lastly, we discuss the backward region inference, which analyzes the body of the function to extract information about the access ranges of each pointer argument.

Chapter 4

Restrictification of Function Arguments

The three different approaches that we use to restrictify function arguments provide different tradeoffs, which we summarize in Figure 4.1. The purely static approach of Section 4.1 yields more precise results if we are able to infer aliasing in an interprocedural fashion; however, in this work, we limit ourselves only to the alias analyses available in LLVM, all of which are intraprocedural. The two approaches discussed in Section 4.2 generate code that decides, at runtime, when it is possible to invoke the restricted version of a function. This decision is based on the arguments passed to the function. The approach of Section 4.1, on the other hand, cannot afford such flexibility: either the restricted function or its original version is always invoked at a given call site. As a consequence, the purely static approach does not involve any runtime overhead, unless we link our code against dynamic libraries, as we explain in Section 4.1.1. In the rest of this section, we discuss each approach.

Section	WPI	Ch.Call	Ch.Fun	Dyn
4.1	Yes	Yes	No	No
4.2.1	Yes	Yes	No	Yes
4.2.2	No	No	Yes	Yes

Figure 4.1. A comparison between the different pointer disambiguation approaches that we use in this dissertation. **WPI:** Whole-program Information affects precision. **Ch.Call:** involves inserting computation at call sites. **Ch.Fun:** involves inserting computation in the target function. **Dyn:** determines, based on runtime values, when it is safe to call optimized code.

4.1 Static Restrictification

As previously mentioned, the purely static approach to the restrictification of function arguments resorts to traditional points-to analysis to identify invocation sites in which the arguments passed to the callee do not alias each other. Whenever this is the case, we replace a function with its restricted clone. Compiler-related literature describes a plethora of different algorithms to solve alias analyses. In this project, we have used the six different analyses available in LLVM 3.7 to disambiguate arguments of pointer type. These analyses are described below:

- **basic-aa**: this points-to analysis uses several known facts to disambiguate pointers. For instance, none of the following pairs of pointers can alias each other: different fields within a structure, different global variables, different local variables, memory allocated in the stack and in the heap, etc. All these checks run in $O(1)$. In spite of being one of LLVM’s simplest implementations of alias analysis, **basic-aa** is one of the most effective.
- **type-based-aa**: this analysis relies on the fact that the C99 standard forbids pointers of different types from referencing overlapping memory regions.
- **globalsmodref-aa**: this analysis uses the fact that global variables that don’t have their address taken cannot alias other pointers.
- **objc-arc-aa**: specially developed for Objective-C, this implementation uses information on the language’s custom Automatic Reference Counting to better provide alias analysis results. All programs tested in this project, however, are written in C or C++, making this option not viable.
- **scoped-noalias**: alias analysis with scope support. Memory instructions, such as loads and stores, can be associated to scopes, which are later used by this analysis to answer aliasing queries. From the module’s documentation¹, “When evaluating an aliasing query, if one of the instructions associated has a set of noalias scopes in some domain that is a superset of the alias scopes in that domain of some other instruction, then the two memory accesses are assumed not to alias”.
- **scev-aa**: this analysis uses scalar evolution expressions [Grosser et al., 2012, p.18] to solve alias queries. Its main target is the disambiguation of pair of pointer operations within a single iteration of a loop, by statically inferring their

¹http://llvm.org/docs/doxygen/html/ScopedNoAliasAA_8cpp_source.html

access distance. For instance, this analysis would be able to tell that the access operations in `a[i]` and `a[i+3]` can never alias each other in the same iteration of a loop, as long as `i` is an induction variable whose value does not vary within a single iteration.

- **cfl-aa**: this analysis implements a context-insensitive alias analysis based on context-free grammars (CFL). This algorithm is implemented after Zheng and Rugina [2008] and Zhang et al. [2013]. More recently, this implementation also includes, to some extent, both Andersen’s alias analysis [Andersen, 1994] and Steensgaard’s [Steensgaard, 1996]. Given, however, that Andersen’s leads to higher execution costs due to its complexity, developers of LLVM chose to cut its reach to reasonable limits. Namely, their implementation goes no further than one level of context-sensitiveness.

The precision of all these analyses is cumulative: if any of them can prove that two pointers are non-aliases, then they are considered non-aliases. Thus, by invoking more or less analyses, the LLVM user trades precision for speed.

The main advantage of the purely static restrictification approach is the fact that it does not incur any runtime overhead. If we can prove the absence of aliasing among the arguments of a function f at a given invocation site s , then we are free to invoke the restricted version of f . In this case, we do not have to insert any dynamic check at s , as the approach of Section 4.2.1 forces us to do. Nor do we need to insert guards in the restricted function, as the technique that we shall discuss in Section 4.2.2 requires. On the other hand, the purely static restrictification of functions is also the most inflexible of our techniques: either we always use the restricted version of a function, at a given invocation site, or we never use it. In other words, we cannot check for no-aliasing based on runtime data, in hopes to call the restricted function more often. Consequently, the technique that we discussed in this section is only applicable in one of the four invocation sites of Figure 3.1, i.e., the call at line 27. In this example, we know that different local variables cannot alias each other; hence, we are free to invoke the restricted version of `sumTillZero` in line 27 of Figure 3.1, as Figure 3.3, line 8, shows.

4.1.1 Dynamic Linkage

We generate optimized clones at compilation time, and replace call sites at link time. This approach has one limitation: it does not work when the function to be restricted f belongs to a dynamically linked library. If this is the case, when code is generated for a caller g , the compiler will not know that f has an optimized version. Going

back to our example of Figure 3.1, to replace the call of `sumTillZero` at line 27 by a call to `sumTillZeroOpt` (Figure 3.2), we need to know, when linking the executable for function `main`, that `sumTillZeroOpt` exists. This shortcoming would prevent us from applying our optimization in programs meant to be used as shared libraries, for example.

To overcome this limitation, the static restrictification of functions happens in two phases. The first phase is called *cloning*: at compilation time, we produce a restricted version of every eligible function that is not marked `static`. A function is deemed eligible if it receives two or more arguments having pointer type. Because a non-static function f can be called outside the module m where f is implemented, restrictification happens even if f is not used within m . The second phase is called *replacement*: at linking time, we replace original function calls by their optimized versions whenever their definition is available and the arguments don't alias. Furthermore, for the remaining functions whose optimized versions aren't known yet, we surround every call site with guards. These guards contain code to check, at execution time, if the optimized version of the function is available at any library loaded dynamically. The client will invoke the optimized clone whenever possible, or the original function otherwise. We have implemented this runtime verification via the `dlsym`² system call, which is provided by any Unix-like operating system, while similar routines also exist for Windows. This system call receives one string as argument – the name of a function – and searches all shared libraries for this routine. Figure 4.2 shows the code that we produce for our running example.

```

1 int main(int argc, char** argv) {
2   char A1[argc], A2 = 0;
3   ...
4   void *opt = dlsym(RTLD_NEXT, "sumTillZeroOpt");
5   if (opt) {
6     void (*sumTillZeroOpt)(int*, int*) = opt;
7     sumTillZeroOpt(A1, &A2);
8   } else {
9     sumTillZero(A1, &A2);
10  }
11  ...
12 }
```

Figure 4.2. Snippet of function `main`, seen in Figure 3.1, rewritten after the replacement phase used in purely static restrictification.

These two phases – cloning and replacement – are independent. If only replacement is performed, then the program still works normally. When the optimized version of a

² <http://linux.die.net/man/3/dlsym>

function f does not exist, the `dlsym` call returns null. In this case, the original version of f is used. At any point in time, the optimized shared library that contains the restricted clone of f can be linked to the client binary. Once this happens, the client program will start using the restricted version of f automatically and seamlessly. One great advantage of this feature is that the library's code doesn't need to be compiled and linked at the same time as the client's for the optimizations to be applied. However, this approach has one shortcoming: name collisions. The `dlsym` call searches functions by name, and the cloning procedure adds to each cloned function's name a string that is unlikely, though not impossible, to produce collisions with the remaining names in the program. Nevertheless, this possibility exists. Also, since `dlsym` performs searches only inside shared libraries, this method doesn't support the optimization of calls to functions that are defined in static libraries.

4.2 Hybrid Restrictification

We call *hybrid restrictification* the combination of static and dynamic techniques to restrictify function arguments. Static analysis is used to infer symbolic expressions that denote the bounds of memory regions. These bounds let us create checks, which will determine, at runtime, when it is safe to assume the absence of aliasing among function arguments. we have tested two different static analyses to infer memory bounds. Henceforth, these approaches will be called *forward* region inference and *backward* region inference. These two analyses are bootstrapped by a *symbolic range analysis*, which produces a map R from every integer variable v to $[v_\downarrow, v_\uparrow]$. We let $v_\downarrow$ be the symbolic lower bound of v , and $v_\uparrow$ be its symbolic upper bound. Programming languages literature contains several implementations of such an analysis [Blume and Eigenmann, 1995; Nazaré et al., 2014; Rugina and Rinard, 2005]. This analysis gives us, for instance, that $R(\mathbf{i}) = [0, \mathbf{size} - 1]$ at line 4 of Figure 3.1.

To represent ranges, we use a set S of *symbolic expressions*. A *symbol* is any name, in the program text, that cannot be reconstructed as function of other names. In our case, symbols are: (i) values returned from functions whose body is not present in the target code; or (ii) values created by load instructions that might dereference more than one memory location. A symbolic expression is defined by the grammar below, where s is a symbol and $n \in \mathbb{N}$:

$$\begin{aligned}
 E ::= & n \mid s \mid \min(E, E) \mid \max(E, E) \mid E - E \\
 & \mid E + E \mid E/E \mid E \bmod E \mid E \times E \\
 & \mid -\infty \mid +\infty
 \end{aligned}$$

We augment S with a partial order, which is defined only between symbolic expressions that are integer constants, e.g.: $-\infty < \dots < -2 < -1 < 0 < 1 < 2 < \dots < +\infty$. There is no ordering between expressions involving different symbols. The product of two instances of this partially ordered set defines a semi-lattice $(S^2, \subseteq, \cup, \emptyset, [-\infty, +\infty])$. The join operator “ $\sqcup$ ” is given by:

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

The least element $\emptyset$ is such that:

$$\emptyset \sqcup [l, u] = [l, u] \sqcup \emptyset = [l, u]$$

and the greatest element $[-\infty, +\infty]$ gives us that:

$$[-\infty, +\infty] \sqcup [l, u] = [l, u] \sqcup [-\infty, +\infty] = [-\infty, +\infty]$$

Neither the forward nor the backward region inference analyses are contributions of this dissertation: they have been described in previous work. Backward region inference has been used to check the complexity of code [Piccoli et al., 2014] and to disambiguate pointers in *doall* loops [Rugina and Rinard, 2005]. Variations of forward region inference have been used in previous work to remove bound checks in Java [Bodík et al., 2000] and in type-safe C code [Nazaré et al., 2014]. Nevertheless, we claim as a contribution the use that we make of them, for, to the best of our knowledge, they have not been used to generate tests that disambiguate pointers at function level. Section 2 provides a deeper comparison of our work and the previous literature.

4.2.1 Forward Region Inference

Forward region inference discovers the sizes of arrays based on the allocation sites visible within the program code. In other words, the arguments of memory allocation routines, such as `malloc`, `valloc`, `calloc`, and `realloc` are bound to the arrays that said routines produce. In this project we have experimented with a very simple version of forward region inference, which is implemented by the transfer functions in Figure 4.3. These transfer functions are standard in symbolic range analysis. Finding their fixed point can be done in time linear on the number of functions, because widening (∇) ensures quick termination. The forward region inference analysis finds, for each pointer variable v , the *maximum addressable offset* $F(v)$ that can use v as a base pointer. Thus, if we say that $F(v) = N$, then we know that any address in the

range $[v + 0, v + N]$ may dereference a region pointed to by v . Notice that this is a fundamentally different abstract semantics than the one adopted by previous forward region inference analyses [Blume and Eigenmann, 1995; Bodík et al., 2000; Nazaré et al., 2014]. In that case, the authors were interested in detecting bugs, so their meet operation was *minimum*. We are interested in finding conservative memory bounds; hence, our meet operation is *maximum*.

$$\begin{aligned}
v_1 = \text{malloc}(v_2) &\Rightarrow F(v_1) = R(v_2)_\uparrow - 1 \\
a = b &\Rightarrow F(a) = F(b) \\
v_1 = v_2 + n &\Rightarrow F(v_1) = F(v_2) - n \\
\text{with } n \in \mathbb{N} & \\
v_1 = v_2 + v_3 &\Rightarrow F(v_1) = F(v_2) - R(v_3)_\uparrow \\
\text{with } v_3 \text{ scalar} & \\
v = \phi(v_0, v_1) &\Rightarrow F(v) = \max(F(v_0), F(v_1)) \\
R_1 \nabla R_2 = [l, u], \text{ where } &\begin{cases} l = R_{1\downarrow} & \text{if } R_{1\downarrow} = R_{2\downarrow} \\ l = -\infty & \text{otherwise} \\ u = R_{1\uparrow} & \text{if } R_{1\uparrow} = R_{2\uparrow} \\ u = +\infty & \text{otherwise} \end{cases}
\end{aligned}$$

Figure 4.3. Forward Region Inference Analysis.

The analysis is run on programs in the Static Single Assignment (SSA) form [Cytron et al., 1991]. This representation ensures that the region information associated with a variable is invariant along the entire live range of that variable. The ϕ -functions, typical of the SSA format, let us join information produced at different program points. Many arrays are defined in functions other than the procedure where they are used; thus, ideally, forward region inference should run interprocedurally. we use ϕ -functions to implement interprocedurality. To explain this last point, let's assume that the target program contains a function declaration such as $f(\dots, v_i, \dots)$, and n different calls such as $f(\dots, a_i^1, \dots)$ and $f(\dots, a_i^n, \dots)$, where the i^{th} argument is of pointer type. If that is the case, then we model the transfer of information from actual to formal arguments via the following n -ary ϕ -function: $v_i = \phi(a_i^1, \dots, a_i^n)$. Similarly, we use simple copies, i.e., $v_1 = v_2$ to model the transfer of information due to the return operation.

Figure 4.4 shows the result of applying our forward range estimation on the program seen in Figure 3.1. We notice that the call to `malloc` in line 2 of Figure 3.1 is

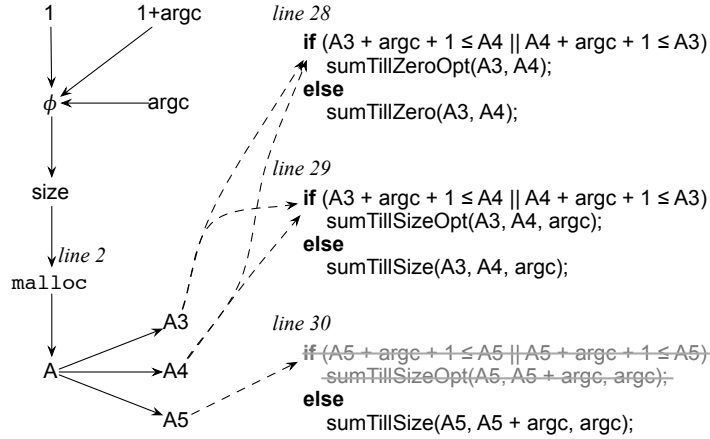


Figure 4.4. The result of applying the forward region inference analysis on the program seen in Figure 3.1.

reached by three different definitions of `size`. Thus, our meet operation gives us that the most conservative estimate on the upper bound of array `A` is $\max(1, 1 + \text{argc}, \text{argc})$, which we can simplify to $\text{argc} + 1$, given the knowledge that $R(\text{argc})_{\downarrow} \geq 0$ ³. These bounds let us modify the call sites at lines 28 and 29 of Figure 3.1, as to check memory ranges at runtime. The rewritten code is shown at the right side of Figure 4.4. The check inserted at the last call of `sumTillSize` is trivially false, and we do not insert it. Notice that the size of a check is quadratic on the number of pointers passed as arguments of the target function.

4.2.2 Backward Region Inference

The forward region inference analysis of Section 4.2.1 is a whole-program analysis. This interprocedurality might be a limitation if we need to analyze code that is only partially available. To mitigate this problem, we have used a backward region inference analysis, which is intraprocedural by nature. This analysis maps every variable v , of pointer type, to a symbolic expression $B(v_1)$, which has the same semantics as F (introduced in Section 4.2.1), i.e., a conservative bound on the maximum valid offset that can use v_1 as base address⁴. To infer the bounds of arrays, we create, for each occurrence of a memory access $v_1[v_2]$, a constraint $B(v_1) \geq R(v_2)_{\uparrow}$. Notice that this analysis is flow-insensitive, and the relative order in which different dereferences of v_1 appear within a function are immaterial.

³In Section 5.1.2.2.1 of the ISO/IEC 9899:2011 C standard, “The value of `argc` shall be nonnegative”.

⁴Even though $F(v)$ (Sec. 4.2.1) and $B(v)$ (Sec. 4.2.2) have the same semantics, we shall keep these two maps to avoid confusion when referring to the different region inference analyses.

As an example, the backward region inference, when applied to the function `get_new_array`, seen in Figure 3.1, would give us the following two constraints: (i) $B(A) \geq R(i)_\uparrow = \text{size} - 2$, due to the access at line 4, and (ii) $B(A) \geq R(\text{size})_\uparrow - 1 = \text{size} - 1$, due to the access at line 6. The union of these two constraints gives us that the largest valid offset from pointer `A`, within function `get_new_array`, cannot be larger than $A + \text{size} - 1$. Similarly, the backward region inference analysis gives us two constraints for function `sumTillSize`, in Figure 3.1: (i) $B(r) \geq R(N)_\uparrow - 1$ and (ii) $B(w) \geq 0$.

```

1 void countCh(int** A, char* B, char* C
2             int M, int N, int X, int Y) {
3     int bA = max(N*M*4, (X*N+Y)*4);
4     int bB = M;
5     int bC = N;
6     if ((A + bA ≤ bB || B + bB ≤ bA)
7         && (A + bA ≤ bC || C + bC ≤ bA)
8         && (B + bB ≤ bC || C + bC ≤ bB)) {
9         // call optimized version of countCh
10    } else {
11        // original code:
12        for (int i = 0; i < M; i++) {
13            for (int j = 0; j < N; j++) {
14                A[i*N+j] += B[i] > C[j] ? 1 : 0;
15            }
16        }
17        return A[X*N+Y];
18    }
19 }

```

Figure 4.5. Example of code produced by our backward region inference analysis. Memory accesses are marked grey. Arrows indicate the constraints produced for matrix `A`.

Figure 4.5 shows an example of code produced by our backward analysis. Every memory access within the target function gives us constraints, as long as we can infer the bounds of this access using a Scalar Evolution analysis [Bachmann et al., 1994]. Whenever we have two or more constraints for the same pointer p , we use the maximum of them to bound p . In this example, we have four memory accesses, two of them to matrix `A`. The memory bounds that our analysis finds are given on lines 3-5 of the example. The tests that we generate can be seen on lines 6-8. Notice that the backward analysis can only infer bounds to memory access functions that use variables bound to known range expressions. It cannot, for instance, analyze expressions such as `a[b[i]]`.

Moreover, we cannot infer bounds of a pointer p , passed as argument of a function f , if p might escape the scope of f into another function g . In practice, we use LLVM’s escape analysis to identify this kind of situation. Yet, our technique is not limited to scientific codes; rather, it is widely applicable. In Section 5 we show that it can handle thousands of benchmarks present in the OpenCV’s test suite.

4.3 Code Generation

Both the forward approach of Section 4.2.1 and the backward approach of Section 4.2.2 produce bound comparisons that, at runtime, check for the possibility of aliasing among pointers. Given two pointers p_1 and p_2 , of type T_1 and T_2 , and bounds M_1 and M_2 , we generate the following guard:

$$p_1 + M_1 \times \text{sizeof}(T_1) \leq p_2 \text{ or } p_2 + M_2 \times \text{sizeof}(T_2) \leq p_1$$

Whenever the guard is true, we divert the program flow to code that can be optimized under the assumption that p_1 and p_2 cannot refer to overlapping memory regions. If a function contains n different pointers used as arguments, then we can generate up to $n \times (n - 1)/2$ such guards for that function, using either the backward or the forward approach. In other words, we generate a *quadratic* number of comparisons per number of memory accesses within the optimized region. Nevertheless, the overhead of the dynamic checks is negligible, as we shall explain in Section 5.

The difference between these two techniques, in terms of code generation, is that the tests of Section 4.2.1 are placed at the function call site, whereas the tests of Section 4.2.2 are placed within the function, at its entry point. This fact is an advantage of the backward approach, because it allows us to apply it independently per program module. In other words, we can link our optimized code against non-optimized binaries without the need to recompile those binaries. Currently, we only generate tests if the symbolic bounds of a variable are expressions of variables *alive* at the program point where the guard will be placed. We say that a variable v is alive at a program point i if it is used at a program point i' , there is a path from i to i' , and v is not redefined along this path. Global variables are alive everywhere in the program code.

A Note on Safety. Our hybrid analyses might generate invalid checks if the target program already presents out of bounds accesses. Let’s imagine, for instance, that function `countCh`, in Figure 4.5, receives a size `M` less than the actual number of lines of `A`. This program has undefined behavior, for an out-of-bounds access would happen

at line 14 of Figure 4.5. This programming error might cause our checks to branch to the optimized code, when pointers indeed alias. Our analyses are used to optimize programs, and we believe that it is meaningless to apply it on incorrect code. Incorrect code is not an issue for the forward analysis. Because it keeps track of the actual expressions used to allocate memory, it only generates tests when such expressions – which denote the correct bounds of memory – are available.

4.4 Final Thoughts

In this chapter we went through the three Restrictification techniques in depth. We showed the static method that uses alias analyses to disambiguate pointers. We talked about the dynamic linkage extension, which enables the static method to work on shared libraries. The rest of the chapter was dedicated to the two hybrid methods. These approaches combine static analyses to gather relevant information on pointer arguments with dynamic runtime checks that are inserted in the program.

The chapter that follows assesses how much Restrictification improves performance of programs. We start with an overview of the benchmarks we used to evaluate our techniques. Then we proceed to evaluate Restrictification under several angles. We aim to answer the following questions about Restrictification: (i) What is the coverage of each technique? (ii) How are compiler optimization passes affected by our methods? (iii) Given that the techniques use function cloning, how is the code size affected by that? (iv) What is the performance gain we achieve after Restrictification? To demonstrate the strength of our approaches, the final section describes Restrictification on the OpenCV library, a widely used computer vision framework.

Chapter 5

Evaluation

To validate our techniques, we have implemented them on the LLVM compilation infrastructure [Lattner and Adve, 2004], version 3.7. In the first part of this evaluation section, we have tested our implementation on four different benchmarks suites: (i) BitBench, composed of different Bit Stream manipulation algorithms, (ii) MiBench, containing a number of applications for embedded systems, (iii) Shootout-C++, which encompasses several scientific and mathematical algorithms, and (iv) Stanford, with common general-purpose routines, such as sorting, permutation, and puzzle-solving. These benchmarks are bundled in the LLVM Test Suite¹ and details about their structure are presented in Table 5.1. Later in this section, we present the evaluation of runtime improvements produced by our techniques on the OpenCV library. These benchmark suites were chosen because they embrace different programming purposes, from basic algorithms to more complex routines, such as scientific functions and computer vision methods. Altogether, we believe they represent a good set of C/C++ programs.

All the benchmarks used in this evaluation have been compiled and linked with the -O3 optimization level of LLVM prior to being transformed by our techniques. At this level, the compiler performs roughly 60 analysis and optimization passes on the target programs. Among these, LLVM applies function inlining, which is one of our direct competitors. Therefore, improvements seen in this section are on top of the highest degree of optimization in LLVM, which includes inlining. The numbers presented here were obtained using an Intel Xeon 2.0GHz machine, with 16GB of RAM, running Linux version 3.13.0. Run time numbers represent the average of at least 10 executions.

In the following subsections we explain the effectiveness of our methods by investigating the different forms in which they can affect the compilation process. We aim to answer the following questions about Restrictification: (i) What is the coverage

¹<http://llvm.org/docs/TestingGuide.html>

of each technique? How many functions are they able to optimize? (ii) How are compiler optimization passes affected by our methods? What is the degree of improvement on their numbers? (iii) Given that the techniques use function cloning, how is the code size affected by that? (iv) What is the performance gain we achieve after Restrictification (in terms of runtime)?

Benchmark	Size	Functions	Eligible	Calls
BitBench	1,167	28	11	12
MiBench	357,159	1,395	525	1,505
Shootout-C++	25,701	2,482	720	1,082
Stanford	2,249	80	7	10
OpenCV	2,150,492	97,123	37,523	94,354

Table 5.1. Benchmark descriptions: size (number of LLVM Intermediate Representation instructions), number of functions overall, number of functions eligible for cloning (take two or more pointer arguments), number of call sites where eligible functions are invoked.

5.1 The Reach of Restrictification

Benchmark	PSA	FRI	BRI
BitBench	37%	6%	17%
MiBench	15%	1%	72%
Shootout-C++	26%	4%	44%
Stanford	60%	10%	63%

Table 5.2. Reach of our different disambiguation methods. **PSA**: number of call sites optimized by the Purely Static Approach. **FRI**: number of call sites instrumented when using Forward Region Inference. **BRI**: number of functions guarded with dynamic checks by the Backward Region Inference.

Table 5.2 shows the reach of our three disambiguation techniques when analyzing and instrumenting the four suites of benchmarks described previously. The numbers shown are for each suite as a whole. For the Purely Static method (Section 4.1), we present the relative number of call sites optimized out of all points in the program that invoke functions passing pointers as arguments. For MiBench, we were able to optimize 15% of the call sites. In Stanford, the best relative result, we could disambiguate 6 out of 10 calls (60%). When considering all four suites, the average reach of the static method was 18%. We also note that 18% of the functions that receive pointer as parameters had all their call sites optimized, i.e., their original – unoptimized – version could be removed from the program after linking.

It is known that actual aliasing is rare at runtime [Hackett and Aiken, 2006], i.e., it is uncommon for different pointers in a program to reference the same memory region during execution. For this reason, the reach of our hybrid techniques can be approximated by the number of functions for which dynamic alias checks can be generated, as such, checks tend to pass at runtime. For our forward region inference method (Section 4.2.1), we present the number of call sites instrumented with dynamic checks. As seen in the table, the numbers for this approach were not as significant as the ones presented for the Purely Static method, with our best result being a 10% reach in Stanford. We attribute such low numbers to the fact that symbols used as argument of dynamically memory allocation functions usually are not alive at the call sites where the arrays created by these functions are passed as parameters. To generate guards, our forward analysis needs to have access to these symbols at the place where the tests must be constructed. As a further experiment, we have tried to identify pairs formed by arrays and integers that represent the size of these arrays among the arguments of functions. Only 4% of these pairs denote such size relations.

Our backward region analysis, on the other hand, instruments function entries rather than call sites and does not require whole-program information to work. To measure the reach of this technique, we present the relative number of functions that receive pointers as inputs and could be guarded with dynamic pointer tests. The best result for this method was in MiBench, where we could optimize 72% of target functions. For BitBench, our rather shy results stem from its memory access patterns, from which it isn't possible to extract suitable scalar evolution expressions that enable our backward region analysis to generate dynamic checks.

Looking at these numbers, we can state that the FRI unfortunately did not fare well for the reasons already presented. Both PSA and BRI, on the other hand, showed that they are able to provide good coverage of function call sites, with BRI having the edge. In the next section, we will inquire how compiler optimizations behave after we create the function clones annotated with the `restrict` keyword.

5.2 Strengthening Code Optimizations

Lack of precise knowledge about dependencies between memory regions can greatly undermine code optimizations. Such an effect is particularly observed in transformations which aim to eliminate redundant computation and exploit parallelism between independent portions of code. Notorious examples of such optimizations are Loop Invariant Code Motion (LICM), Global Value Numbering (GVN), and Automatic

Loop Vectorization, all three currently available in LLVM. Therefore, more precise disambiguation techniques, such as the ones we propose in this work, are expected to considerably improve the reach of such program transformations.

Table 5.3 shows how our disambiguation methods improve program optimizations. *Locations promoted to registers* represents the number of memory operations that LICM could convert into less costly scalar instructions. We see that both our purely static and backward inference approaches delivered a significant improvement in this metric for the Stanford and MiBench suites. *Redundant loads/stores deleted* represent the number of memory access instructions that GVN could completely eliminate from the program’s code. In general, all three methods could improve one of these metrics to some extent, with the purely static and forward inference approaches being the most effective. *Vector instructions generated*, in turn, is the number of vector operations resulting from the combination of different independent instructions. This transformation is also known as Superword-Level Parallelization. An increase in this metric, such as the one that we observed in MiBench, also indicates an increase in the number of loops for which the compiler can prove independence between iterations, making them also suitable for more aggressive optimizations, such as automatic GPU code placement [Margiolas and O’Boyle, 2014].

Benchmark	Locations promoted to registers			Vector instructions generated		
	PSA	FRI	BRI	PSA	FRI	BRI
BitBench	0%	0%	0%	0%	0%	0%
MiBench	32%	0%	34%	63%	38%	63%
Shootout-C++	0%	0%	0%	0%	0%	0%
Stanford	200%	0%	200%	0%	0%	0%

Benchmark	Redundant loads deleted			Redundant stores deleted		
	PSA	FRI	BRI	PSA	FRI	BRI
BitBench	800%	800%	0%	0%	0%	0%
MiBench	4%	4%	4%	63%	77%	65%
Shootout-C++	0%	0%	0%	0%	0%	0%
Stanford	26%	18%	0%	0%	0%	0%

Table 5.3. Relative improvement of code optimizations when combined with our different disambiguation techniques. **PSA**: purely static approach. **FRI**: forward region inference. **BRI**: backward region inference.

The next experiment measures the code size expansion after Restrictification is performed. As said previously, this is an expected effect of our techniques given that we rely on function cloning to perform our optimizations.

5.3 Code Size Expansion

Benchmark	PSA	FRI	BRI
BitBench	8%	8%	0%
MiBench	9%	0%	74%
Shootout-C++	1%	0%	3%
Stanford	4%	0%	26%

Table 5.4. Relative increase in executable’s size when using each disambiguation method. **PSA**: purely static. **FRI**: forward region inference. **BRI**: backward region inference.

Table 5.4 shows the relative increase in the program’s size when using each of the three techniques. The numbers represent the ratio between the space occupied on disk, in bytes, of the generated binaries. The increase promoted by the purely static method is relatively small, under 10% in all cases, mainly due to its lower precision when compared, for instance, to the more aggressive backward inference method. The forward analysis does not increase executable size by a significant factor, except in BitBench. This result is due to the low coverage of call sites that the forward region inference achieves. Furthermore, if the original function is no longer called in the program, we can eliminate it; hence, keeping only the optimized function. The highest increase is observed when using our backward inference technique. Such a result comes from the fact that this method overcomes the major limitations faced by our other two approaches: it uses information contained only within the target function itself and it has a dynamic component that lets us evaluate pointers at runtime.

In the following section, we present a study on how Restrictification impacts runtime performance in terms of speedup. For that purpose, we show the ratios between the runtimes of optimized version and original version of programs.

5.4 Speedup

We measured how our techniques enable runtime improvements with the compiler optimizations that exist today in LLVM. As previously noted, these runtime numbers were obtained after performing the highest level of optimization provided by LLVM, -O3, which also includes function inlining, our most direct competitor. Figure 5.1 shows the ratio between the execution time of the optimized program and the run time of the original program for the first four benchmarks suites. Each bar gives the geometric mean of the run time ratios of all the programs in the corresponding benchmark.

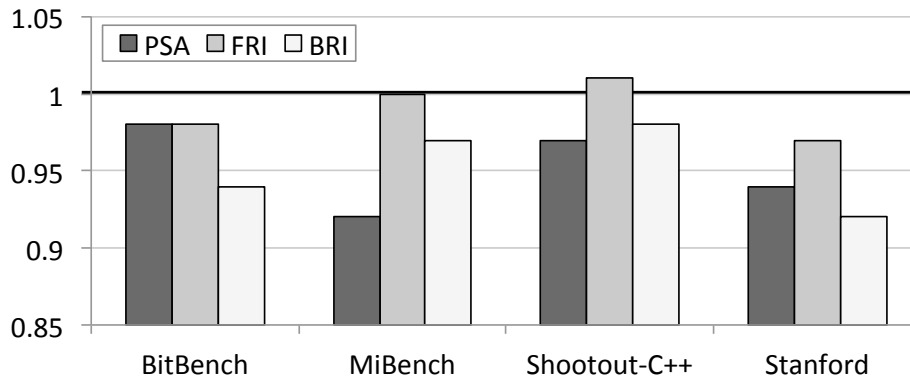


Figure 5.1. Execution time ratio of each benchmark suite for our three techniques (lower is better).

Even though our methods were able to improve the effectiveness of major program optimizations, the resulting runtime improvements were relatively small. In the best case we could speedup the MiBench and Stanford programs by 8% using the purely static and the backward inference approach, respectively. In the other cases, however, our gains were under 5%, with one specific slowdown in Shootout-C++ using the forward inference method. Figure 5.2 shows the runtime ratios for the programs in Shootout-C++ suite individually. We achieved mixed results in this benchmark: while there were some good improvements, such as in `ackermann` for the PSA (13%) and `nestedloop` for BRI (15%), we also had more than 10% slowdowns in three programs. At the vast majority of the programs, though, ratios were very close to 1. The slowdowns that we have observed were not caused by the overhead of the guards. Rather, they are the result of optimizations that still offer room for tuning. It is important to notice that our pointer disambiguation techniques enable optimizations, but they do not implement them: this task is left to the compiler.

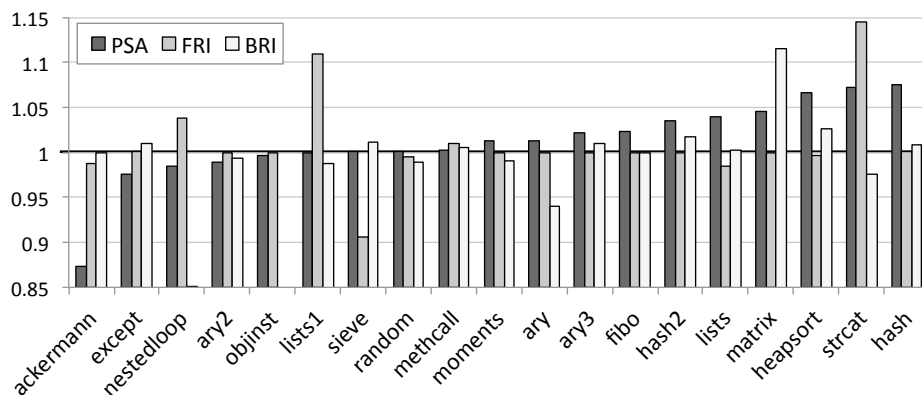


Figure 5.2. Execution time ratio for each benchmark in Shootout-C++, using our three methods (lower is better).

Thus we can conclude that the boost in compiler optimization numbers, seen in Section 5.3, doesn't necessarily translate into gains in runtime performance. In the majority of cases, optimized programs had their performance improved, but still very close to the original. Nevertheless, some benchmarks did experience good enhancement in performance. Moving to the next section, we will employ our techniques onto OpenCV, a major library for computer vision applications.

5.5 Testing Our Approach on Larger Programs

In order to investigate the effectiveness of our approach on large programs, we tested our pointer disambiguation methods on the widely used computer vision library OpenCV². OpenCV is organized as a set of shared libraries to be used by client applications, which are built separately. This represents a perfect scenario for the dynamic linkage extension of our static method, presented in Section 4.1.1, which provides client code access to optimized versions of library functions. In this scenario, pointer disambiguation via function inlining is not possible, since the definitions of core functions are inside the libraries, external to the client code. Here we present results for the PSA and BRI approaches, as they produced the best results for the benchmarks in Figure 5.2. Sadly, the disappointing numbers of the FRI method for the first four benchmarks suites did not justify using it on OpenCV. In addition, we investigate the effects of these two techniques when used in a complementary fashion: PSA is initially used to replace function calls at linking time through static analysis whenever possible. Then, the remaining calls are handled by BRI, inserting dynamic disambiguation checks.

Figure 5.3 shows the runtime ratios between the optimized and the regular versions of OpenCV. The OpenCV Benchmark Suite contains thousands of tests, which are bundled together with a simple infrastructure to run and time them all. These tests are grouped into 13 modules. In this experiment we chose two of them: `core` and `imgproc`, because they contain the largest number of benchmarks: 1,682 and 2,146 respectively, and also because they don't require non-free inputs, such as proprietary images or videos. The former measures performance of basic data structures, including dense multi-dimensional arrays and functions that are used across the entire library. The latter is composed mostly by image processing functions, such as filtering and transformations. Each point on the X-axes of Figure 5.3 represents a different test case, e.g., a whole program that calls OpenCV primitives. On each case, we highlight the number of benchmarks that we have speeded up with our approach. The tests in

²<http://www.opencv.org>

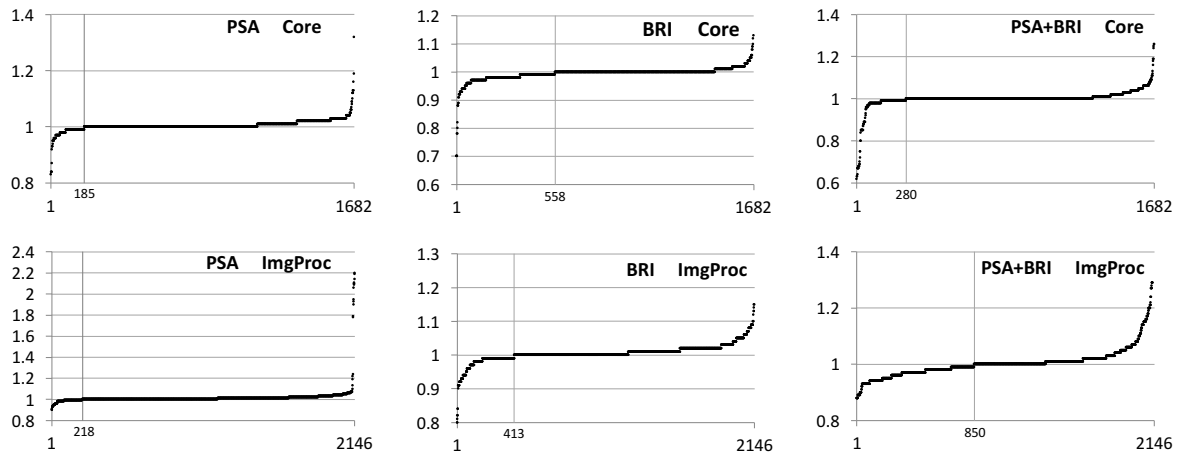


Figure 5.3. Execution time ratio for the `core` and `imgproc` test modules of OpenCV, using the PSA and BRI methods. Each point along the horizontal axis represents a different test. Lower is better. The vertical line in each chart marks the point where ratios become greater than or equal to 1.

each figure took about two hours to run.

The leftmost column corresponds to the purely static approach (PSA). In the `core` suite, 11% of the benchmarks had improvements in runtime, with individual speedups of as much as 1.20x. In the `imgproc`, 10% of the benchmarks showed speedups, with individual numbers as high as 1.11x. The center column of Figure 5.3 presents runtime ratios for the Backward Region Inference method (BRI). We notice improvements over the PSA: in the first benchmark suite of OpenCV, the BRI produced speedups as high as 1.72x (decrease in runtime of 42%); in the second, 19% of the benchmarks experienced speedups, with the highest decrease in runtime of 20% (speedup of 1.25x).

Combining Different Approaches. The third column shows ratios for the combined method, where PSA is run over programs in a first moment, and BRI is then used to catch disambiguation opportunities that weren't handled statically by PSA. On the `core` suite, 16% of the benchmarks had gains, of which the best gain gave us 39% of runtime decrease (speedup of 1.64x). If we compare solely the average runtime ratios (lower is better in this case) of the benchmarks that had gains (defined here by runtime ratio < 0.97 in the chart), we notice that for BRI there was a geomean ratio of 93%, while for PSA+BRI the geomean ratio is 81%, accounting for an improvement of 14% in the average. On the second suite, `imgproc`, 39% of the benchmarks experienced speedups, which is an evident step up compared to both PSA and BRI by themselves (10% and 19% respectively). The best ratio on this suite for PSA+BRI was 88% (speedup of 1.14x). There was, however, a peculiar result for the combined approach

on the `core` benchmark suite: the number of improved programs was actually smaller when compared to using the BRI alone. This is an odd outcome and we couldn't find a cause for such occurrence. As such, the investigation on what leads to this phenomenon remains as an open issue.

Based on the much larger number of OpenCV benchmarks that met improvements overall, we conclude that combining the static method, PSA, with the Backward Region Inference, BRI, is the most promising approach to optimize programs. It is still possible to obtain further gains with restrictification, if we tune our techniques more carefully. The use of improved cost models is left as future work, as we discuss in Chapter 6.

Understanding Slowdowns. A number of tests yielded slower execution times when using our disambiguation approaches. After a thorough inspection, we noticed that several slowdowns were caused by poorly tuned optimizations. Our techniques widen the reach of many optimizations, as seen in Table 5.3, but the new version of a program may sometimes be slower than its original version. Figure 5.4 (a) shows a loop extracted from one of the tests in the `core` module. In this fragment, the developer manually unrolled the array copy operation in order to increase its performance. Without knowing if `src` and `dst` can point to the same place in memory, LLVM's memory optimization passes are not able to modify this code. When the library is compiled using our disambiguation techniques, LLVM becomes aware of the absence of aliasing and can perform more aggressive transformations. In particular, the loop in the Figure is subject to an optimization called *Superword-Level Parallelism*. This transformation combines different load and store operations in single vector instructions, generating the loop seen in Figure 5.4 (b). When testing in a development version of LLVM 3.7, however, the optimized version of the loop is around 1.3x slower than its original, manually unrolled, counterpart.

This anecdote illustrates the well-known fact that the code generated by compiler optimizations is not always faster than its original version. Nevertheless, we also have our share of blame in the slowdowns. When looking at the results for the Purely Static method for the `imgproc` module (leftmost column, second row of Figure 5.3), we see that a very small subset of the tests (13 out of 2146) were subject to slowdowns of around 2x. We attribute these slowdowns to extra calls to `dlsym`, a hook used by client applications to find the alias-free version of a function. All this said, we reinforce that these benchmarks constitute a very tiny portion of test cases. Moreover, approaches like BRI and FRI can be used to avoid such slowdowns. We did not perceive large overheads with the other approaches: as a simple experiment, we have added the `restrict` modifier to the arguments of functions in OpenCV, and this new version was

```

1 for (int i = 0; i <= len - 4; i += 4 ) {
2   int s0, s1;
3   s0 = dst[i] + src[i];
4   s1 = dst[i+1] + src[i+1];
5   dst[i] = s0; dst[i+1] = s1;
6   s0 = dst[i+2] + src[i+2];
7   s1 = dst[i+3] + src[i+3];
8   dst[i+2] = s0; dst[i+3] = s1;
9 }

```

(a)

```

1 for (int i = 0; i <= len - 4; i += 4) {
2   dst[i:i+3] = dst[i:i+3] + src[i:i+3];
3 }

```

(b)

Figure 5.4. (a) Example of loop found in OpenCV’s core benchmarks. (b) Version of the loop that LLVM -O3 produces.

less than 2% faster than the version that we produce with our dynamic tests.

5.6 Final Thoughts

In this chapter we discussed the evaluation of Restrictification. We used a set of open-source benchmarks taken from the LLVM Test Suite. In these benchmarks, we evaluated points such as the reach of the techniques, the effect on relevant compiler optimizations, the code size expansion, and the speedup. We saw that the Forward Region Inference did not perform well in our evaluation, whereas the static and the backward methods were able to achieve some good numbers, even though for the majority of cases our techniques could not lead to significant runtime improvements.

The later part of the chapter was dedicated to OpenCV, a large library of computer vision algorithms. In our experiments, we selected the two largest benchmark suites from the OpenCV framework of tests. We evaluated runtime improvements enabled by Restrictification, and also tried the combination of the techniques. We were able to speed up 39% of the benchmarks when combining the Purely Static and the Backward Region Inference methods, among which the best individual result was an impressive 39% of runtime reduction, which accounts to a speedup of 1.64x.

Such results lead us to the conclusion that the best approach is to use the Purely Static method in conjunction with the Backward Inference, as they cover different scenarios and thus have proven to achieve more together. At last, to better understand why some of the benchmarks experienced slowdowns, we presented an example of how poorly done manual optimizations can obstruct the compiler’s job.

Chapter 6

Conclusion

This dissertation has presented three different techniques to disambiguate pointers used as arguments of functions: the Purely Static Approach, the Forward Region Inference and the Backward Region Inference. They constitute the Restrictification suite. This disambiguation helps compilers to carry out more extensive optimizations, as we have demonstrated experimentally. One of our three techniques, the PSA, relies on the static alias analysis already available in mainstream compilers to perform pointer disambiguation. Although quite effective, alias analyses present in the LLVM compiler are still limited, which raise the need of other techniques to catch uncovered cases. To this end, we deployed two other techniques that combine static bound inference (via forward or backward analysis) with the insertion of run time checks to disambiguate pointers; hence, extending the reach of pointer disambiguation. These two methods are the Forward Region Inference and the Backward Region Inference. Also, the three techniques presented here can be used in tandem, as the application of one does not hinder the application of the other. Whenever possible to disambiguate pointers statically, the purely static approach should be used, due to its low overhead. However, the other techniques, mainly the backward approach, can be used more extensively.

To demonstrate the effects of using our approaches, we have applied Restrictification onto four benchmark suites taken from the LLVM Test Suite. In these benchmarks, we evaluated points such as the reach of the techniques, the effect on relevant compiler optimizations, the code size expansion, and the speedup. We saw that the Forward Region Inference did not perform well in our evaluation, whereas the static and the backward methods were able to achieve some good numbers, even though for the majority of cases our techniques could not lead to significant runtime improvements. Furthermore, to illustrate the advantages of our approaches over techniques such as inlining, we have applied it onto OpenCV, a widely used library for computer vision

applications. Since OpenCV is regularly built as a shared library, inlining of functions cannot be applied. In our experiments, we selected the two largest benchmark suites from the OpenCV framework. We were able to speed up 39% of the benchmarks when combining the Purely Static and the Backward Region Inference methods, among which the best individual result was an impressive 39% of runtime reduction, which accounts to a speedup of 1.64x. Such results lead us to the conclusion that the best approach is to use the Purely Static method in conjunction with the Backward Inference, as they cover different scenarios and thus have proven to achieve more together.

This work is the first to automatically insert the `restrict` keyword in pointer arguments. This is achieved by the usage of function cloning, where an optimized version of the function is created by annotating its pointer arguments with `restrict`. Our techniques then determine which function invocations can be optimized by disambiguating their real pointer arguments. Before our techniques, programmers had the burden of annotating arguments with the keyword. This task is tedious, time consuming (for legacy code to be annotated can be large), and error prone (given that a mistakenly annotated function can severely impact the program’s correctness). Of the three techniques described in this dissertation, we developed from scratch the first one, the Purely Static Method, including the extension for dynamic linkage. We are clients of the other two methods, the hybrid ones. Nonetheless, the work of deployment, integration, experiments, and empirical analysis of the three methods was done entirely by us.

6.1 Future Work

Our current implementation of the three different restrictification techniques is mature enough to handle large code bases, as we have seen in Section 5.5. Yet, much work is still to be done towards the craft of a industrial-quality pointer disambiguation framework. Below we list four possible directions in which our work can be improved:

- Handle name collisions: although unlikely, our current implementation still leaves room for this possibility. There are several ways to resolve the implicit link between the original function and the cloned function (hidden map, link time replacement, name convention, etc). We intend to explore some of these venues in the short-term.
- Test different cost models for judging when optimizations are profitable: as we have seen, restrictification sometimes leads to slower code, due to poorly tuned optimizations. Identifying and discarding unprofitable functions remains a

problem to be solved. We have been using a static profiler, à la Wu & Larus Wu and Larus [1994], to solve this problem, but our results, so far, have not been fully satisfactory.

- Combine the purely static approach of Section 4.1 with more powerful alias analyses, e.g., flow sensitive, context sensitive, inter-procedural, etc: so far, we have only used our restrictifier together with the five default analyses available in the LLVM framework. This omission is due to our inability to find implementations of more comprehensive alias analyses in LLVM that can handle very large programs.
- Further investigate why some programs experienced slowdowns: our experiments chapter (Chapter 5) showed that some programs had their runtimes increased, while our goal is naturally the opposite. Badly tuned optimizations are the major suspects from our point of view. Some of these are too aggressive and may end up slowing the program down due to a lackluster analysis of its profitability. We plan to run finer experiments to pinpoint the bad optimizations. Moreover, a future project can be on improving these optimizations.

6.2 Final Thoughts

We conclude our dissertation with a final observation on the future of techniques such as those that we have introduced. Given the progress that we have recently observed towards the automatic extraction of data parallelism from programs, be it through vectorization Karrenberg [2015], be it through SIMT parallelization Nickolls and Dally [2010], we believe that approaches such as ours, which decrease the amount of memory dependencies in programs, will have an importance that transcends classic compiler optimizations.

Bibliography

- Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley.
- Alves, P., Gruber, F., Doerfert, J., Lamprineas, A., Grosser, T., Rastello, F., and Pereira, F. M. Q. a. (2015). Runtime pointer disambiguation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 589–606, New York, NY, USA. ACM.
- Andersen, L. O. (1994). *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen.
- Andión, J. M., Arenaz, M., Bodin, F., Rodríguez, G., and Touriño, J. (2016). Locality-aware automatic parallelization for gpgpu with openhmpd directives. *International Journal of Parallel Programming*, 44(3):620–643. ISSN 1573-7640.
- Appel, A. W. and Palsberg, J. (2002). *Modern Compiler Implementation in Java*. Cambridge University Press, 2nd edition.
- Bachmann, O., Wang, P. S., and Zima, E. V. (1994). Chains of recurrences – a method to expedite the evaluation of closed-form functions. In *Proceedings of the International Symposium on Symbolic and Algebraic Computation*, ISSAC '94, pages 242–249, New York, NY, USA. ACM.
- Bertolli, C., Antao, S. F., Eichenberger, A. E., O'Brien, K., Sura, Z., Jacob, A. C., Chen, T., and Sallenave, O. (2014). Coordinating gpu threads for openmp 4.0 in llvm. In *Proceedings of the 2014 LLVM Compiler Infrastructure in HPC*, LLVM-HPC '14, pages 12–21, Piscataway, NJ, USA. IEEE Press.
- Blume, W. and Eigenmann, R. (1995). Symbolic range propagation. In *Parallel Processing Symposium, 1995. Proceedings., 9th International*, pages 357–363.

- Bodík, R., Gupta, R., and Sarkar, V. (2000). Abcd: Eliminating array bounds checks on demand. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 321--333, New York, NY, USA. ACM.
- Campos, V. H. S., Alves, P., and Pereira, F. M. Q. (2015a). Restrictifier: a tool to disambiguate pointers at function call sites. In *Congresso Brasileiro de Software - Sessão de Ferramentas*, CBSOft '15.
- Campos, V. H. S., Alves, P., and Pereira, F. M. Q. (2015b). Restritificação. In *Brazilian Symposium on Programming Languages*, SBLP '15.
- Campos, V. H. S., Rodrigues, R. E., de Assis Costa, I. R., and Pereira, F. M. Q. (2012). *Speed and Precision in Range Analysis*, pages 42--56. Springer Berlin Heidelberg, Berlin, Heidelberg.
- Chen, T., Lin, J., Hsu, W.-C., and Yew, P.-C. (2005). *An Empirical Study on the Granularity of Pointer Analysis in C Programs*, pages 157--171. Springer Berlin Heidelberg, Berlin, Heidelberg.
- 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. *ACM Transactions on Programming Languages and Systems*, 13(4):451--490. ISSN 0164-0925.
- Dagum, L. and Menon, R. (1998). Openmp: an industry standard api for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46--55. ISSN 1070-9924.
- Gallagher, D. M., Chen, W. Y., Mahlke, S. A., Gyllenhaal, J. C., and Hwu, W.-m. W. (1994). Dynamic memory disambiguation using the memory conflict buffer. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VI, pages 183--193, New York, NY, USA. ACM.
- Grosser, T., Groesllinger, A., and Lengauer, C. (2012). Polly - performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 04(22):X.
- Guo, B., Wu, Y., Wang, C., Bridges, M. J., Ottoni, G., Vachharajani, N., Chang, J., and August, D. I. (2006). *Selective Runtime Memory Disambiguation in a Dynamic Binary Translator*, pages 65--79. Springer Berlin Heidelberg, Berlin, Heidelberg.

- Hackett, B. and Aiken, A. (2006). How is aliasing used in systems software? In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '06/FSE-14, pages 69–80, New York, NY, USA. ACM.
- Hardekopf, B. and Lin, C. (2007). The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 290–299, New York, NY, USA. ACM.
- Hardekopf, B. and Lin, C. (2011). Flow-sensitive pointer analysis for millions of lines of code. In *Code Generation and Optimization (CGO), 2011 9th Annual IEEE/ACM International Symposium on*, pages 289–298.
- Hind, M. (2001). Pointer analysis: Haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '01, pages 54–61, New York, NY, USA. ACM.
- Jones, N. D. and Muchnick, S. S. (1982). A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, pages 66–74, New York, NY, USA. ACM.
- Karrenberg, R. (2015). *Whole-Function Vectorization*, pages 85–125. Springer Fachmedien Wiesbaden, Wiesbaden.
- Lattner, C. and Adve, V. (2004). LLVM: a compilation framework for lifelong program analysis transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86.
- Lesk, M. E. and Schmidt, E. (1975). Lex: A lexical analyzer generator.
- Logozzo, F. and Fähndrich, M. (2010). Pentagons: A weakly relational abstract domain for the efficient validation of array accesses. *Sci. Comput. Program.*, 75(9):796–807.
- Margiolas, C. and O'Boyle, M. F. P. (2014). Portable and transparent host-device communication optimization for gpgpu environments. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 55:55–55:65, New York, NY, USA. ACM.
- Nazaré, H., Maffra, I., Santos, W., Barbosa, L., Gonnord, L., and Quintão Pereira, F. M. (2014). Validation of memory accesses through symbolic analyses. In *Proceedings of*

- the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 791–809, New York, NY, USA. ACM.
- Nickolls, J. and Dally, W. J. (2010). The GPU computing era. *IEEE Micro*, 30:56–69.
- Nicolau, A. (1989). Run-time disambiguation: coping with statically unpredictable dependencies. *IEEE Transactions on Computers*, 38(5):663–678. ISSN 0018-9340.
- Noble, J., Vitek, J., and Potter, J. (1998). *Flexible alias protection*, pages 158–185. Springer Berlin Heidelberg, Berlin, Heidelberg.
- Pereira, F. M. Q. and Berlin, D. (2009). Wave propagation and deep propagation for pointer analysis. In *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, pages 126–135, Washington, DC, USA. IEEE Computer Society.
- Piccoli, G., Santos, H. N., Rodrigues, R. E., Pousa, C., Borin, E., and Quintão Pereira, F. M. (2014). Compiler Support for Selective Page Migration in NUMA Architectures. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 369–380, New York, NY, USA. ACM.
- Reilly, E. D. (2003). *Milestones in computer science and information technology*. Greenwood Publishing Group.
- Rodrigues, R. E., Campos, V. H. S., and Pereira, F. M. Q. (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.
- Rugina, R. and Rinard, M. C. (2005). Symbolic bounds analysis of pointers, array indices, and accessed memory regions. *ACM Trans. Program. Lang. Syst.*, 27(2):185–235. ISSN 0164-0925.
- Rus, S., Rauchwerger, L., and Hoefflinger, J. (2003). Hybrid analysis: Static & dynamic memory reference analysis. *International Journal of Parallel Programming*, 31(4):251–283. ISSN 1573-7640.
- Serebryany, K., Bruening, D., Potapenko, A., and Vyukov, D. (2012). Addresssanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 28–28, Berkeley, CA, USA. USENIX Association.

- Sperle Campos, V. H., Alves, P. R., Nazaré Santos, H., and Quintão Pereira, F. M. (2016). Restrictification of function arguments. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, pages 163–173, New York, NY, USA. ACM.
- Steensgaard, B. (1996). Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 32–41, New York, NY, USA. ACM.
- Whaley, J. and Lam, M. S. (2004). Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, pages 131–144, New York, NY, USA. ACM.
- Wu, Y. and Larus, J. R. (1994). Static branch frequency and program profile analysis. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, MICRO 27, pages 1–11, New York, NY, USA. ACM.
- Zhang, Q., Lyu, M. R., Yuan, H., and Su, Z. (2013). Fast Algorithms for Dyck-CFL-reachability with Applications to Alias Analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 435–446, New York, NY, USA. ACM.
- Zheng, X. and Rugina, R. (2008). Demand-driven Alias Analysis for C. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 197–208, New York, NY, USA. ACM.

