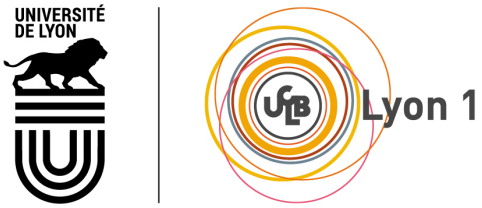


N° d'ordre NNT : 2017LYSE1167

THÈSE DE DOCTORAT DE L'UNIVERSITÉ DE LYON
opérée au sein de
l'Université Claude Bernard Lyon 1

École Doctorale ED512
InfoMaths

Spécialité de doctorat : Informatique

Soutenue publiquement le 26/09/2017, par :

Maroua MAALEJ KAMMOUN

Low-cost memory analyses for efficient compilers

Devant le jury composé de :

Xavier Urbain, Professeur des Universités, Université Claude Bernard Lyon 1	Examinateur
Sanjay Rajopadhye, Professeur, Colorado State University	Rapporteur
Corinne Ancourt, Maître de Recherches, École supérieure des mines de Paris	Rapporteure
Alain Girault, Directeur de Recherches, Inria	Examinateur
Karine Heydemann, Maître de Conférences, Université Pierre et Marie Curie	Examinatrice
Vincent Loechner, Maître de Conférences, Université de Strasbourg	Examinateur
Frédéric Vivien, Directeur de Recherches, Inria	Directeur de thèse
Laure Gonnord, Maître de Conférences, Université Claude Bernard Lyon1	Co-directrice de thèse

UNIVERSITE CLAUDE BERNARD - LYON 1

Président de l'Université	**M. le Professeur Frédéric FLEURY**
Président du Conseil Académique	M. le Professeur Hamda BEN HADID
Vice-président du Conseil d'Administration	M. le Professeur Didier REVEL
Vice-président du Conseil Formation et Vie Universitaire	M. le Professeur Philippe CHEVALIER
Vice-président de la Commission Recherche	M. Fabrice VALLÉE
Directrice Générale des Services	Mme Dominique MARCHAND

COMPOSANTES SANTE

Faculté de Médecine Lyon Est – Claude Bernard	Directeur : M. le Professeur G.RODE
Faculté de Médecine et de Maïeutique Lyon Sud – Charles Mérieux	Directeur : Mme la Professeure C. BURILLON
Faculté d'Odontologie	Directeur : M. le Professeur D. BOURGEOIS
Institut des Sciences Pharmaceutiques et Biologiques	Directeur : Mme la Professeure C. VINCIGUERRA
Institut des Sciences et Techniques de la Réadaptation	Directeur : M. X. PERROT
Département de formation et Centre de Recherche en Biologie Humaine	Directeur : Mme la Professeure A-M. SCHOTT

COMPOSANTES ET DEPARTEMENTS DE SCIENCES ET TECHNOLOGIE

Faculté des Sciences et Technologies	Directeur : M. F. DE MARCHI
Département Biologie	Directeur : M. le Professeur F. THEVENARD
Département Chimie Biochimie	Directeur : Mme C. FELIX
Département GEP	Directeur : M. Hassan HAMMOURI
Département Informatique	Directeur : M. le Professeur S. AKKOUCHE
Département Mathématiques	Directeur : M. le Professeur G. TOMANOV
Département Mécanique	Directeur : M. le Professeur H. BEN HADID
Département Physique	Directeur : M. le Professeur J-C PLENET
UFR Sciences et Techniques des Activités Physiques et Sportives	Directeur : M. Y.VANPOULLE
Observatoire des Sciences de l'Univers de Lyon	Directeur : M. B. GUIDERDONI
Polytech Lyon	Directeur : M. le Professeur E.PERRIN
Ecole Supérieure de Chimie Physique Electronique	Directeur : M. G. PIGNAULT
Institut Universitaire de Technologie de Lyon 1	Directeur : M. le Professeur C. VITON
Ecole Supérieure du Professorat et de l'Education	Directeur : M. le Professeur A. MOUGNIOTTE
Institut de Science Financière et d'Assurances	Directeur : M. N. LEBOISNE

Remerciments

Lors de la préparation et de la soutenance de ma thèse, j'ai bénéficié du soutien et de l'aide de plusieurs personnes. Par ces quelques lignes, je tiens à les remercier.

Je voudrais remercier en premier lieu Laure Gonnord qui m'a encadrée tout au long de la thèse avec le sérieux et l'enthousiasme d'une directrice, la complicité d'une amie, et l'affection et la bienveillance d'une mère. Merci pour ses relectures nombreuses du manuscrit et pour la disponibilité dont elle a fait preuve tout au long de ce travail. Je remercie aussi Frédéric Vivien qui m'a accepté dans son équipe ROMA et m'a encadré pour la fin de cette thèse. J'exprime par la même occasion ma grande gratitude à Alain Darte, qui m'a intégré dans son équipe Compsys et assuré le début de ma thèse. Sans eux, cette thèse n'aurait jamais pu avoir lieu.

Je tiens ensuite à remercier les membres du jury : merci à Xavier Urbain pour avoir accepté d'être président de mon jury de thèse. Merci aussi aux rapporteurs Sanjay Rajopadhye et Corinne Ancourt et aux examinateurs Alain Girault, Karine Heydemann et Vincent Loechner pour leurs lectures minucieuses du manuscrit et les remarques pertinentes qu'ils m'ont fait parvenir.

Je tiens aussi à remercier Fernando Magno Quintão Pereira et Sylvain Collange. J'ai eu l'opportunité d'avoir plusieurs collaborations avec Fernando et ses étudiants m'ont été d'une aide considérable vis-à-vis de l'expertise que j'ai pu acquérir dans le compilateur LLVM. J'ai beaucoup apprécié mes interactions scientifiques avec Sylvain à travers lesquelles j'ai pu me familiariser aux GPUs et découvrir de nouvelles pistes d'applications des analyses de pointeurs.

Je remercie également toutes les personnes qui ont indirectement contribué au bon déroulement de ce travail : Alexandre Isoard, mon premier et préféré collègue de bureau mais aussi l'ami qui, même après son départ, a su être là pour me motiver, me conseiller, et m'aider à surmonter des difficultés techniques ; Aurélien Cavelan et Oguz Kaya, mes collègues de bureau pour avoir partagé leur amour de travail, de persévérance et d'acharnement en fin de thèse. J'aimerais aussi exprimer ma profonde gratitude à tous les doctorants que j'ai pu croiser à un moment donné de ma thèse, pour la confiance qu'ils m'ont accordée pour les représenter au conseil du laboratoire et pour les moments de détente qu'on a pu partager ensemble (une pensée spéciale à Issam Rais pour tous les repas qu'on a partagés).

Je n'oublie évidemment pas les assistantes des équipes du LIP : Laëtitia Lecot, Evelyne Blesle, Chiraz Benamor Marie Bozo et Sylvie Boyer. Immanquablement de bonne humeur, elles ont toujours des réponses à tout !

Enfin, je remercie bien sûr ma famille :
Mes parents Ferdaous et Habib qui m'ont appris le goût de la réussite. Ils ont été toujours et depuis longtemps à mes côtés pour me soutenir, m'encourager et me combler d'amour. Mes deux sœurs, Sonia et Sirine, mes modèles de réussite et de persévérance. Mes beaux-parents Naama et Salah et mes belles-sœurs Yosra et Manel pour avoir cru en moi et m'avoir supportée.

Finalement, je remercie à mon grand amour, Adam, d'avoir supporté la distance géographique qui nous séparait tout au long de la thèse, d'avoir accepté mes humeurs et mes doutes, et d'avoir géré mes indisponibilités et mes moments de travail les plus intenses.

Résumé long

Les technologies de l'information sont devenues de plus en plus importantes dans nos vies quotidiennes, et maintenant nous vivons en grande partie ce qui a été extrapolé dans les oeuvres de littérature majeures de science fiction.

Grâce à la production de massives machines puissantes, de nouvelles tablettes et smartphones, certaines tâches sont devenues moins complexes, et nous ont permis de gagner du temps. Néanmoins, dans le même temps, l'interaction entre les différents systèmes de calcul et d'informations eux-mêmes est devenue complexe, et la manipulation de tels systèmes nécessite de plus en plus une gestion automatisée et optimisée, et donc une meilleure compréhension des problématiques liées aux ressources de calcul et de mémoire, dont dépendent leur efficacité.

L'évolution des performances des machines est allée de pair avec ce que l'on appelle "le saut performance-mémoire". L'agrandissement des capacités de stockage mémoire n'a pas été accompagnée d'une diminution substantielle de la durée d'accès aux données. En conséquence, les temps d'accès à la mémoire sont maintenant non négligeables dans le coût d'un calcul.

Un compilateur prend en entrée un programme écrit en un langage de haut niveau et génère le code bas niveau qui sera exécuté sur la machine cible. Un des moyens de rendre les programmes performants est de profiter de la rapidité des processeurs tout en diminuant le plus possible le besoin d'accéder à la mémoire. Pour cela, un compilateur peut effectuer des transformations ("optimisations") plus ou moins automatiquement, à sémantique constante.

Dans un compilateur, un des concepts cruciaux est la notion de *dépendance* entre les variables du programme : cette information sert à valider ou invalider une transformation donnée. Calculer efficacement ces dépendances reste un challenge dans les compilateurs, notamment à cause des problèmes d'*aliasing* induits par les variables pointeurs. Ces pointeurs sont en effet l'un des fondamentaux des langages de bas niveau comme le `C` car ils permettent au développeur de manipuler explicitement la mémoire. Ce sont des outils puissants et expressifs.

Démontrer que deux pointeurs `p` et `q` sont disjoints (n'aliasent pas) permet de garantir qu'une écriture dans `p` ne modifie pas la valeur pointée par `q`. Dans le cas contraire, une modification de la région pointée par `p` a un impact "silencieux" (i.e. non détectable syntaxiquement en regardant les modifications apportées à `q`) sur `q`. Par conséquence, toute optimisation utilisant le fait qu'une dépendance de données n'est pas modifiée en changeant une lecture/écriture dans le programme doit vérifier qu'il n'y a pas de dépendance "cachée" par des pointeurs.

Les optimisations de code faisant appel à une analyse d'alias sont donc nombreuses ; parmi elles on peut citer la vectorisation, l'élimination de code mort et le déplacement d'instructions dans le code. Ces optimisations sont cruciales à l'intérieur des compilateurs. Fournir aux passes d'optimisation des analyses d'alias précises et sûres (dépourvus de fausses alarmes, ce qui veut dire si l'analyse conclut que `p` et `q` sont disjoints, alors ils le sont réellement) est donc d'une extrême importance.

Au sein d'un compilateur, une analyse de pointeurs peut être réalisée statiquement (sans exécuter

le programme), dynamiquement (en exécutant une partie du code), ou de manière hybride. L'état de l'art des analyses statiques de pointeurs est pléthorique, depuis l'analyse d'Andersen [And94] jusqu'aux analyses les plus récentes telles que celle de Zhang *et al.* [Zha+14]. Les caractéristiques de ces analyses varient. Elles prennent plus ou moins en compte le flot de programme, le contexte d'appel des fonctions et les caractéristiques de structures, induisant des variations en terme de coût et de précision.

Cependant, malgré toute l'attention accordée au sujet des analyses d'alias, notre état de l'art démontre que les analyses implémentées dans les compilateurs en production ont de nombreuses faiblesses lorsqu'il s'agit de prendre en compte l'arithmétique de pointeurs. En particulier, les compilateurs actuels pêchent à désambiguïser des accès à deux cases ou deux régions d'un même tableau. Certaines analyses dites "sensibles à la structure" [PKH04; Sui+16], ou encore la *shape analysis* (capturant des structures de données complexes) fournissent une solution partielle mais encore trop coûteuse. Dans notre cas plus spécifique d'étude, l'arithmétique de pointeurs, les analyses conçues existantes [BR04; Eng+04; RR00; Wol96; Aho+06] montrent aussi leurs faiblesses. En effet, elles utilisent des intervalles numériques pour désambiguïser les pointeurs ou ont recours à des techniques de résolutions (programmes linéaires, résolution d'équation diophantiennes) que nous jugeons trop chères pour être implémentées dans des compilateurs. Les analyses que nous suggérons seront quant à elle conçues pour être à la fois sûres et efficaces. Elles utiliseront le cadre théorique de l'interprétation abstraite.

Contributions

Les contributions de cette thèse sont trois nouvelles techniques d'analyse prenant efficacement en compte les relations induites par l'analyse de pointeurs. L'objectif est de fournir au compilateur des passes d'analyse suffisamment précises pour permettre des optimisations plus agressives. Nous affirmons que la clef pour développer de telles analyses à la fois rapides et précises est la *sparsité* qui consiste à calculer (et stocker) une information invariante sur toute la durée de vie d'une variable, permettant ainsi le passage à l'échelle.

Dans une première partie, nous rappelons les notions de base d'analyse statique, et les notions importantes sur les pointeurs. Nous étudions ensuite quelques compilateurs du langage C et discutons l'état de l'art des analyses d'alias. Dans une deuxième partie, nous présentons en détail les analyses proposées et leur évaluation expérimentale. Ensuite, nous étudions l'impact de ces analyses sur les optimisations réalisées par la suite par le compilateur LLVM.

Dans la suite nous présentons l'organisation du manuscrit et donnons un bref résumé de chaque chapitre.

Part I : Context

Chapter I.1 : Background. Dans ce chapitre, nous définissons le contexte et les notions que nous utilisons au long de la thèse. Nous commençons par présenter l'interprétation abstraite qui est une méthode d'analyse de programmes basée sur l'approximation de la sémantique opérationnelle. Cette méthode permet de calculer des invariants sûrs. Nous introduisons ensuite la représentation des programmes que nous allons utiliser dans la suite, la forme *Static Single Assignment* (dans laquelle chaque variable statique n'est affectée qu'une fois), ainsi que les propriétés que nous utiliserons pour proposer des analyses statiques à faible coût. Le reste du chapitre est dévolu à la définition des concepts utiles pour réaliser des analyses de pointeurs pour les langages à la C. Nous introduisons les différentes métriques que nous utiliserons par la suite pour valider

nos analyses. Nous clôturons ce chapitre par une étude des analyses d'alias existantes dans les compilateurs de production.

Ce travail a fait l'objet d'un rapport de recherche [MG15].

Chapter I.2 : Alias Analyses : Context. Dans ce chapitre, nous nous intéressons aux analyses d'alias dans les compilateurs. Ces analyses peuvent être *statiques*, *dynamiques* ou *hybrides* : les premières sont moins coûteuses mais aussi moins précises puisqu'elles ne disposent que d'informations partielles, les secondes permettent des optimisations fines mais imposent de travailler avec une gigantesque masse d'information, enfin les analyses hybrides sont un compromis entre les deux. Le chapitre se poursuit par une classification des méthodes d'analyse statique, le choix fait dans cette thèse. Enfin, nous présentons quelques compilateurs C présents dans la littérature et commentons leurs performances en terme d'optimisation de programmes, avec un accent particulier sur le compilateur LLVM.

Chapter I.3 : State-of-the-Art of Alias Analysis Techniques. Le problème de l'analyse d'alias a suscité un grand nombre de publications dans le domaine des langages de programmation, ainsi qu'en compilation. Dans ce chapitre, nous présentons une vue d'ensemble des algorithmes que nous jugeons pertinents et les plus proches de notre axe de recherche. Les techniques que nous étudions ont été essentiellement introduites pour déterminer si deux pointeurs donnés n'*aliasent* pas, dans un but d'optimisation de code. Nous nous intéressons en particulier aux pointeurs "reliés", c'est-à dire définis à l'aide d'un même pointeur de base (via l'arithmétique de pointeur), car ils sont plus difficiles à analyser . Dans ce chapitre, nous nous intéressons aux techniques qui permettent de capturer les relations entre pointeurs qui ont un même pointeur de base, et commentons sur leur efficacité et applicabilité dans des compilateurs.

Part II : Static Alias Analyses for Pointer Arithmetic

Chapter II.1 : Symbolic Range Analysis of Pointers. Dans ce chapitre nous proposons une nouvelle analyse d'alias suffisamment expressive pour capturer les relations induites par l'analyse de pointeurs. Nous proposons de générer des invariants sous la forme d'un couple pointeur de base/ décalage, les décalages étant calculés à l'aide d'une analyse d'intervalles symboliques. Nous motivons cette analyse par un schéma de programmation typique des envois de message dans les systèmes distribués. L'analyse présentée permet de répondre de manière efficace et précise à des requêtes de la forme "est-ce que p_1 et p_2 *aliasent* ?".

L'algorithme a été implémenté à l'intérieur du compilateur LLVM (version 3.5) [LA04]. La suite du chapitre est dédiée à l'évaluation expérimentale, qui montre que l'analyse est capable de désambiguïser 1,35 fois plus de paires de pointeurs que les analyses d'alias actuellement disponibles dans LLVM. De plus, notre analyse est très rapide : nous analysons un million d'instructions assembleur en dix secondes.

Cette contribution a été publiée dans [Pai+16].

Chapter II.2 : Pointer Disambiguation via Strict Inequalities. Dans ce chapitre nous explorons une idée inspirée par le domaine abstrait des Pentagons de Logozzo et Fähndrich [LF08] qui permet de propager des informations relationnelles de type "plus petit que" entre les variables d'un programme. Nous proposons une analyse de pointeurs qui permet d'inférer de telles relations sur les variables pointeurs : si deux pointeurs `p` et `q` sont liés par une telle

relation (stricte), nous concluons qu'ils n'*aliasent* pas. Pour motiver le besoin d'une telle analyse, nous étudions une routine de tri pour laquelle les analyses basées sur les intervalles ne sont pas capables de conclure. La validation expérimentale a cette fois été réalisée dans LLVM (version 3.7). Pour certains *benchmarks* nous sommes six fois plus précis que les techniques existantes.

Cette contribution a été publiée dans [Maa+17b].

Chapter II.3 : Combining Range and Inequality Information for Pointer Disambiguation. Dans ce chapitre nous proposons une combinaison des deux techniques détaillées dans les chapitres II.1 et II.2. Nous proposons un algorithme unifié mettant en œuvre des adaptations des deux méthodes précédentes, et augmentant strictement l'expressivité de celles-ci. Cette nouvelle proposition permet en particulier de répondre plus précisément aux requêtes d'alias provenant de pointeurs non reliés (et qui représentent un large pourcentage des requêtes). Cette combinaison d'analyses, implémentée dans le compilateur LLVM (version 3.7), est capable d'analyser des programmes de grande taille comme gcc en quelques minutes. Pour certains *benchmarks*, nous sommes quatre fois plus précis que les techniques existantes.

Cette contribution a été publiée dans [Maa+17a].

Chapter II.4 : Alias Analyses for LLVM Optimizations. Dans les chapitres précédents, l'évaluation expérimentale de nos analyses a été faite en comparant le pourcentage de paires de pointeurs effectivement désambiguïsées au pourcentage obtenu par les analyses concurrentes.

Dans ce chapitre, nous étudions l'impact de l'augmentation de la précision des analyses de pointeurs sur les optimisations effectuées par le compilateur, qui est à notre avis une métrique plus significative. Nous avons choisi d'étudier en profondeur trois passes disponibles dans LLVM, pour lesquelles nous mesurons le gain induit par nos nouvelles analyses.

Contents

Abstract

This thesis was motivated by the emergence of massively parallel processing and supercomputing that tend to make computer programming extremely performing. Speedup, the power consumption, and the efficiency of both software and hardware are nowadays the main concerns of the information systems community. Handling memory in a correct and efficient way is a step toward less complex and more performing programs and architectures. This thesis falls into this context and contributes to memory analysis and compilation fields in both theoretical and experimental aspects.

Besides the deep study of the current state-of-the-art of memory analyses and their limitations, our theoretical results stand in designing new algorithms to recover part of the imprecision that published techniques still show. Among the present limitations, we focus our research on the pointer arithmetic to disambiguate pointers within the same data structure. We develop our analyses in the abstract interpretation framework. The key idea behind this choice is correctness, and scalability: two requisite criteria for analyses to be embedded to the compiler construction. The first alias analysis we design is based on the range lattice of integer variables. Given a pair of pointers defined from a common base pointer, they are disjoint if their offsets cannot have values that intersect at runtime. The second pointer analysis we develop is inspired from the Pentagon abstract domain. We conclude that two pointers do not alias whenever we are able to build a strict relation between them, valid at program points where the two variables are simultaneously alive. In a third algorithm we design, we combine both the first and second analysis, and enhance them with a coarse grained but efficient analysis to deal with non related pointers.

We implement these analyses on top of the LLVM compiler. We experiment and evaluate their performance based on two metrics: the number of disambiguated pairs of pointers compared to common analyses of the compiler, and the optimizations further enabled thanks to the extra precision they introduce.

Résumé

La rapidité, la consommation énergétique et l'efficacité des systèmes logiciels et matériels sont devenues les préoccupations majeures de la communauté informatique de nos jours. Gérer de manière correcte et efficace les problématiques mémoire est essentiel pour le développement des programmes de grande tailles sur des architectures de plus en plus complexes. Dans ce contexte, cette thèse contribue aux domaines de l'analyse mémoire et de la compilation tant sur les aspects théoriques que sur les aspects pratiques et expérimentaux. Outre l'étude approfondie de l'état de l'art des analyses mémoire et des différentes limitations qu'elles montrent, notre contribution réside dans la conception et l'évaluation de nouvelles analyses qui remédient au manque de précision des techniques publiées et implémentées. Nous nous sommes principalement attachés à améliorer l'analyse de pointeurs appartenant à une même structure de données, afin de lever une des limitations majeures des compilateurs actuels. Nous développons nos analyses dans le cadre général de l'interprétation abstraite « non dense ». Ce choix est motivé par les aspects de correction et d'efficacité : deux critères requis pour une intégration facile dans un compilateur. La première analyse que nous concevons est basée sur l'analyse d'intervalles des variables entières ; elle utilise le fait que deux pointeurs définis à l'aided'un même pointeur de base n'aliasent pas si les valeurs possibles des décalages sont disjointes. La seconde analyse que nous développons est inspirée du domaine abstrait des Pentagones ; elle génère des relations d'ordre strict entre des paires de pointeurs comparables. Enfin, nous combinons et enrichissons les deux analyses précédentes dans un cadre plus général. Ces analyses ont été implémentées dans le compilateur LLVM. Nous expérimentons et évaluons leurs performances, et les comparons aux implémentations disponibles selon deux métriques : le nombre de paires de pointeurs pour lesquelles nous inférons le non-aliasing et les optimisations rendues possibles par nos analyses.

Introduction

Computer technologies are more and more important in our lives and have brought humanity closer to realizing what we have read and seen in science fiction literature and films. By producing powerful machines, new tablets and smartphones, we are able to manage multiple problems simultaneously and control a precious resource: time. Hence, computer systems and interaction between them are becoming more complex and the need to simplify, verify, and manage well their performance is becoming more and more pressing. The performance of computers essentially depends on the speed of two different components: the CPU and the memory. Both processors and memory have significantly improved their performance. However, they did so at different rates and directions. The speed of processors has significantly increased while the capacity of memory has been improved. This created the so called *processor-memory performance gap*. The bigger the memory is, the longer the processor needs to access it. One major challenge today with computer systems is then bridging this performance gap while ensuring correctness. This is a complex role of compilers. A compiler takes a program written in a high level language as an input and generates the low level code to execute as an output in the target machine. One way to get programs performing is to profit from the speed of processors while reducing as much as possible the need to access memory. Hence, the compiler needs to transform programs such that the code produced usually computes the same result as the original one. We call these transformations "*optimizations*". Inside compilers, they can be ideally performed automatically, or with user assistance, but should usually guarantee the correctness.

This brings us to the following problem: How to detect dependencies between variables and especially pointers to optimize code efficiently and prove its correctness. Pointers are in fact one of the fundamentals and particularities of C-like languages since they allow programmers to manipulate memory. They are a powerful tool to write efficient code and perform quick computations. Analyzing the relations between pointers is called pointer or alias analysis. Given two pointers `p` and `q`, knowing that they are disjoint ensures that any write to `p` keeps `q` value unchanged. Having the fact that `p` and `q` potentially point to the same region, or lacking the information, makes `q` potentially affected by any change made on `p` and vice versa. In such a situation, optimizations should consider the dependency between the two variables and can therefore rarely be performed. Program optimizations based on alias analyses include parallelization, vectorization, dead code elimination, and code motion, etc. They are actually crucial for runtime speedup and locality enhancement. The main issue of pointer analyses is henceforth correctness. Correct alias algorithms are those without false negative answers, which means if `p` and `q` are said to be disjoint, then they really do. In the contrary, if the analysis concludes that `p` and `q` may dereference overlapping memory regions while they do not, the analysis is called *conservative*.

Inside compilers, pointer analysis can be performed at different stages. It can be *static*: at compile time, or *dynamic*: at runtime, or also *hybrid* which is a combination of the static and dynamic approaches. From Andersen's work [And94] to the more recent technique of Zhang *et al.* [Zha+14] many alias analysis techniques have been proposed. The difference in flow/context/field sensitivity lets these approaches trade precision for efficiency, efficiency for precision, or attempt

to balance between both.

However, in spite of all the attention that this topic has received, the current state-of-the-art approaches inside compilers still face challenges regarding precision and speed. In particular, pointer arithmetic, a key feature in C and C++, is far to be handled satisfactorily. Mainstream compilers still struggle to distinguish intervals within the same array. In other words, state-of-the-art pointer analyses often fail to disambiguate regions addressed from a common base pointer via different offsets. Field-sensitive pointer analyses [PKH04; Sui+16], focusing on precision at the expense of scalability, provide a partial solution for this problem. They deal in fact with `struct` data structures and can therefore distinguish different fields within a record. Shape analyses [JM82] which are mostly used for program verification can disambiguate sub-parts of data-structures such as arrays, yet their scalability remains an issue to be solved. There exist points-to analyses designed specifically to deal with pointer arithmetics [BR04; Eng+04; RR00; Wol96; Aho+06]. Nevertheless, none of them works satisfactorily for the pattern of C programs we shall deal with in this thesis. The reason for this ineffectiveness lies in the fact that these analyses use *numeric* range intervals to disambiguate pointers or resort to non efficient techniques that we believe are too expensive to be embedded in mainstream compilers. To solve relations between pointers they may use Integer Linear Programming (ILP) or the Greatest Common Divisor test to solve diophantine equations. We develop, on the contrary, alias analyses in the abstract interpretation framework, so get correctness for free.

Contributions

In this thesis, we propose some techniques to efficiently handle the pointer arithmetic problem inside compilers and therefore provide them with the information needed to perform program optimizations. We believe that the key idea behind designing efficient (cheap but precise) alias analysis algorithm is *sparsity*. A static analysis is said to be sparse if it associates information to variable and not to pairs of variables and program points, enabling it to scale to large programs. The main contributions of this thesis will be presented in its second and third parts after recalling the basic notions of static and pointer analysis, studying C compilers, and discussing the state-of-the-art of alias analysis. In the following, we give the organization of this manuscript and present a brief summary of each chapter.

Part I: Context

Chapter I.1: Background. In this chapter, we define the context and notions that we shall use in this thesis. We start by presenting the abstract interpretation framework which is a method to analyze programs based on approximations. In this framework computations are made simpler and cheaper due to over-approximations. Next, we introduce the *Static Single Assignment* (SSA) forms and properties since we shall analyze programs under this representation to design static, cheap but precise, analyses on a sparse fashion. A program is said to be in SSA form if each static variable is assigned only once. The rest of this chapter will be dedicated to the definition of pointer notions and possible relations between pointers in the C language. We shall define in deep detail the pointer analysis and discuss the different metrics used to evaluate such analyses. We close the chapter by motivating the need for alias analyses for program verification and optimization inside compilers. This work figures also in the research report [MG15].

Chapter I.2: Alias Analyses: Context. In this chapter, we shall be interested in alias analyses inside compilers. We start by discussing when a pointer analysis could be executed

regarding the runtime. The three different types: *static*, *dynamic*, and *hybrid* compete in precision and scalability. Static analyses are known to be cheaper but less precise than dynamic ones while hybrid approaches try to stand between them. This thesis falls into the context of static analysis, so we shall develop it more than the two other approaches. The static alias analysis problem is then presented under different classes and approaches. Since our goal is to design static analyses for compilers, we will present in this chapter some of the state-of-the-art C compilers and comment on their performance to optimize programs with a focus on the LLVM compiler.

Chapter I.3: State-of-the-Art Alias Analysis Techniques. The alias analysis problem has spurred a long string of publications within the programming language literature. In this chapter, we give an overview of algorithms we found the most relevant and the most related to our line of research. The techniques we study were mainly introduced to disambiguate pointers for program performance purposes. To that end, we adopt a classification for the pointer analysis problem. We distinguish between two types of pointers: related and non-related. We call related pointers those defined from the same base pointers. Using pointer arithmetic, one pointer is defined based on the other. Pointers which are not related are defined from different base pointers and are easier to disambiguate. In this chapter, we focus on the techniques that deal with regions addressed from a common base pointer, and comment on their efficiency and applicability inside compilers.

Part II: Static Alias Analyses for Pointer Arithmetic

Chapter II.1: Symbolic Range Analysis of Pointers. In this chapter, we design a new alias analysis to solve pointer arithmetic. The key insight of our approach is the combination of alias analysis with symbolic range analysis. To motivate the new analysis, we start by showing a pattern typically found in distributed systems where the goal is to disambiguate pointers sharing a base pointer with *symbolic* offsets. The analysis we present has two sub-parts to answer alias queries. To evaluate our algorithm, we have implemented it on top of the LLVM compiler (version 3.5) [LA04]. In this chapter, we discuss its precision compared to the LLVM alias analyses and the current state-of-the-art techniques. Actually, for the benchmarks we analyzed, we can disambiguate 1.35x more queries than the alias analyses currently available in LLVM. Furthermore, our analysis is very fast: we can go over one million bytecode instructions in 10 seconds. This contribution was published in [Pai+16].

Chapter II.2: Pointer Disambiguation via Strict Inequalities. In this chapter, we start from an obvious, yet unexplored, observation to design a new static alias analysis. Given two pointers `p` and `q`, we conclude that they do not alias if our analysis discovers a "strict less than" relation between the two variables. This idea of comparing program variables to collect information about them was inspired by Logozzo and Fähndrich [LF08] Pentagons abstract domain. To motivate the need for an analysis based on strict inequalities, we recall in the beginning of this chapter some sort routines in which interval-based analyses are not able to disambiguate array access pointers. We then explain the algorithm we design to analyze and disambiguate pointers. To validate our ideas, we have implemented this technique in the LLVM compiler (version 3.7). It runs in time linear on the number of program variables and, depending on the benchmark, it can be as much as 6 times more precise than the pointer disambiguation techniques already in place in that compiler. This contribution was published in [Maa+17b].

Chapter II.3: Combining Range and Inequality Information for Pointer Disambiguation. This chapter is dedicated to the combination of the two pointer analyses we introduce in Chapter II.1 and Chapter II.2. To join both these research directions into a single path, we propose adaptations to both methods and also design an analysis able to disambiguate pointers defined from different base pointers (non-related pointers). This way, added to pointer arithmetic, we are able to precisely answer more queries issued from non-related pointers (which are actually quite numerous). Furthermore, we reduce the number of queries, analyzed but not handled, by the two techniques. This combination implemented in the LLVM compiler (version 3.7) is able to handle programs as large as SPEC's gcc in a few minutes. Furthermore, we have been able to improve the percentage of pairs of pointers disambiguated, when compared to LLVM's built-in analyses, by a four-fold factor in some benchmarks. This contribution was published in [Maa+17a].

Chapter II.4: Alias Analyses for LLVM Optimizations. In the previous chapters, we evaluated out novel alias analysis techniques by comparing their capabilities in disambiguating pairs of pointers against those of analyses implemented in same compiler. In this chapter, we adopt another technique of evaluating alias analyses which is less straightforward to handle however much more signi cant. This technique consists in evaluating the optimizations performed by the LLVM optimizer when our alias analyses are run. In this work, we chose to test our analyses on three (among few others) of the LLVM optimizations that carry out optimizations based on the alias analysis information. In each section of this chapter we present one optimization, comment on the collected results and in some cases investigate the optimization programs to better understand these results.

Part I

Context

Chapter I.1

Background

Contents

I.1.1 Abstract Interpretation Framework

Abstract interpretation [CC77] is a method to analyze the semantic of C programs based on approximations. It aims to gather *statically* information about runtime variable states. Given a property to prove on a program, the idea is to abstract program variables, instructions, calculus, and reasoning, and to prove this property in the abstract domain. The goal is to reduce the analysis complexity in the concrete world. The correctness of the abstract interpretation depends on the correctness of translating the concrete semantic to the abstract one.

For termination reasons, the abstract interpretation framework performs an *over* approximation of the concrete semantic, which introduces a lack of precision. The main trade-off is indeed soundness/precision in order to avoid false alarms while ensuring correctness.

According to [CC77], most program analysis techniques can be considered as abstract interpretation applications. Recently, this framework has been put onto work to develop production tools as Astrée [Bla+03], a real-time embedded software static analyzer, and PolyspaceCodeProver[1] to prove the absence of runtime errors in C and C++ like overflow, and out-of-bounds array access.

In this thesis, we shall use the abstract interpretation framework, with adaptations, to design static analyses of pointers. Our goal is to make these analyses scale well when embedded inside compilers, without sacrificing much of precision.

Definitions, example
In the abstract interpretation framework, to make computation simpler, we abstract program variables and then perform computations based on over-approximation in the abstract domain. In Definition 2 we give the relation between the concrete domain $\mathscr{C}$ (possible values for the analyzed variables) represented by a complete lattice $(\mathscr{C}, \sqsubseteq, \sqcup, \sqcap, \top, \bot)$ (cf. Definition 1) and the abstract domain denoted by $(\mathscr{C}^\sharp, \sqsubseteq^\sharp, \sqcup^\sharp, \sqcap^\sharp, \top^\sharp, \bot^\sharp)$. Theorem 1 shows how to over-approximate concrete operations.

Definition 1 *Lattice*

A lattice *is a partially ordered set, such that each pair of elements has an upper and a lower bound. A lattice is said "complete" if each subset (finite or infinite) of elements has a lower and an upper bound.*

Definition 2 *Galois Connection*

A Galois connection [CC92] between $\mathscr{C}$ and $\mathscr{C}^\sharp$ is a couple of functions (α, γ):

$$\begin{cases} \alpha : \mathscr{C} \to \mathscr{C}^\sharp : \textit{ abstraction} \\ \gamma : \mathscr{C}^\sharp \to \mathscr{C} : \textit{ concretization} \end{cases},$$

such that

$$\forall x \in \mathscr{C}, \forall \in \mathscr{C}^\sharp, \alpha(x) \sqsubseteq^\sharp y \Leftrightarrow x \sqsubseteq \gamma(y)$$

A scheme for Galois connection is given by Figure I.1.1.

In our setting, Galois connections are used to *abstract* a (possibly infinite) set of values of program variables (numerical variables, pointer addresses, ...):

[1] http://www.mathworks.com/products/polyspace-code-prover/index.html?s_tid=gn_loc_drop.

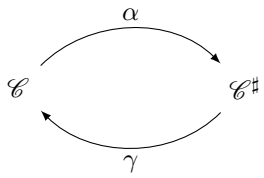


Figure I.1.1 – Galois connection.

- A set of concrete values of program variables $C \in \mathscr{C}$ is abstracted as an abstract value $\alpha(C) \in \mathscr{C}^\sharp$.
- The concrete operations of the program (tests, computations...) are "simulated" on abstract values.
- These abstract operations are used to compute incrementally (a superset of) the (abstract) reachable values for all executions of the program.
- The concretization function is used to get back in the concrete world.

The correctness of the technique is ensured by Theorem 1.

Theorem 1 *If $\mathscr{C}$ and $\mathscr{C}^\sharp$ are two domains related by the Galois connection (α, γ), if $\mathtt{F}$ is a continuous function from $\mathscr{C}$ to itself, and $\mathtt{G}$ is a continuous function from $\mathscr{C}^\sharp$ to itself, then:*

$$\textit{If } (\forall x \in \mathscr{C}, \alpha(\mathtt{F}(x)) \sqsubseteq^\sharp \mathtt{G}(\alpha(x))) \textit{ then } \textit{lfp}(\mathtt{F}) \sqsubseteq \gamma(\textit{lfp}(\mathtt{G}))$$

where lfp denotes the least fixpoint.

In our setting, $\mathtt{F}$ denotes the effect of transitions of our program to analyze (it computes a "next" state of values from a given set of values), and $\mathtt{G}$ denotes the same "in the abstract world". If $\mathtt{G}$ is correctly designed, i.e. it satisfies the property $\forall x \in \mathscr{C}, \alpha(\mathtt{F}(x)) \sqsubseteq^\sharp \mathtt{G}(\alpha(x))$, then the least fixpoint of the program in the abstract world (i.e. the least set of values that can be reached during the execution of the abstract transition function $\mathtt{G}$) can be used (after concretization) to overapproximate the least fixpoint of the real initial program (i.e. the actual set of values).

An example of such Galois connection is the interval abstract domain. The set of values of a given numerical variable $\mathtt{x}$ is abstracted by an interval $[l, u]$ that contains it, u and l being elements of $\mathbb{Z} \cup \{+\infty, -\infty\}$. The concretization of a given interval $[l, u]$ is the set of all values $\mathtt{y}$ that satisfy $l \leq \mathtt{y} \leq u$.

A way to compute the least fixpoint in the abstract world consists in beginning from the initial control point of the program with an abstract value expressing "any value", and applying the "abstraction transition function" on it, following the natural order of the control flow graph of the program. The computation is stopped when the growing abstract sets remain stable. This is illustrated in Example 1. This least fixpoint is however not always computable (it is the result of a possibly infinite computation). To make the computation finish, we make use of a widening operator that has the property to make the computation terminate in a finite number of steps. We introduce this operator in Definition 3.

Definition 3 *Widening operator*

A widening operator [CC92] is a function $\nabla : \mathscr{C}^\sharp \times \mathscr{C}^\sharp \mapsto \mathscr{C}^\sharp$ that satisfies the following conditions:

1. $\forall y_1, y_2 \in \mathscr{C}^\sharp, (y_1 \sqcap^\sharp y_2) \sqsubseteq^\sharp (y_1 \nabla y_2)$,

2. *given* $(x_n)_{n\in\mathbb{N}}$ *a growing arithmetic progression defined on* $\sqsubseteq^\sharp$, *the progression defined by* $y_0 = x_0$ *and* $y_{n+1} = y_n \nabla x_{n+1}$ *converges within a finite number of iterations.*

The widening operator ensures that the computation of the sequence X_0, $X_1 = X_0 \nabla (X_0 \sqcup G(X_0))$, $X_0 = X_1 \nabla (X_1 \sqcup G(X_1)) \ldots (\sqcup$ is the abstract union) terminates and stabilizes with X_k being an overapproximation of the least fixpoint of G. We can choose to apply this operator only on control points that are a cut set of the graph.

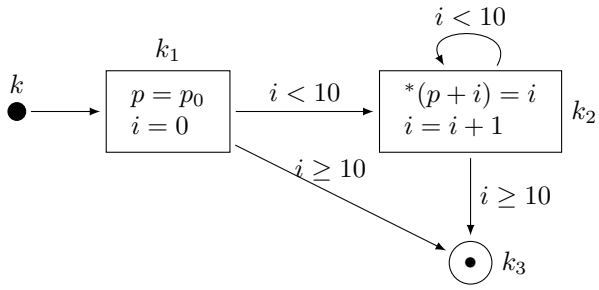
For intervals, the widening operator is defined in Definition 4.

Definition 4 *Widening for intervals [CC77]*

Given two intervals $[l_1, u_1]$ *and* $[l_2, u_2]$, *we define the widening operator:*

$$[l_1, u_1] \nabla [l_2, u_2] = \begin{cases} [l_1, u_1] \text{ if } l_1 = l_2 \text{ and } u_1 = u_2 \\ [l_1, +\infty] \text{ if } l_1 = l_2 \text{ and } u_2 > u_1 \\ [-\infty, u_1] \text{ if } l_2 < l_1 \text{ and } u_1 = u_2 \\ [-\infty, +\infty] \text{ if } l_2 < l_1 \text{ and } u_2 > u_1 \end{cases}$$

```
1 void abstraction(int N, int* p){
2   for(int i = 0; i < N; i++)
3       *(p+i) = i;
4 }
```



(a) Program code (b) CFG of program in Figure I.1.2a.

Figure I.1.2 – Simple example for abstract interpretation.

Example 1 *Consider the program in Figure I.1.2a and its control flow graph (cfg) given in Figure I.1.2b. We want to find a superset of possible values of i at each control point of the program (let us assume that* $N = 10$*). For that purpose, we compute abstract values from the entry point of the program, apply the abstract operations on intervals, apply union at merge nodes and widen at node* k_2.

- *Initialization: For all control points k different from the beginning,* $X_k = \emptyset$, *and in the initial control point i can be of any value* ($X_k = [-\infty, +\infty]$).
- *First step: the effect of the instruction* `i = 0` *on* $[-\infty, +\infty]$ *gives* $X_{k_1} = [0, 0]$. *From this point we obtain* $X_{k_2} = [0, 0] +^\sharp 1 = [1, 1]$ *at the end of block 2. From these two values we compute* X_{k_3} *as the union of* $[0, 0]$ *and* $\emptyset$, *since* $1 \geq 10$ *is false.*
- *Second step:* $X_{k_1} = [0, 0]$ *remains stable,* X_{k_2} *is temporarily updated as the union of* $[1, 1]$ *(its previous value) and the new value* $[2, 2]$, *thus* $X_{k_2} = [1, 2]$. *We apply the widening operator to obtain the value of* X_{k_2} *after step 2:* $[1, 1] \nabla [1, 2] = [1, +\infty]$. *Then we update* X_{k_3} *which is now* $[0, +\infty]$.
- *Third step: one more iteration gives the same results.*

We have reached a fixpoint, and the result of the analysis is one abstract (interval) value at each control point. Among these values, at control point k_2 we obtain $[1, +\infty]$ as the abstract value, which after concretization gives that $i \geq 1$ at this control point.

In Example 1, after applying the widening at block k_2, possible values of variable i are $[1, +\infty]$. This is a very imprecise result since in the concrete domain, we have at most $i = 10$. To recover part of this imprecision, we may use a descending sequence of finite size after convergence. We explain this technique and give examples in Section II.1.4.2.

In the next section, we introduce the *Static Single Assignment* (SSA) form, which is an *Intermediate Representation* for static analyzed programs. We will show in Part II of this thesis, how using SSA form in the abstract interpretation framework allows us to build cheap static pointer analyses for compilers.

I.1.2 Static Single Assignment (SSA)

The static single assignment form [AWZ88; Cyt+91] is an intermediate representation of program code. A program is said to be in SSA form if each static variable is defined only once. Its value is then independent of its position in the program. Variables with multiple definition sites should be renamed and new names are propagated to read statements. Since each variable in SSA form is defined once and never changed during its live range, the conversion to SSA form is not straightforward for *address-taken variables*. This kind of variables presents indirect definitions and uses through pointers. Before detailing this complication, let us start by introducing the SSA form for *top-level variables*, whose addresses are not shared with other variables and therefore no indirect access is possible. We use the code fragment in Figure I.1.3a to show how the SSA form is built:

```
1 y = ...;
2 x = y + 1;
3 y = 2;
4 x = x + 3;
5 z = x + y;
```

(a)

```
1 y0 = ...;
2 x0 = y0 + 1;
3 y1 = 2;
4 x1 = x0 + 3;
5 z0 = x1 + y1;
```

(b)

Figure I.1.3 – Example of variable renaming.

Variables `x` and `y` are defined, updated, and read, a couple of times which gives after renaming the code in Figure I.1.3b.

Note that in the transformed program, the first definitions of `x` and `y` and their updates ($\texttt{x} = \texttt{x} + 3$ and $\texttt{y} = 2$ respectively) have different names. Given the statement, $\texttt{z}_0 = \texttt{x}_1 + \texttt{y}_1$, the last defined version of `y`, namely $\texttt{y}_1$ is used. We now suppose that the first update of `y` ($\texttt{y} = 2$) is under a given condition $\texttt{Cond} = (\texttt{x} < 4)$ (If (`Cond`) then $\texttt{y} = 2$ else $\texttt{y} = 1$). A simple renaming of variables is no more efficient since the executed path is unknown before compile time or runtime. Then, the value taken by `y` cannot be determined after the merge as depicted in Figure I.1.4a (block D). We let block B be the true branch of condition `Cond` and C the false one. To merge values after different branches, SSA defines a special statement called ϕ. The ϕ function is inserted at the beginning of each merge block to rename redefinitions in conditional branches. Figure I.1.4b shows the new control flow in full SSA form using ϕ function.

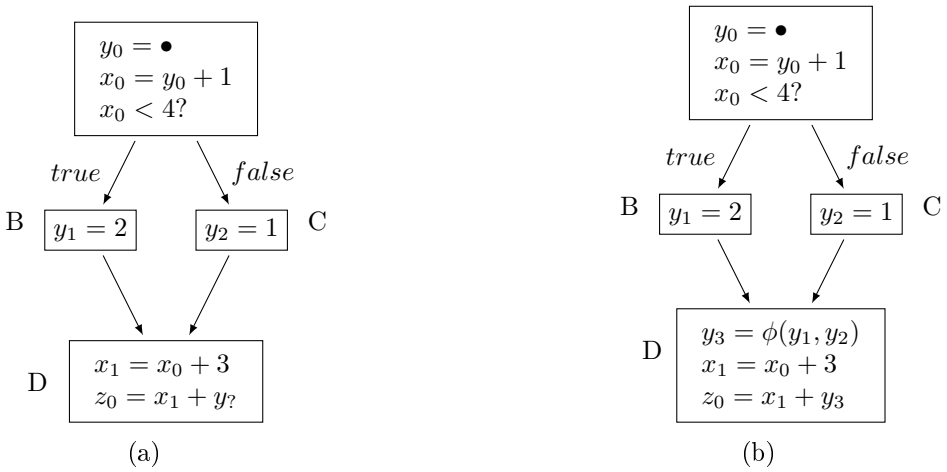


Figure I.1.4 – Transforming a code into SSA form. On the left `y` variable is not renamed after split. On the right we introduce ϕ to join new names of `y`.

We now go back to address-taken variables. We consider example in Figure I.1.5 to show the difference compared to top-level variables and see why this variable class should be considered separately. Note that in Figure I.1.5a `p` takes the address of variable `a`. Using the simple renaming of variables as we saw for top-level variables gives the code in Figure I.1.5b. The new name `a0` of `a` is propagated from the definition site at line 2 to use sites at lines 4 and 6. At line 5, we store the value 3 in pointer `p0`. Note that since `p0` points to `a` (`p0` = &`a`), this assignment modifies `a`'s value. The new name `a0` now contains 3 at line 4 and no more 2. This is contradictory with the definition of SSA form which should ensure the same value of each defined variable at all its use sites. To fix this incoherence, one should discover the indirect definitions and uses to handle them, which can only be done using a points-to information. Since in this thesis, we are mainly interested in analyzing programs on the top of compilers, we shall explain in deep detail in Section I.2.3.4 how the LLVM compiler handles address-taken variables to perform SSA transformation. We yet refer the reader to [HL11; YSX14] [Ras16, §25.2] for further reading on transforming address-taken variable into SSA form.

```
1 int* addressTaken(int* q){
2   int a = 2;
3   int c = *q + 4;
4   int* p = &a;
5   *p = 3;
6   int b = a + c;
7 return p;
8 }
```

(a) Program code.

```
1 int* addressTakenSsa(int* q){
2   int a0 = 2;
3   int c0 = *q0 + 4;
4   int* p0 = &a0;
5   *p0 = 3;
6   int b0 = a0 + c0;
7 return p;
8 }
```

(b) Wrong SSA form for program in Figure I.1.5a

Figure I.1.5 – Example to illustrate the complication of transforming address-taken variables into SSA form.

As we shall discuss in Paragraph I.2.2.3.2, SSA form is useful when analyzing and optimizing programs. However, ϕ functions are program representations which are not present in compiler instruction sets. Therefore, compilers should eliminate ϕ functions and related variables before code generation. This is what we call *SSA elimination* or *SSA destruction*. Most compilers remove

SSA form before register allocation (mapping program to a bounded number of registers) by inserting variable copies instead of ϕ functions. Further reading on SSA form destruction before and after register allocation may be found in [Ras16, §3.2] and [PP09].

SSA renaming *splits* the *live-ranges* of variables and therefore creates a direct link between the *unique* variable definition and its different uses known as *"variable def-use chain"*. The def-use chain associates an *information* directly to a variable `v` and ensures it holds whenever `v` is alive. A program representation that guarantees this relationship between variables and information satisfies the *Static Single Information* (SSI) property. In Definition 5 we quote Tavares *et al.* [Tav+14] definition of SSI property.

Definition 5 (Static Single Information Property) *A dataflow analysis bears the static single information property if it always associates a variable with the same abstract state at every program point where that variable is alive.*

Extended Static Single Assignment form (e-SSA) and *Static Single Information* form are two classic extensions of the SSA form that define additional live-ranges splitting of variables. Compared to the standard SSA form, these extensions, presented in Section I.1.2.1 and Section I.1.2.2, are more accurate to provide the Static Single Information property. The reader may refer to [Ras16]{§14} for further details about the SSI property (examples and implementation).

I.1.2.1 Extended Static Single Assignment form

The *extended* SSA [BGS00] (e-SSA) is a flavor of SSA form which can be computed cheaply from it. This extension ensures the SSI property for program analyses that extract information at *definition sites* and also at *conditionals*. Besides the SSA renaming, it introduces σ-functions to redefine program variables at split points such as branches and switches. These special instructions, the σ-functions, rename variables at the out-edges of conditional branches. They ensure that each outcome of a conditional is associated with a distinct name. Figure I.1.6 shows the control flow graph of program Figure I.1.4b into e-SSA form. Sigma nodes are added in block B and C to ensure the life range splitting after and therefore a new information related to variable $\mathtt{x_0}$ after the conditional test: $\mathtt{x_0} < 4$. In the *true* branch, $\mathtt{x_{0t}}$ is defined to denote possible values of $\mathtt{x_0}$ that should be strictly less than 4. In the *false* branch, possible values of $\mathtt{x_0}$ are those greater than 4 denoted by $\mathtt{x_{0f}}$. Note that a new ϕ function is defined in block D to merge $\mathtt{x_{0t}}$ and $\mathtt{x_{0f}}$. Both ϕ functions in the beginning of block D are executed simultaneously.

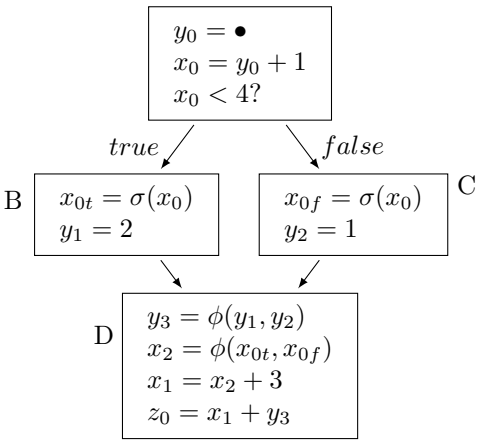


Figure I.1.6 – Transforming code in Figure I.1.4b into e-SSA form.

I.1.2.2 Static Single Information form

The Static Single Information form (SSI form) was coined by Ananian and described in his Master's thesis [Ana97]. It ensures the SSI property for program analyses that extract information at *definition sites* and some *use sites*. To propagate information from use sites, this form may not alter program instructions like e-SSA form does, but consider the renaming only during analysis. However, some data-flow analyses require the insertion of σ-functions to rename used variables and ϕ-functions to join new names after the split. In Part II, we shall give examples of SSI form and show how it can be used to ensure SSI property for a data-flow analysis.

I.1.3 Pointers In C

Pointers are one of the fundamentals and particularities of C language since they allow programmers to manipulate memory. They are indeed a powerful tool to write efficient code and perform quick computations. A pointer variable `x` is used to store the address of another variable `y`. We say that *x points to y* and write `x` = &`y`. Like any variable in C, a pointer has a type `type_t` and should be declared before use: `type_t` ∗ `p`. However, a declared pointer does not point to anywhere. To access the value pointed by `x`, we use the asterisk ∗ and we say that ∗`x` *dereferences* `y`. Two types of pointer declarations exist in C: *Stack* and *Heap* pointers. Stack pointers are statically allocated while heap pointers are dynamically allocated. When running a program, the difference is invisible. However, from a programming point of view, routines like `malloc` and `alloc` are used to define a pointer on the heap and the memory should be freed using `free` if the pointer and its aliases are no longer used in the program. Defined pointers in the stack or in the heap may point to a bench of memory slots, this is the case or array definition. Both arrays have contiguous elements in the memory. Unlike static arrays (defined on stack), a dynamic array may have a non constant length. Its name `p` is a pointer to `p`[0] that can be changed using *pointer arithmetic* to point to another memory location. In this thesis, we introduce techniques to analyse memory accesses through pointers as they are performed in the low level representation. However, for the sake of simplicity, we shall represent pointer accesses as array accesses in most of the examples we study.

I.1.3.1 Pointer arithmetic

An arithmetic operation on a pointer deals with its value which is an address. Hence, a pointer value can only be incremented or decremented. We let `p` be a pointer of type `type_t`: `type_t` ∗ `p` and consider Figure I.1.7.

- Increment: `p` ++ ≃ `p` = `p` + `1`. `p` + `1` points to the element of memory that follows the one pointed by `p`.
- Decrement: `p` - - ≃ `p` = `p` − `1`. `p` − `1` points to the element of memory that preceeds the one pointed by `p`.

To better understand pointer arithmetic, we perform these operations on the address scale. We let `type_t` be the short integer type `int` (4 bytes) and suppose that `p` points to address `0x0000`. Each time `p` is incremented, it will point to the next element which is an integer. `p` is hence incremented by 4 bytes and its address become `0x0000` + `4` = `0x0004`.
In general, we can write `p` = `p` + `x` with `x` a positive or negative integer. We can also define a new pointer p_1 as an *offset* `x` from the *base pointer* `p` such that p_1 = `p` + `x`. However, to be correct, after such operation, p_1 (or incremented `p`) should be pointing to an allocated memory. In other words, if `type_t` ∗ `p` = `malloc`(`N` ∗ `sizeOf`(`type_t`)), then `x` should be strictly smaller than `N`

($x < N$) and the maximum address that can be taken is $addr_p + (N - 1) * sizeOf(type\_t)$) when $addr_p$ refers to the original address of p.
Given a pair of pointers $p_1 = p + x$ and $p_2 = p + y$ defined from the same base pointer p, we can also subtract one from the other. The result we get is an integer value. For instance, $p_1 - p_2 = x - y$. However, the C standard does not allow pointer addition.

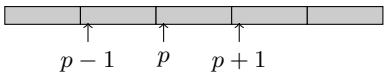


Figure I.1.7 – One step increment and decrement for pointer p.

I.1.3.2 Pointer comparison

The ISO C Standard prohibits comparisons between two references to separately allocated objects [ISO11, S6.5.8, p.5], even though they are used in practice [Mem+16, p.4]. Hence, two pointers p_1 and p_2 can be legally compared if they are related to each other. In other words, one is the base pointer of the other or they have been potentially transitively defined from the same base pointer p. We compare p_1 and p_2 using relational operators: $<$, $\leq$, $>$, $\geq$, $==$ and get an integer result for True or False. Thanks to pointer subtraction, we can evaluate pointer comparison as an integer comparison. Example 2 illustrates this statement.

Example 2 *Consider the program snippet below.* p_1 *and* p_2 *are both defined from* p. *Their comparison is then legal.* $p2 - p1 > 0$ *and the while loop condition is* true.

```
1 int* p = malloc(10*sizeof(int));
2 int* p1 = p + 4;
3 int* p2 = p + 9;
4 while (p1 < p2){
5   p1 ++;
6 }
```

I.1.3.3 Pointer and Alias analysis

Pointer analyses or points-to analyses are program analyses that aim to discover all the memory locations a pointer expression can refer to. It builds the set of variables a given pointer can reference. Alias analyses is being interested in relationship between pointers and compute pairs of expressions to discover whether the two expressions might refer to the same location. Although the difference between both concepts in the literature is quite unclear, we believe that alias analysis is a near-synonym and goes only one step beyond pointer analysis. Provided the information on memory locations pointed by two pointer variables, an alias analysis can be easily derived by comparing those locations [Ema93; Sui+16]. In this thesis, we shall use both terms indifferently while focusing on the alias analysis concept.

Alias analysis techniques are actually widely used to perform, on languages with pointers, verification, and optimizations inside compilers. Knowing that two pointers p and q do not refer to overlapping memory regions, ensures that any write to p keeps unchanged q's value. Having the fact that p and q potentially point to the same region, or ignoring the information, makes q potentially affected by any change made on p and vice versa. In compiler documentation, we can distinguish exactly *four* behaviors for p and q (Figure I.1.8).

- No Alias: `p` and `q` are disjoint. They do not dereference overlapping memory regions.
- Partial Alias: `p` and `q` overlap in some way but do not start at the same address.
- Must Alias: `p` and `q` alias and start at the same address.
- May Alias: `p` and `q` might reference regions that overlap.

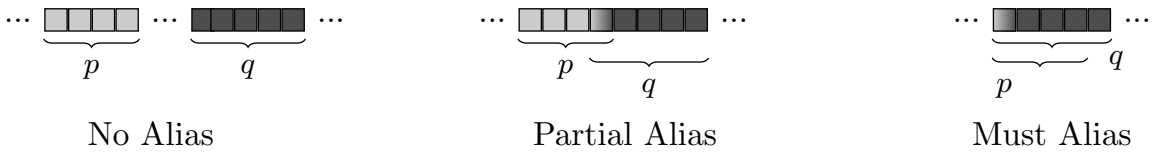


Figure I.1.8 – Concrete representation of memory regions in case of No, Partial and Must aliasing. Each □ represents a memory slot.

Since most alias analysis techniques use abstract domains to represent pointers and track their behaviors, and also use widening and semi-lattice join operations, the slight differences between Partial, Must, and May Alias is generally lost and a non No Alias information is assimilated to a May one.

The main issue of pointer analyses inside compilers is *correctness*. Correct alias algorithms are those without false negative answers, which means two pointers are said "no alias" when they do. As demonstrated by [Wu+13], this condition is often compromised but not usually detected. Therefore, correct analyses like those we shall introduce in Part II tend to be conservative: a positive answer "no, they do not alias" will guarantee that the two pointers never overlap during execution. On the contrary, these analyses return a "may alias" answer when they fail to disambiguate some pairs of pointers, even if there is no execution where the two pointers actually access the same block.

I.1.3.3.1 Pointer analyses evaluation

To evaluate a pointer analysis algorithm, one shall answer three questions dealing with the precision, the scalability and the applicability of the approach:

- Precision: how effective is the analysis to disambiguate pairs of pointers?
  The precision of an analysis is measured by its capacity to disambiguate pairs of pointers compared to other analyses. The more NoAlias answers, the more precise the analysis.
- Scalability: can the analysis scale up to handle very large programs?
  The scalability of an alias analysis algorithm refers to its ability to analyze big programs in a reasonable time. The complexity of the analysis as a function of the size of target program reveals its scalability.
- Applicability: is the analysis able to increase the effectiveness of existing program analyses?
  To measure the applicability of an alias analysis, one can measure how far it improves existent analysis clients (optimizations) or its designed client, or the quality of implemented alias analyses. Even though measuring the improvement of the efficiency of clients seems to be the most relevant metric, we assume that its implementation is not straightforward. As we will show in Section I.2.4, in the LLVM compiler many analyses and transform passes interfere and are managed by the *pass manager*. Hence, evaluating the effect of a given analysis on its own is quite hopeless and rarely predictable. One way to avoid such group results is to design an optimization pass for each analysis approach and measure its effectiveness. Nevertheless, the applicability of the analysis should be very limited in production compilers. The second approach for evaluating the applicability of a new analysis is to measure the extra precision resulting in combining it with already implemented alias

analyses.

I.1.3.3.2 Alias analysis trade-off

Pointer analysis has been shown to be undecidable [Ram94; Lan92]. Therefore, many approaches that trade precision for efficiency [Oh+14], efficiency for precision [NG15; Ste96], or even attempt to balance between both [YSX14; HL11; Oh+12; Pai+16], have been proposed and improved in time and space [PB09]. When a pointer `p` is within an alias set of another pointer `q` (i.e. `q` is supposed to may alias `p`) but pointers are disjoint and do not dereference overlapping memory regions, the algorithm is said to be imprecise. Actually, these kinds of analyses are cheaper than precise ones that have been shown to be NP-hard [LR91; Hor97].

I.1.4 The Need For Alias Analyses

Alias analyses are proposed and used for verification and optimization. Inside compilers, they are mostly used to perform transformations on source code, leading to many optimizations. In the sequel, we make an assessment of some optimizations that are *clients* of alias analyses. We provide examples in which we illustrate the need for alias analyses to enable more efficient program optimizations.

I.1.4.1 Program verification: array out-of-bounds check

A violation of memory safety in C leads to undefined behavior that may lead to incorrect program or even a non-termination. One form of memory violation is array out-of-bounds access. To avoid this bug, many alias analyses [Naz+14; Str+14; VB04; RR00; BGS00] have been proposed. Some of these approaches shall be described in Chapter I.3. In Example 3, we use the analysis introduced by Nazare *et al.* [Naz+14] to illustrate how pointer analysis may avoid out-of-bound access checks. The main idea of this technique is to use symbolic range analysis to track valid references from array base pointers. Gards for bound checks may be eliminated if the memory access has been validated.

Example 3 *Consider the program shown in Figure I.1.9. After renaming variables (cf. I.1.2), we let* $\mathtt{p_i} = \mathtt{p} + \mathtt{i}$ *at line 6 and* $\mathtt{p_j} = \mathtt{p} + \mathtt{j}$ *at line 7. W denotes the function that maps a pointer to its addressable offsets. The static analysis detailed in [Naz+14], proves that* $\mathtt{W(p)} = [0, \mathtt{N} - 1]$, $\mathtt{W(p_i)} = [0, 1]$ *and* $\mathtt{W(p_j)} = [-1, 0]$. *In other words, this analysis tells us that* $\mathtt{p_i}$ *for instance can* ***safely*** *dereference addresses from 0 to 1 starting from* $\mathtt{p_i}$*, and that the largest addressable offset from p is* $\mathtt{N} - 1$. *Therefore, in this case we no longer need to add runtime checks to guarantee the safety of memory accesses.*

I.1.4.2 Alias analysis for loop code motion

Code motion consists in interchanging some statements or moving others. This optimization is beneficial especially in loops where *loop invariant code motion* (LICM) removes unchanged statements from the loop by hoisting or sinking them and therefore avoiding useless executions. Code motion is also used to improve memory and cache accesses.

The program snippet in Figure I.1.10a shows a loop with a read from `p` and a write in $\mathtt{p_2}$. Clearly $^*\mathtt{p}$ is never changed inside the loop since we don't write in `p`, and `p` and $\mathtt{p_2}$ access disjoint pieces

```
1  unsigned N;
2  scanf("%d", &N);
3  int* p = malloc(N*sizeof(int));
4  int i = 0, m = 0, j = N − 1;
5  while(i < j){
6        p[i] = −1;
7        p[j] = 1;
8        i ++; j −−; m ++;
9  }
10 p[m] = 0;
```

Figure I.1.9 – Example for out-of-bounds memory analysis.

```
1  assert(N>1);
2  int* p = malloc (2*N*sizeof(int));
3  int *p1, *p2, a;
4  *p = 8;
5  a = 10;
6  p1 = p + N;
7  p2 = p + 2 * N − 1;
8  while(p2>p1){
9        a = *p;
10       *p2 = 4;
11       p2 −−;
12 }
```

(a) Example for possible loop invariant code motion.

```
1  assert(N>1);
2  int *p = malloc (2*N*sizeof(int));
3  int *p1, *p2, a;
4  *p = 8;
5  a = 10;
6  p1 = p + N;
7  p2 = p + 2 * N − 1;
8  a = *p;
9  while(p2 > p1){
10       *p2 = 4;
11       p2 −−;
12 }
```

(b) Example I.1.10a with hoisted load.

Figure I.1.10 – Optimizing a program with code motion given a no-alias information.

of memory. Thus, an analysis that provides this information would enable hoisting the load from the loop. This optimization shown in Figure I.1.10b, tends to speed up the program because it removes a memory access (one load) at each loop iteration and enables vectorization.

I.1.4.3 Alias analysis for automatic parallelization

Parallelization can be done within loops, or between procedures using threads, but in both cases we need an alias analysis information to detect data race freedom.

Figure I.1.11a shows a procedure `parallel` that fills, in each loop iteration, two values in `p`, an integer array. Let $\mathtt{p_i} = \mathtt{p} + \mathtt{i}$ and $\mathtt{p_N} = \mathtt{p} + \mathtt{N} - \mathtt{i}$ denote the array accesses at lines 4 and 5 respectively. Since $\mathtt{i} < \frac{\mathtt{N}}{2}$, writes in $\mathtt{p_i}$ and $\mathtt{p_N}$ are always independent and can be done in parallel. Provided this "no alias" information, one can parallelize the loop which would give program in Figure I.1.11b. In some cases, for instance for the program depicted in Figure I.1.12, an inter-procedural or context-sensitive (Section I.2.2.3.3) alias analysis is required to detect the absence of data-races between the two `fill_table` calls at line 18 and 20. An automatic parallelization is then possible.

I.1.4.4 Alias analysis for instruction rescheduling

Rescheduling code means interchanging some of its statements. This option can be profitable to improve data locality, or to enable further optimizations.

```
1 void parallel(int* p, int N){
2   int i;
3   for(i = 0; i < N/2; i ++){
4         p[i] = i;
5         p[N-i] = N - i;
6   }
7 }
```

(a) Example for possible parallelization.

```
1 void parallel_thread(int* p, int N){
2   int i;
3   #pragma omp parallel for
4   for(i = 0; i < N/2; i ++){
5         p[i] = i;
6         p[N-i] = N - i;
7   }
8 }
```

(b) `parallel` procedure parallelized after no alias analysis result.

Figure I.1.11 – Use of no alias result for parallelization.

```
1  void fill_table(int* p, int n1, int n2){
2    int i, *p1;
3    for(i = n1; i < n2; i ++){
4          p1 = p + i;
5          *p1 = i;
6    }
7  }
8
9
10 int main(int argc, char** argv){
11   int pid;
12   pid = fork();
13   assert(atoi(argv[1]) < atoi(argv[2]));
14   int* p = malloc (3 * atoi(argv[2]));
15   int* q = p + atoi(argv[1]);
16   switch (pid){
17     case(0):
18       fill_table(p, atoi(argv[1]), atoi(argv[2]));
19     case(1):
20       fill_table(q, atoi(argv[2]), 2 * atoi(argv[2]))
21     default:
22       return -1;
23   }
24 return 0;
25 }
```

Figure I.1.12 – Example for data race freedom detection.

Figure I.1.13 shows a procedure `Reschedule` that fills data into two tables `A` and `B`, then copies `B` into table `C`. Since `B` is read in the first and third loop, interchanging the first and second loop would improve data locality. Such an optimization is only valid if `A` and `B` are disjoint tables and values of `B` are never overwritten by those of `A`, which is implied by a non-alias property between `A` and `B`. If this is the case, the program can be rescheduled as shown in Figure I.1.13b, and even further optimized by a loop fusion as shown in Figure I.1.13c.

I.1.4.5 Dead code elimination

Dead code elimination optimization consists in removing useless statements (generally assignments) such as overwritten variables or writes that are never read.

Let us consider again the program of Figure I.1.13a. We now suppose that `B` and `C` "must alias". Recall that must alias pointers start at the same location. Hence, there is no need to fill in table `C` in the third loop at line 10 because it has already been done in the first loop which fills table `B`. Loop at line 10 could be removed (the number of iterations is the same in both loops so the same fields are filled to and `C` receives `B` values).

```
1  void Reschedule(int* A, int* B, int* C, int L, int var){
2    int i, j, k;
3    /* ... */
4    for(i = 0; i < L; i ++){
5         B[i] += var;
6    }
7    for(j = 0; j < 2*L; j ++){
8         A[j] = 2 * var;
9    }
10   for(k = 0; k < L; k ++){
11        C[k] = B[k];
12   }
```

(a) Initial `Reschedule` procedure.

```
1  void Reschedule(int* A, int* B, int* C, int L, int var){
2    int i, j, k;
3    /* ... */
4    for(j = 0; j < 2*L; j ++){
5         A[j] = 2 * var;
6    }
7    for(j = 0; i < L; i ++){
8         B[i] += var;
9    }
10   for(k = 0; k < L; k ++){
11        C[k] = B[k];
12   }
```

(b) Procedure `Reschedule` after rescheduling.

```
1  void Reschedule(int* A, int* B, int* C, int L, int var){
2    int i, j, k;
3    /* ... */
4    for(j = 0; j < 2*L; j ++){
5         A[j] = 2 * var;
6    }
7    for(k = 0; k < L; k ++){
8         int tmp = B[k];
9         B[k] = tmp + var;
10        C[k] = tmp + var;
```

(c) Optimized `Reschedule` procedure.

Figure I.1.13 – Using alias analysis for rescheduling.

I.1.5 Conclusion and Future Work

As we have seen, alias analyses could be used to perform numerous verifications and optimizations on C code such as code motion and parallelization. Inside compilers, roles are divided into optimizing tools that transform the program, and analyzing tools that perform analyses, including alias analysis. We further explain these roles in Section I.2.5.

However, the role of optimizing tools, clients of analysis programs, is limited to the information these analyses provide them. Even though the analysis is correct and precise, transformations could not be performed if they are not provided the form of information they are expecting. In other words, analyses and transformations should speak the same language. In addition to the four types (at most) of alias responses (may, must, partial alias, and alias), the compiler might need extra facts such as relations between variables and variable offsets (lower and upper bound).

In the next chapter, we shall be interested in alias analyses inside compilers. We will hence start by discussing when such analysis could be executed compared to the program runtime and then divide the alias analysis problem into different classes and approaches. Finally, we will present some of the most used C compilers and comment on their performance to optimize programs.

Chapter I.2

Alias Analyses: Context

Contents

I.2.1 Alias Analyses Inside Compilers

Depending on when the analysis is done compared to the execution, pointer analyses can be classified into three main types: static, static with runtime checks, and dynamic.

I.2.1.1 Static analyses of pointers

Static pointer analyses are program analyses that aim to discover *at compile time* the memory locations a pointer expression can refer to *at runtime*. The main goal of being static is to avoid analysis overhead at runtime and also to enable static code optimizations based on pointer analyses.

This thesis falls within the framework of static analyses. We shall explain in deep detail this approach in Section I.2.2 and introduce some state-of-the-art techniques and contributions in the next chapters. For now, we present the second type of alias analysis which is static with runtime check.

I.2.1.2 Static analyses with runtime check

```
1  int main(int argc, char** argv){
2    int i;
3    int N = atoi(argv[1]);
4    int* p = malloc(N/2 + 42);
5    for(i = 0; i < N/2; i++){
6          p[i] = i;
7          p[i+42] = N − i;
8    }
9  return 0;
10 }
```

(a) Example illustrating the need for hybrid analysis.

```
1  int main(int argc, char** argv){
2    int i;
3    int N = atoi(argv[1]);
4    int* p = malloc(N/2 + 42);
5    if(N < 84){
6          #pragma omp parallel for
7          for(i = 0; i < N/2; i++){
8            p[i] = i;
9            p[i+42] = N − i;
10         }
11   }
12   else{
13         for(i = 0; i < N/2; i++){
14           p[i] = i;
15           p[i+42] = N − i;
16         }
17   }
18 }
```

(b) Program in Figure I.2.1a transformed based on hybrid analysis information.

Figure I.2.1 – Hybrid Analysis.

Hybrid analysis [RRH02; Alv+15] gathers information during compile time and uses them at runtime to decide whether an optimized version can be used. The goal is to reduce time and memory overheads compared to dynamic analyses (defined later) and to be more precise than static analyses. As an example of a hybrid analysis, consider the program in Figure I.2.1a. We want to parallelize the "for loop" at line 5. This is possible if memory locations `p[i]` and `p[i + 42]` at lines 6 and 7 do not access overlapping memory regions, i.e., if `N < 84`. A static analysis would conservatively answer "may alias" since pointer accesses depend on the `N`, which is unpredictable at compile time. A hybrid analysis technique such as the one given by [RRH02] applied to the original program generates the test `N < 84` at compile time and asserts it at runtime. The transformed program is depicted in Figure I.2.1b. A parallel version of the loop is run *by threads*

if the test `N` $<$ `84` is true at runtime. Otherwise, the original sequential loop is executed.

I.2.1.3 Dynamic analyses

Incomplete control-flow and input information decrease the efficiency and accuracy of static analyses. Consequently, compilers are not able to perform some code optimizations. To improve the quality of alias analysis results and to increase its efficiency while avoiding memory overheads, *dynamic analysis* [Moc03; Guo+06] techniques have been proposed. These techniques attempt to gather pointer values at runtime and check their aliasing during the program execution. The information gathered during executions, namely, the paths taken and the memory actually accessed, contribute to make the analysis more precise *on the tested inputs*.

Comparing two pointer addresses takes only one cycle and thus, for *one* execution, dynamic alias analyses should be faster than static ones. However, results of dynamic analyses are specific to that execution since it is input-dependent. It cannot be easily generalized to other inputs and analyses and optimizations should be done for each input.

Compared to static analyses, which are done once (at compile time), the overall (for a bench of executions) overhead of dynamic analyses can be more significant. The reader may refer to [Ern04; Moc+01] for further comparisons between static and dynamic analyses.

Generally static analysis algorithms provide an imprecise analysis compared to the dynamic ones since they aim to be conservative and thus give a safe estimation of pointed locations.

I.2.2 Static Alias Analyses: Approaches and Classification

Historically, static analysis of pointers has received much attention. The current state-of-the-art shows numerous analysis approaches and directions under which alias analyses may be classified.

I.2.2.1 Approaches

I.2.2.1.1 Constraint-based pointer analyses

In this approach, a set of constraints is derived from program statements involving assignments, loads, and stores. These constraints are then solved by iterating on the control flow program or on the constraint system itself until reaching a fixpoint. The final result is an abstract value for each program variable. The use of constraint sets lets us apply ad-hoc algorithms (graph algorithms, worklist algorithms, etc.) to make simpler and faster analyses. In fact, this better performance related to constraint resolution makes such kind of analysis efficient for programs with millions of lines of code [Fäh+98]. For these reasons, we adopt, in Part II, this approach to build new alias analysis algorithms on top of the LLVM compiler.

I.2.2.1.2 Pointer analyses by Model Checking

The term Model Checking was coined by Clarke and Emerson [CE81] in the early eighties. It can be considered as an extension of automated protocol validation techniques done by Hajek [Haj78] and West [Wes89] since it examines the entire state space of a system. These techniques were restricted to checking the absence of deadlocks or live-locks. To do so, the relevant features of a given code are represented as a finite state automaton in which the model checker verifies the

validity of a given property generally expressed as a temporal logic formula. A counterexample is generated if the property is not satisfied.

In [MP01], Martena *et al.* introduced a novel prototype of alias analysis by means of model checking techniques. The key insight of their technique is to transform the alias query into an unreachability analysis. A simple test of aliasing between a pair of pointers followed by a null (`skip`) are added on demand. The `skip` is executed if the test is verified. Finally, to answer the alias query they check the satisfiability of "null" reachability, *i.e.*, if null is unreachable, then pointers do not alias. Since the program to be analyzed may have a huge number of states, the main challenge of this idea is to use *abstractions* to reduce them. Martena *et al.* present a scheme to translate the program into *Promela*, a `C` like language, used by the *Spin* [Hol97] Model Checker without losing the correctness of the results. However, although the performance in terms of verification time and scalability seems promising, we believe that such kinds of program transformations cannot be embedded in compilers, in particular because tracking back information from results obtained after transformation is tedious and error prone.

I.2.2.1.3 Shape Analyses

Shape analysis [SRW98; Cal+06], also known as *storage analysis*, is a kind of static alias analysis mostly used to verify programs that perform destructive updating on dynamically allocated storage. Its goal is to give, for each program point, a finite and conservative characterization of the possible *shapes* of program data structures allocated in the heap. A Static Shape Graph (SSG), which is a finite, directed, and labeled graph, is computed at each program point to approximate stores done during the execution and until this point. The SSG, where nodes are structures and edges are pointers, is used to check memory safety (looks for memory leaks), and to find out-of-bounds accesses and discover properties of linked dynamically allocated data structures. Graphs are used to model program stores. Much attention has been paid to this technique in the last decade. However, the effort has mainly been done on precision issues. The state-of-the-art [LCR15] is able to express complex properties about graph structures, but the current version of the analyzer only deals with tiny programs (20 to 30 lines of code). These analyses are indeed too greedy in terms of memory resources to be embedded in state-of-the-art compilers.

I.2.2.2 Classic algorithms: Andersen's and Steensgaard's

The best known techniques of pointer analyses are perhaps those introduced by Andersen [And94] and Steensgaard [Ste96]. These analyses, presented below, are constraint-based. To solve a set of constraints they use efficient graph algorithms.

I.2.2.2.1 Andersen's analysis

Andersen's algorithm [And94] is a constraint-based points-to analysis computing $\texttt{pt(p)}$ for each pointer `p` such that $\texttt{pt(p)}$ denotes the "points-to set" of `p`. Constraints are classified by [HT01] into 3 classes: base, simple, and complex. The statement $\texttt{a} = \&\texttt{b}$ gives the base constraint $\texttt{b} \in \texttt{pt(a)}$ where $\texttt{pt(a)}$ denotes the points-to set of `a`. The simple statement $\texttt{a} = \texttt{b}$ implies $\texttt{pt(b)} \subset \texttt{pt(a)}$. Complex constraints refer to load and store statements. $\texttt{a} = {}^*\texttt{b}$ generates $\forall\ \texttt{x} \in \texttt{pt(b)}, \texttt{pt(x)} \subset \texttt{pt(a)}$ and ${}^*\texttt{a} = \texttt{b}$ gives $\forall\ \texttt{x} \in \texttt{pt(a)}, \texttt{pt(b)} \subset \texttt{pt(x)}$. After collecting constraints, the points-to problem is solved with efficient graph algorithms. It amounts to computing the transitive closure on the constraint graph where nodes are pointer variables and edge from node `a` to node `b` means that the points-to set of `a` is a subset of the points-to set of `b`.

This algorithm runs in the worst case in $O(n^3)$ where n is the number of assignments in the program. Andersen's algorithm is an inclusion-based algorithm. He assumes that after a pointer copy, one points-to set is a subset of the other. In other words, points-to set of `b` is included into the points-to set of `a` after `a = b`. Figure I.2.2b gives the constraint graph of Andersen's algorithm run on program of Figure I.2.2a.

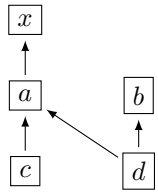
I.2.2.2.2 Steensgaard's analysis

Steensgaard's algorithm is also a constraint-based algorithm. It introduces a unification based analysis, the simple statement of assignment `a = b` generates `pt(a) = pt(b)`. As depicted in Figure I.2.2c, nodes `b` and `a` are collapsed so that pointers `d` and `c` have the same points-to set after statement `d = c` (line 4 Figure I.2.2a). This algorithm represents in fact one of the first approaches that proposed unification-based techniques to solve the points-to analysis problem. Although Steensgaard's analysis is very fast in practice and is almost linear in time [Ste96], its main drawback is the lack of precision introduced by the unification-based approach, when compared to the inclusion-based one. Example 4 illustrates the difference in precision between both approaches.

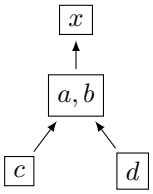
Example 4 *Consider the program snippet in Figure I.2.2a. The difference between both analyses comes from statement* `d = c`. *In the inclusion based approach (Andersen), only the points-to set of* `a` *is included into the points-to set of* `d`. *However, in the unification based (Steensgaard), the points-to set of* `b` *is also included into* `a`*'s one. This approach introduces a loss of precision since* `c` *may points to* `b` *now.*

1: $a = \&x$;
2: $c = \&a$;
3: $d = \&b$;
4: $d = c$;

(a) Code



(b) Inclusion based analysis.



(c) Unification based analysis.

Figure I.2.2 – Andersen (left) and Steensgard (right) analyses, comparison.

I.2.2.3 Classification

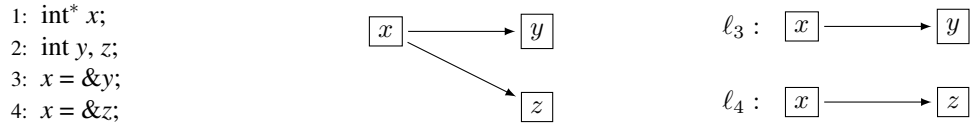
Depending on the kind of information they track and the type of abstraction they do, static analyses can be classified in different ways.

I.2.2.3.1 Flow sensitivity

A flow-insensitive analysis ignores the order of the statements of the program. On the contrary, a flow sensitive analysis is capable of propagating flow information, as a result it is capable of storing a different information per control point of the program.

The main trade-off between flow-sensitive and flow-insensitive analyses is scalability/precision. In Figure I.2.3, we compare results given by each kind of analysis run on the program of Figure I.2.3a. The points-to set of the flow-insensitive analysis illustrated by Figure I.2.3b contains both `y` and `z` at the end of the analysis (all statements of the program are evaluated globally), thus

before control point ℓ_4, x can take any value in $\{\&x, \&y\}$. On the contrary (Figure I.2.3c), a flow-sensitive analysis should be capable of deriving two different alias sets for control points ℓ_3 and ℓ_4.



(a) Code (b) Flow insensitive analysis. (c) Flow sensitive analysis.

Figure I.2.3 – Flow sensitivity in pointer analysis.

Note that both Andersen's and Steensgaard's algorithms are flow-insensitive. The difference in precision is related to inclusion and unification approaches.

I.2.2.3.2 Sparse analysis and Dense analyses

Classicaly, in abstract interpretation as well as in compilation dataflow analyses, information is propagated along the control flow and stored at *each control point* (or at least at each point belonging to a cutset of the control flow graph, [Bou93]). However, this kind of analysis, called *dense*, adapted to the computation of complex information such as numerical polyhedral invariant, does not scale well. To overcome this difficulty, *sparse* dataflow analyses propose to assign a unique information to each variable which is *invariant in its entire life range*.

Sparse pointer analysis implementations often tend to show better time and space performance compared to dense ones [CCF91; Oh+12]. Actually, as demonstrated by Choi *et al.*, the main advantage of a sparse analysis is efficiency: the product of the analysis - the information that is bound to each variable - requires $O(N)$ space, where `N` is the number of variable names in the program.

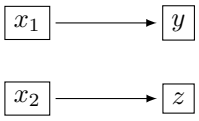
Designing a sparse version of a given dense analysis is not immediate, and it implies reasoning about:

- the abstract domain itself: a given information must be assigned to variables, thus any global information must be dispatched among variables. Relational analyses like linear relation analysis [CH78] can not be easily sparsified.
- the representation of the program: as we will see later, we have to invent suitable *splitting strategies* for our information to be attached to a unique variable name.

Flow-insensitivity under SSA form Flow-sensitive analyses are usually more precise and more costly in terms of time and space than flow-insensitive ones. However, the gap between both techniques can be reduced by using the static single assignment form (Section I.1.2). Recall that splitting the live-range of a variable `v` lets us associate an information directly to `v`. In [HH98], Hasti and Horwitz prove that using SSA form produces in a pointer flow-insensitive algorithm results as good as those of a flow-sensitive technique. To that end, they normalize programs to eliminate pointer dereferences and translate them to SSA form.
Going back to Figure I.2.3a, we show how a classic flow-insensitive pointer analysis applied to the program transformed in the standard SSA form gives the same results as a flow-sensitive technique. Figure I.2.4a gives the SSA form. Note that pointer `x` has been renamed to $\texttt{x}_1$ and $\texttt{x}_2$. Figure I.2.4b illustrates a flow-insensitive analysis of the program under SSA form. We note that compared to the flow-sensitive analysis on the original program given in Figure I.2.3c, results are the same.

1: int* x;
2: int y, z;
3: x_1 = &y;
4: x_2 = &z;

(a) Program of Figure I.2.3a under SSA form.



(b) Flow-insensitive analysis with SSA form.

Figure I.2.4 – Using SSA form to improve flow-insensitive pointer analyses. Note that using two different names for `x` lets us track its different behaviors in a flow-insensitive fashion.

I.2.2.3.3 Interprocedural and Intraprocedural analyses, context sentifivity

Intraprocedural analysis analyzes each single function regardless of its callers or callees. It is simple but makes conservative assumptions due to the lack of information about the calling context. On the contrary, interprocedural analysis is more precise since it collects information across function boundaries. The analysis is propagated from callers to callees in a *top-down* approach, or from callees to callers in a *bottom-up* approach. To deal with procedure calls, interprocedural analyses may build a control flow supergraph. Another used technique consists in computing summary information for each method. A summary stores effects of the callee on global variables and callers, and information from callers. A third approach to interprocedural analysis consists in inlining callees: copying the callee at each call site to be analyzed with its caller. A new copy is indeed needed for each call site, which is costly.

Context Sensitivity Context sensitivity refers to function analyses.

A context-insensitive analysis merges (or ignores) all the context information coming from all function call sites. A given function is analyzed once with respect to this unique information. In the most imprecise version, all parameters of the function to be analyzed are considered to be unknown.

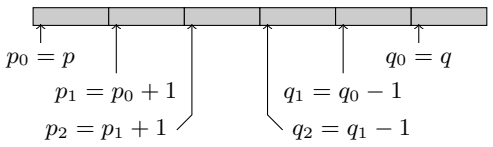
On the contrary, a context-sensitive analysis propagates information across function boundaries. It corresponds to a *top-down* interprocedural analysis approach. To illustrate the difference, let us consider the program in Figure I.2.5a in which the `main` function calls `f` with two different values: 4 and 5. We want to disambiguate pointers `p` and `q` within the same loop iteration in function `f`. Note that stores at lines 7 and 8 may be executed in parallel if these pointers do not reference the same memory locations. Depending on `x` value given as input to `f`, `p` and `q` point to the same or to different locations in the last loop iteration. A conservative context-insensitive pointer analysis returns a may alias information for pointers `p` and `q` in `f`. If we analyze the program with a context-sensitive analysis, we consider independently both call sites `f(4)` and `f(5)`. Hence, we conclude that `p` and `q` may alias in the same loop iteration if `x` = `4` while they do not if `x` = `5`. Figures I.2.5b and I.2.5c gives a memory scheme in each case.

Due to the large number of considered contexts and which grows up exponentially in the length of the program call-chain, most proposed approaches in literature face scalability problems.

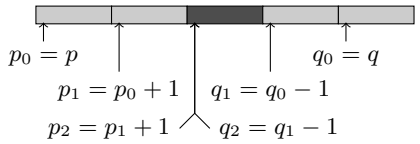
Actually, context-sensitive analyses are expensive for a comparable precision [LH06]. To handle large programs without sacrificing precision, some context-sensitive analyses reduce the number of analyzed contexts. In [Oh+14], Oh *et. al* use a pre-analysis to determine whether a context sensitive analysis is requested or not to improve precision. A context-sensitivity parameter K is constructed to select which calling contexts should be analyzed by the main analysis. Instead of selectively choosing analyzed contexts, [WL04] Whaley *et al.* handle context sensitivity by running a context-insensitive analysis on a cloned graph that contains all methods of the program,

```
1  void f(int x){
2    int* p = malloc((x+1)*sizeof(int));
3    int* q = p + x;
4    while(p < q){
5         p ++;
6         q --;
7         *p = 1;
8         *q = 2;
9    }
10 }
11
12
13 int main(int argc, char** argv){
14   f(4); // call site 1
15   f(5); // call site 2
16 return 0;
17 }
```

(a) Program with different function call sites.



(b) p and p do not alias, x = 5.



(c) p and p alias, x = 4.

Figure I.2.5 – Example to study context sensitivity.

but one per context. Whereas these analyses reduce the scope of context sensitivity to scale, in [VR01] Vivien and Rinard vary the interprocedurability application in a full context-sensitive fashion. The points-to technique they introduce starts by running an intra-procural analysis that could be incremented by an interprocudural one. The goal is to analyze just enough of the program to capture objects of interest.

I.2.2.3.4 Field Sensitivity.

Field sensitivity deals with `struct` data structures. It handles differently fields and field aggregates. Besides field-sensitive and field-insensitive analyses, a common classification of field sensitivity defines what we call a *field-based* approach. A field-insensitive analysis treats all fields of an aggregate as a single variable. A field-based approach however, treats all aggregates of a field as a single variable. Field-sensitive approach, being the most precise one, models each field of an aggregate with a single variable. To illustrate the difference, we consider the program in Figure I.2.6 and let $\mathtt{s_1}$ and $\mathtt{s_2}$ be two instances of struct `S` with fields $\mathtt{f_1}$ and $\mathtt{f_2}$.

Program	Field-insensitive	Field-sensitive	Field-based
typedef struct {int* $\mathtt{f_1}$, int* $\mathtt{f_2}$} S;			
S $\mathtt{s_1}$, $\mathtt{s_2}$;			
int a, b, c, *d;			
$\mathtt{s_1}$.f1 = &a	$s_1 \supset \{a\}$	$s_{1_{f_1}} \supset \{a\}$	$f_1 \supset \{a\}$
$\mathtt{s_1}$.f2 = &b	$s_1 \supset \{b\}$	$s_{1_{f_2}} \supset \{b\}$	$f_2 \supset \{b\}$
$\mathtt{s_2}$.f1 = &c	$s_2 \supset \{c\}$	$s_{2_{f_1}} \supset \{c\}$	$f_1 \supset \{c\}$
d = $\mathtt{s_1}$.f1	$d \supset \{s_1\}$	$d \supset \{s_{1_{f_1}}\}$	$d \supset \{f_1\}$

Figure I.2.6 – Example to illustrate the difference between field-sensitivity approaches.

In this example, we can see the main difference between the three approaches in the last statement d = $\mathtt{s_1}$.f1 where we loose precision in both insensitive and filed-based analyses as we conclude $d \supset \{a, b\}$ and $d \supset \{a, c\}$ respectively. However, a field sensitive analysis gives $d \supset \{a\}$.

Many proposed field-sensitive analysis approaches [BS16; Min07; PKH04] focused on precision at the expense of scalability. In [PKH04], Pearce *et al.* present a flow- and context-insensitive pointer analysis to model `struct` fields and indirect function calls. To be field-sensitive, they extend Andersen's constraint set and reference variables as offsets from others. We here recall the new added constraints:

$$\mathtt{q} \supseteq *(\mathtt{p}+\mathtt{k}) \mid *(\mathtt{p}+\mathtt{k}) \supseteq \mathtt{q} \mid *(\mathtt{p}+\mathtt{k}) \supseteq \{\mathtt{q}\}$$

where `q` is a pointer program variable, `p` is the *object* (struct or function) pointer and `k` the offset. For instance, the *i-th* field of a struct pointer `p` is referenced by $\mathtt{p}+\mathtt{i}$. Hence, $\mathtt{q} = \&(\mathtt{p} \rightarrow \mathtt{f_i})$ gives $\mathtt{q} \supseteq \mathtt{p}+\mathtt{i}$: the points-to set of `q` is a super-set of the points-to set of $\mathtt{p}+\mathtt{i}$. This form is represented by an edge of weight `i` added to the constraint graph. Recently, based on the line of work of Pearce *et al.*, Balatsouras *et al.* [BS16] proposed an approach to maintain a better level of precision by keeping track of types for abstract objects (stack or heap allocations, fields, elements of arrays, ...). Each abstract object has then a single type even heap allocations used to allocate objects of different types (abstract objects are duplicated with one copy for each type). In this work, we still note that precision comes at the expense of scalability due to the important number of abstracted objects of known types. Since scalability is an important issue for alias analysis to be embedded in compilers, we will not handle field-sensitivity and thus sacrifice the precision related to `structs`.

Our main goal of studying static alias analyses is in fact contributing to compiler construction to make them more efficient for `C`-like languages. After presenting analysis approaches, we now move to study the architecture of some `C` compilers and to evaluate their efficiency.

I.2.3 Architecture of C Compilers

Compilers are responsible for transforming abstracted programming languages into machine code. A compiler includes, *at least*, two main components that process the input source file. These components, depicted in Figure I.2.7 are: the front-end and the back-end. The front-end parses the source code and generates an **I**ntermediate **R**epresentation (IR). It is source language dependent and target machine independent. The IR will be analyzed and optimized in the back-end. Depending on the machine architecture, the back-end generates the machine code based on the optimized representation. Sometimes, the compiler will have a third part which stands between the front-end and the back-end. This is what we call the *middle-end*. Its role is to perform independent source and target optimizations on the Intermediate Representation. Advantages of such decomposition inside compilers may be found in [CIA12, §1.3].

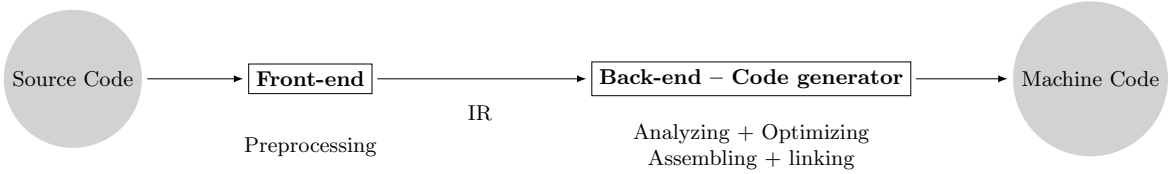


Figure I.2.7 – Compiler's components and processes.

The list of C compilers used in research and industrial fields is very long. These compilers differ mainly in their architectures but also in the type of license, compatibilities, and standard they support. In the sequel, we present four of C compilers that we could use to develop static alias analyses for with a focus on the LLVM compiler.

I.2.3.1 GCC

The **G**NU **C**ompiler **C**ollection (GCC) is a collection of front-ends of C, C++, and Java among others and their libraries. GCC is originally written as *the* compiler of the GNU operating system and is composed of the three parts: front-end, middle-end, and the back-end. In the context of this thesis, we are particularly interested in gcc, the C front-end of GCC compiler. After parsing the input code, gcc converts it to the *Abstract Syntax Tree* (AST) representation. AST is then transformed to *GENERIC*, a unified version of AST which might slightly differ from an entry language to another. In the middle-end called *tree optimizer*, *GENERIC* is transformed to *GIMPLE*, a three-address representation derived from *GENERIC*. That is to say, each instruction has at most three operands. The gcc IR is then put into the SSA form used to perform code optimizations. Before generating the machine code in the back-end, optimized IR is converted back to *GIMPLE* (after the *SSA destruction* phase (Section I.1.2)) then to *Register-Transfer Language* (RTL) where further optimizations are performed.

I.2.3.2 ICC

ICC is the **I**ntel **C** **C**ompiler. The main purpose of ICC is to boost the performance of C/C++ programs by making them run faster on Intel architectures. Actually, it takes further advantage of the abundant number of cores and the vector register width available in certain Intel processors such as Intel® Xeon® processors. Its optimizations are mainly based on loop optimization and auto vectorization for data-parallelism. Intel claims ICC delivers better performance compared to gcc and clang (defined later in Section I.2.3.4) when run on Intel microprocessors. Figure I.2.8 illustrates a comparison between these compilers when computing SPEC2006 *SPEC_rate_base performance*[1]. We borrowed chart figures from Intel ICC Overview.[2]

ICC, the Intel compiler is available *for sale* among other Intel products such *Intel Parallel Studio XE* [3] tool. Therefore, it might be a good infrastructure and a powerful compiler but since it source code is not available, we are not able to make our own alias analysis contributions on top of it.

I.2.3.3 PIPS

PIPS is a *research* compilation framework designed to perform an Interprocedural Paralellization for Scientific Programs. ("***P**arallélisation **I**nterprocédurale de **P**rogrammes **S**cientifiques*"). This project was initiated in 1988 to automatically analyze and transform scientific applications. PIPS is a *source-to-source* compilation framework that handles C and Fortran languages. This very framework is designed to perform several source-to-source transformations and optimizations such as parallelizations in the CUDA, OpenCL, and OpenMP programs and adding directives to Fortran and C programs. Therefore, it does not support all the back-end functionalities of a compiler. Although PIPS is a compiler deeply involved in the pointer analyses line of research and aims to carry out optimizations based on such information, we do not consider it as a suitable framework to develop the new alias analyses we shall design. In fact, this compiler is

[1] SPEC_rate_base is the Geometric Mean of SPEC when compiled with base tuning. It is a metric to measure the performance on SPEC CPU benchmark suite https://www.spec.org/gwpg/gpc.static/geometric.html. For SPEC CPU200, each benchmark is run tree times and each runtime is measured. The median of three runtimes is used to compute the Ratio of that program using the formula: $Ratio = 100 \times RefTime/RunTime$ where $RefTime$ denotes the reference machine (Sun Ultra 5/10 300MHz) runtime. Ratios of all programs are finally used to compute the Geometric Mean.

[2] https://software.intel.com/en-us/c-compilers/ipsxe

[3] https://software.intel.com/en-us/intel-parallel-studio-xe

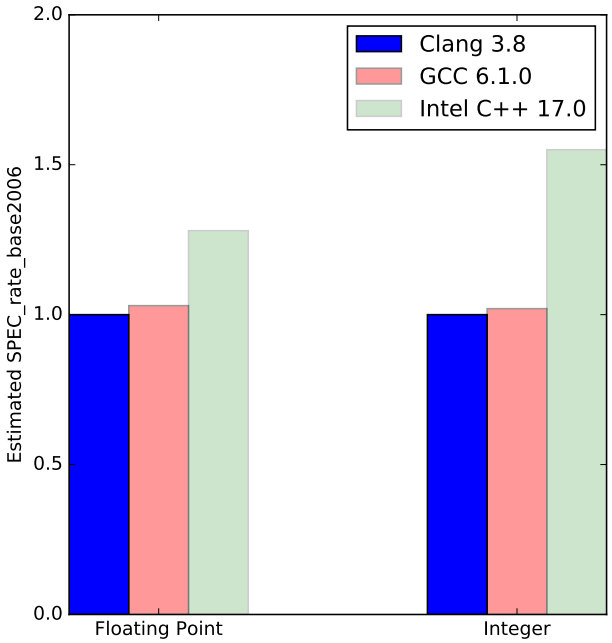


Figure I.2.8 – Comparing relative geomean performance of clang, gcc, and icc run on Intel devices. X axis represents SPEC CFP2000 and SPEC CINT2000. Y axis shows measured performance: the higher, the better.

still a research project and lacks documentation. As we mentioned before, it is a source-to-source framework which exceeds the context of this thesis and adds an extra complexity for an extra precision we may not need. The third reason lies in the way PIPS handles optimizations. Notably, some of them are enabled via the pips-make mechanism while others are not automatically enabled whenever the needed alias analysis information is provided, but on user's demand. This is due to the source-to-source design and the lack of information related to the back-end architecture. This is a limitation in our sense since compared to a production compiler, we may evaluate pointer analyses in terms of a set of interacting optimizations.

I.2.3.4 LLVM

LLVM is now the full name of a production compiler that has begun as a research project at the *University of Illinois* as the *Low Level Virtual Machine*. Introduced by Lattner and Adve [LA04] in 2004, this project has contributed to a collection of modular and reusable compiler and tool-chain technologies. It defines an intermediate language-independent code representation which is based on the Static Single Assignment form [Cyt+91]. LLVM handles high level languages (C/C++, Objective-C, Java, etc.) and is able to analyze, transform and optimize arbitrary programs both at compile and runtime. The LLVM project is widely used in research projects and industrial production. It is licensed under the "UIUC" (University of Illinois Urbana-Champaign) license. It is an open source license but not a *leftright* one. That is to say, commercial products can be derived from LLVM. In this thesis, we shall be using clang as an LLVM front-end. Clang is a C/C++/Objective-C and Objective-C++ compiler front-end about three times faster than gcc[4] (for Objective-C programs in debug mode).

In LLVM, the optimizer stands alone as a middle-end between the front and the back-end and is responsible for analyzing and optimizing LLVM Intermediate Representation. The LLVM compiler design is given in Figure I.2.9.

[4] http://llvm.org/

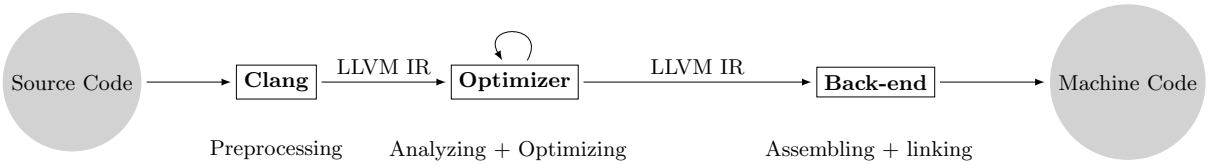


Figure I.2.9 – The components and processes of the LLVM compiler.

The LLVM *Intermediate Representation* (LLVM IR) is the language used by the LLVM compiler to represent code. Similar to RISC (Reduced Instruction Set Computing) and *GIMPLE* instruction set, LLVM IR instructions handle linear simple instructions of a given language like `add`, `subtract`, `load`, and `store`, and is in three-address form. However, unlike RISC that supports a reduced number of data types, LLVM IR is a well typed language. A variable `v` of integer type is represented by `int32 %v` to denote a 32-bit integer. `int32*` `%v` is a pointer to a 32-bit integer. This is indeed useful to perform *type-based* analysis like [BS16; YH04] and also makes the low-level language human readable. In Figure I.2.10 we give a simple C code example and its LLVM IR representation.

```
1 int LLVMIRfunc(int a){
2   return a + 42;
3 }
```

(a) C code to be transformed to LLVM IR.

```
1 define i32 @LLVMIRfunc(i32 %a) #0 {
2 entry:
3   %add = add nsw i32 %a, 42
4   ret i32 %add
5 }
```

(b) LLVM IR of code in Figure I.2.10a.

Figure I.2.10 – Example for LLVM IR representation.

As depicted in Figure I.2.9, inputs and outputs of the middle-end are both in LLVM IR representation in the LLVM compiler. Output IR is then equivalent to the input one *but optimized*.

The LLVM IR is an SSA form-based representation. As we discussed in Section I.1.2, SSA form transformation is less straightforward for address-taken variables due to the existence of indirect definitions and uses through pointers. To handle this complication, LLVM does not even place these variables under SSA. In fact, new names for top-level variables are created thanks to the *memory to register* (`mem2reg`) LLVM transformation which promotes memory references to be register references using unlimited number of virtual registers. In the contrary address-taken variables are kept in memory. No new names are needed to ensure the unique definition site for each variable since any access to that variable is done through loads and stores. To better understand this approach, let us consider the program code in Figure I.2.11. As we said before, the naive renaming of address-taken variables is unsound. Variable `a` is an address-taken variable defined at line 2, indirectly modified at line 5 and used at line 6 while variable `c` is a top-level variable defined at line 3 and used at line 6. We will show how LLVM handles differentely these two classes of variable using variables `a` and `c`.

Figure I.2.12 illustrates the LLVM IR of code in Figure I.2.11. Note that at line 3 of the LLVM IR representation a memory slot (in the stack) has been allocated to variable `a` (address-taken). Variable `c` (top-level) is calculated at line 6 and stored into the register `%add`. Let us now see how these variables are used at line 9. Variable `c` is simply propagated using its new name `%add`. However, variable `a` is loaded from memory at line 8. Now its new name is `%tmp1`. This way, LLVM garantees the *variable def-use chain* for both top-level and address-taken variables (Section I.1.2).

```
1 int* addressTaken(int* q){
2   int a = 2;
3   int c = *q + 4;
4   int* p = &a;
5   *p = 3;
6   int b = a + c;
7 return p;
8 }
```

Figure I.2.11 – Program with address-taken variable.

```
1  define i32* @addressTaken(i32* %q) #0 {
2  entry:
3    %a = alloca i32, align 4
4    store i32 2, i32* %a, align 4
5    %tmp = load i32, i32* %q, align 4
6    %add = add nsw i32 %tmp, 4
7    store i32 3, i32* %a, align 4
8    %tmp1 = load i32, i32* %a, align 4
9    %add1 = add nsw i32 %tmp1, %add
10   ret i32* %a
11 }
```

Figure I.2.12 – LLVM IR representation of program in Figure I.2.11.

I.2.4 The LLVM Compiler

In this section, we will be mainly interested in the LLVM compiler and shall start by showing why we choose this very compiler to design and implement new alias analyses. We will indeed compare LLVM to gcc in terms of performance and implementation issues. Recall that we do not consider the Intel compiler icc option since it is a compiler for sale and therefore not open source. The PIPS framework exceeds yet the context of this thesis in terms of architecture and design. We then move to explain in deep detail the LLVM "pass" framework.

I.2.4.1 LLVM vs. gcc

Differences between LLVM and gcc are abundant and are mainly related to the middle-end and back-end of both compilers. We believe gcc may have some good points compared to LLVM. For instance, gcc generates code for far more targets than LLVM can. However, in this section, we will focus on pros of LLVM compared to gcc that made us choose LLVM to study and develop alias analyses for.

Compared to *GIMPLE*, LLVM IR has a reduced number of instructions that need to be handled by a static analysis thanks to its new instruction set. In fact, in the LLVM IR we note the absence of copy instruction and unary operations which are replaced by an implicit copy propagation done on the fly and by binary operations. Moreover, LLVM IR is not only a strongly typed representation but also a well-defined one. The instruction reference[5] is detailed in the LLVM documentation and illustrated via examples. Whereas, GCC documentation provides a weak overview of GIMPLE. This makes the analysis of LLVM IR much easier compared to gcc IR.

LLVM, as a modular optimizer, is based on a "pass" infrastructure that enables users to easily contribute to the compiler. It offers in fact support for developers to fit their code into the system [LA03] as the programmer may check the consistency of his code and whether the new pass is corrupting the input of the other passes or not using the consistency checker. This utility is automatically run after passes in development mode and is able to detect SSA property violation and memory leaks in the code. The LLVM pass manager provides the developer with information related to other passes in order to enable him to choose the most suitable order for running his pass among others. These information are related to the required *pre-passes* and those have being destroyed after a pass has being running. Unlike LLVM, gcc middle-end is not modular. One can dump the IR out, alter the code, visualize optimization, and then make the compiler read it back easily and without rebuilding the whole compiler. Regarding optimizer

[5] The reader may refer to http://llvm.org/docs/LangRef.html#instruction-reference for an exhaustive list and definitions of LLVM IR instructions.

performance, Lattner *et al.* show in [LA03] how efficient are LLVM analyses and optimizations compared to gcc ones. Their experiments on 176.gcc benchmark of SPEC CPU2000 show that LLVM equivalent *Common Subexpression Elimination* (`cse`) optimizations are more powerful and more than two times faster than gcc ones.

I.2.4.2 LLVM passes

We mainly distinguish two classes of passes: analysis passes that analyze the intermediate representation of the code and optimization passes that, based on analysis results, alter the code to optimize it. Passes are run by the LLVM optimizer on three levels: O1, O2, and O3. The O0 is the no optimization level. The list of these passes is maintained by the LLVM pass manager which keeps the list of available analysis information and its last user at each state of the pass execution pipeline. However, the pass manager is not responsible for finding an optimal optimization sequence for a given program, which is indeed unpredictable.

I.2.4.2.1 Analyses passes for pointers

Analysis passes do not alter the program IR. They collect information to be used later by transform passes. In this thesis we shall be interested in alias analysis passes built on top of the LLVM compiler. We fit our new developed techniques to the system and shall compare their performance in terms of number of disambiguated pairs of pointers against the main LLVM analyses in Chapter II.1 and Chapter II.2. Their applicability in terms of optimizations performed shall be discussed in Chapter II.4. We use `-aa-eval` for *Exhaustive Alias Analysis Precision Evaluator* to perform pairwise alias queries. Although many analyses were proposed and evaluated in the state-of-the-art, (the reader may refer to Section I.2.1 for state-of-the-art details), only five alias analyses are fully implemented and available within the LLVM distribution:

1. *basicaa*: for basic alias analysis. It is currently the most effective alias analysis in LLVM, and is the default choice at the -O3 optimization level. It relies on a number of heuristics to disambiguate pointers[6]:
   - Distinct globals, stack allocations, and heap allocations can never alias.
   - Globals, stack allocations, and heap allocations never alias the null pointer.
   - Different fields of a structure do not alias.
   - Indexes into arrays with statically differing subscripts cannot alias.
   - Many common standard C library functions never access memory or only read memory.
   - Pointers that obviously point to constant globals "pointToConstantMemory".
   - Function calls can not modify or reference stack allocations if they never escape from the function that allocates them.

2. *globalsmodref-aa*: It is a simple implementation of a context-sensitive mod/ref[7] and alias analysis. This analysis is able to prove a non alias global pointer if its address is never taken. It also keeps track of functions that do not read or write memory.

3. *steens-aa*: It offers an implementation of a variation of the Steensgaard's algorithm. This approach is described in Paragraph I.2.2.3.2.

[6] This list has been taken from the LLVM documentation, available at `http://llvm.org/docs/AliasAnalysis.html` in August of 2016

[7] The mod/ref information indicates whether the execution of an instruction can read or modify a memory location.

4. *ds-aa*: It denotes a data structure analysis. This analysis is a unification-based, flow-insensitive, context-sensitive, and field-sensitive alias analysis. It runs in $\mathcal{O}(n \times \log(n))$.

5. *scev-aa*: for scalar evolution alias analysis. This analysis tries to infer closed-form expressions for the induction variables used in loops. For each loop such as: `for(i = B; i < N; i+ = S){...a[i]...}`. This analysis associates variable `i` with the expression $\texttt{i} = \texttt{B} + \texttt{iter} \times \texttt{S}, \texttt{i} \leq \texttt{N}$. The parameter `iter` represents the current iteration of the loop. With this information, *scev* can track the ranges of indices which dereference the array `a` within the loop. Contrary to our analysis, *scev* is only effective to disambiguate pointers accessed within loops and indexed by variables in the expected closed-form.

Analysis information are in fact used by optimization passes to enable or not the optimizations they could perform. There are three types of alias analysis results and uses:

- MemoryDependenceAnalysis: gives information about stores feeding loads. Such analysis is used for instance by the *Dead Store Elimination* and *Global Value Numbering*.
- AliasSetTracker: instead of generating pairs of aliasing pointers, the AliasSetTracker provides to optimization passes, the set of pointers aliasing (or may alias) a given pointer. A pointer with no aliases forms a singleton. An example of optimization pass using the AliasSetTracker is *Loop Invariant Code Motion*. A loop invariant pointer loaded from memory can be hoisted out of the loop if its alias set does not include a pointer which is modified within the loop. If an alias set contains only loop invariant pointers stored to and must alias, then these stores can be sunk out of the loop.
- AliasAnalysis interface: it provides an alias information about pairs of pointers. Recall that this can take four shapes: MustAlias, PartialAlias, MayAlias, or NoAlias.

Besides the fact that passes still cannot offer a strong efficient analysis as we shall discuss in Section I.2.5, the alias analysis infrastructure of LLVM shows many limitations related to alias analysis chaining behavior. Analysis information is consumed by optimization passes to perform optimizations. Therefore, it would be useful if all optimization passes could take profit from all analysis passes including the analysis developed by users and dynamically compiled in LLVM. For instance, if `myAApass` is a new alias analysis pass, to date, it is not possible in LLVM to write `opt -myAApass -O3` to combine `myAApass` analysis with other analysis passes and use alias information to enable more optimization without changing the source code. The `myAApass` analysis is only useful when called before a specific optimization pass. A couple of analysis passes can thereby be combined. Note that this is not the case of *basicaa* analysis which is called *"ImmutablePass"*. In other words, it is run only once and the alias results are updated through the whole execution using the methods Copy/delete/replace Value responsible for keeping the alias information consistent since the transform passes may alter the source code.

I.2.4.2.2 Transform passes

In LLVM, *transform passes* are passes that alter the program code in its IR representation. Transformations applied to the program might be for analysis or optimization issues. The e-SSA form presented in Section I.1.2 is an example of program transformation for analysis purposes. We recall that e-SSA form is a program transformation derived from SSA form to split variable live ranges and perform efficient analyses. Note that LLVM does not contain an official implementation of such technique at the time of this writing. Optimization passes are analyses passes clients. They transform the code for a better *runtime/memory consumption* trade-off. In LLVM, there are numerous optimization passes related to dead code, loops, global variables, etc. As we discussed in Section I.1.3.3.1, in a perfect world, one would efficiently evaluate an alias analysis through its impact on optimization passes. However, as we shall discuss in Chapter II.4,

the LLVM framework still lack opportunities to properly handle optimizations based on alias analyses.

For the sake of completeness, we give a survey of some LLVM optimizations that, most luckily, might perform optimizations based on alias analyses inside LLVM.

- GVN: *Global Value Numbering* is sometimes used to eliminate redundant code. The same value is assigned to equivalent variables or expressions to avoid redundant instructions. We shall give an example of this optimization with alias analysis in Section I.2.5.2.
- LICM: *Loop Invariant Code Motion* attempts to remove as much as possible invariant code from loops, either by hoisting them into the pre-header block or sinking them into the exit block when it is safe to do it. It uses alias analysis for instance to remove loads from the loop if it aliases anything stored to.
- DSE: *Dead Store Elimination* eliminates trivial dead stores within the basic-block. Alias analysis is needed for instance to safely remove redundant stores after checking that they do not overlap with alive pointers.

I.2.5 Motivation for Pointer Analyses Inside Compilers

The output of alias analyses do not depend on whether their results are used for verification or optimization. However, what matters most is the efficiency and scalability of the approach. In this section, we evaluate the precision of the current analyses inside production compilers, and show that there is a huge gap between theory and practice. Note that for the sake of simplicity, we will show experiments for a simple, yet representative program. In the first subsection, the precision of C compilers is studied using a pattern typically found in linear algebra and in sort programs. For linear algebra, for instance in polybench[8][YP15] programs, nested loop iterators are compared and defined from each other to access matrices. In sort algorithms, like quicksort, and in partitioning algorithms, given base pointers some related pointers and offsets are compared, and are used to access arrays. Many examples of accesses to different array offsets and of pointer and offset comparisons may be found in the libc and polybench methods.

I.2.5.1 C compilers: a lack of precision?

In this section, we will study some of the production C compilers (CGG, LLVM, and ICC) to show that the alias analyses inside compilers are far from being satisfactory. We show how these three compilers handle the simple example of Figure I.1.10a (that we recall below) with the maximum level of optimization (O3).

I.2.5.1.1 GCC

In this section, we study assembly code generated by GCC 4.9.2 (`gcc -S`) when applied to the code snippet in Figure I.1.10a with O3 optimizations. Simplified results are depicted in Figure I.2.13.

The C compiler gcc at this level of optimization is able to find out that the loop will be executed at least once. Thus, it removes the assignment that associates 10 to `a` and instead assigns to `a` the variable stored in `p`, which is 8 at this stage of the program. It then executes the loop and, at each iteration, loads the value of `p` and assigns it to `a`. As a conclusion gcc, even with -O3, fails to disambiguate pointers `p` and p_2 and does not hoist or sink the instruction `a = *p` outside the

[8] Available at http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/.

```
1  assert(N>1);
2  int* p = malloc (2*N*sizeof(int));
3  int *p1, *p2, a;
4  *p = 8;
5  a = 10;
6  p1 = p + N;
7  p2 = p + 2 * N - 1;
8  while(p2>p1){
9        a = *p;
10       *p2 = 4;
11       p2 --;
12 }
```

Figure I.1.10a – Example candidate for loop invariant code motion.

loop. Optimization level O3 relies on the default alias analysis, which is type-based, that only marks all variables of compatible types as "may alias".

```
1  .file   "code_motion.c"
2
3       ...                             # %edx -> a; %rcx -> p2, %rax -> p
4       movl    $8, %edx                # a = 8
5       movl    $8, (%rax)              # *p = 8
6       jmp .L3
7  .L6:
8       movl (%rax), %edx    # a = *p
9  .L3:
10      movl    $4, (%rcx)              # *p2 = 4
11      subq    $4, %rcx                # p2 - -
12      cmpq    %rsi, %rcx              # p2 >? p1
13      jne .L6                         # if p2 <= p1 jump to .L6
14      ...
```

Figure I.2.13 – Simplified assembly code given by gcc – O3 applied to program of Figure I.1.10a. The boxed instruction stays inside the L6 loop. We use comments to recall program instructions.

I.2.5.1.2 LLVM

We now try to optimize the loop in the program of Figure I.1.10a using "*opt*" to activate LLVM optimizer (version 3.8). A simplified Intermediate Representation (IR) of optimized bytecode is depicted in Figure I.2.14. Similarly to GCC, LLVM which mainly relies on the default optimization *basicaa* (for basic alias analysis) fails to disambiguate pointers p and p_2 and the *Loop Invariant Code Motion* (LICM) pass fails to remove the load from p_2 from the while loop.

I.2.5.1.3 ICC

We have tested the code in Figure I.1.10a using ICC 15 released in 2014. Simplified assembly code generated with O3 optimizations is depicted in Figure I.2.15. In-box code shows the load from p which is kept inside the while loop. We conclude that the Intel compiler is also unable to disambiguate pointers p and p_2 or to prove some other conditions needed to hoist the load.

```
1  while.body.lr.ph:                                              ; preds = %entry
2    %add.ptr5 = getelementptr inbounds i32* %1, i64 %conv
3    br label %while.body
4
5  while.body:
6    %2 = phi i32 [ 8, %while.body.lr.ph ], [ %3, %while.body ]
7    %p2.010 = phi i32* [ %add.ptr5, %while.body.lr.ph ], [ %incdec.ptr, %while.body ]
8    store i32 4, i32* %p2.010, align 4                             ; *p2 = 4
9    %incdec.ptr = getelementptr inbounds i32* %p2.010, i64 -1      ; p2 -
10   %cmp = icmp ugt i32* %incdec.ptr, %add.ptr                     ; p2 >?  p1
11   %3 = load i32* %1, align 4                                     ; a = *p
12   br i1 %cmp, label %while.body, label %while.end.loopexit
```

Figure I.2.14 – Simplified IR given by `clang -O3` applied to program of Figure I.1.10a. %1 denotes pointer p and %3 variable a in the `while` body. Here again, the boxed instruction is not hoisted out of the loop. Added comments may help the reader recognize program instructions.

```
1  ..B1.12:                                        # Preds ..B1.12 ..B1.11
2          lea       (,%rdi,8), %r9
3          incq      %rdi                          # p1 ++
4          negq      %r9
5          addq      %r8, %r9
6          cmpq      %rbx, %rdi
7          movl      %edx, (%r9)                   # *p2 = 4
8          movl      (%rax), %esi                  # a = *p
9          movl      %edx, -4(%r9)                 # *(p2 - 1) = 4
10         jb        ..B1.12                       # if p1 < p2 jump ..B1.12
```

Figure I.2.15 – Simplified assembly code given by icc – O3 applied to the program of Figure I.1.10a. The loop has been unrolled by a factor of two. Another loop is also generated by ICC where the load is removed but replaced by a = 8. The flow gets into this loop as soon as $p > p_2$ (which implies that p and p_2 will never alias anymore, since p_2 is decreasing).

To fix the lack of precision of alias analyses inside compilers and enable more code optimizations, the programmer may force-add *"non alias"* information for some pointer variables using the *restrict* keyword.

I.2.5.2 The restrict keyword

The *restrict* keyword is a C99 standard keyword [ISO99, §6.7.3.1 p110-112] [ISO11, §6.7.3.1 p123-125] that can be specified by the programmer to give the compiler information about aliasing. In fact, it is applied to a pointer p to say that only p or a pointer derived from it can access that memory region during its lifetime. Hence, if p is a restricted pointer then, any access to p's block must occur through p while p is alive. Otherwise, we have an undefined behavior. In Figure I.2.16, p1 is allowed to access p0's block since it is assigned a value based on p0 while it is not the case for p2. p2 = p1 is then an undefined behavior. Note that the compiler *trusts* the developer regarding the provided "no alias" information. In other words, if p and q are annotated "*restrict*" then the compiler does not check if they really do alias or not and supposes they do not.

```
1 int* restrict p0 = malloc(3*sizeof(int));
2 int* restrict p1 = p0 + 1; // OK
3 int* p2;
4 p2 = p1; // Undefined behavior
```

Figure I.2.16 – Example of using the *restrict* keyword.

Provided a "no alias" information, compilers tend to perform aggressive optimizations on programs as we detailed in Section I.1.4. Now we consider the example in Figure I.2.17 to study the effect of using the *restrict* keyword inside the LLVM compiler. Let `p` and `q` be the two pointers arguments of procedure `F` in Figure I.2.17a. In the absence of alias information between `p` and `q`, the compiler is not able to perform optimizations. In fact, if `F` is not *inlined*[9], only the use of an inter-procedural or of a context-sensitive analysis could *statically* be of any help. Note that this is not true for library functions since linking is done after optimizations. Optimizing a standalone function without keeping a trace of its non optimized version would be incorrect in some other call contexts. Adding the restrict keyword to procedure `F` gives the program in Figure I.2.17b. It is a declaration that programmers assume that `p` and `q` never alias. No further alias analysis will be done by the compiler. Figure I.2.18 gives a simplified intermediate representation generated after LLVM O3 optimizations applied to `F` and `F_RES` in Figure I.2.17. In the LLVM IR representation (Figure I.2.18b), a *"no alias"* keyword has been added to the function definition because of the use of the *restrict*. The compiler here associates `%arrayidx` to pointer `p + a` and `%arrayidx2` to pointer `q + b`. The main difference between I.2.18a and I.2.18b is two extra memory accesses (a load and a store) done in I.2.18a that have been eliminated by the *global value numbering* (GVN) optimization pass. In fact, with the restrict keyword, all accesses to `p` and `q` and their related pointers are supposed not to alias. Hence, addresses `(p + a)` and `(q + b)` are disjoint and any write to `p[a]` does not affect any read from `q[b]`. The compiler can safely optimize this code by loading only once the value stored in `q[b]` and not storing the first sum `(p[a] + q[b])` to `q[a]` since we have a write-write dependence. Extra store and load are depicted respectively line 10 and line 11 in Figure I.2.18a. Without such aliasing information, the compiler is unable to perform this optimization. We further study the GVN optimizations in Chapter II.4.

```
1 void F(int* p, int* q, int a, int b){
2   p[a] = p[a] + q[b];
3   p[a] = p[a] + q[b];
4 }
```

(a) Procedure F without restrict keyword.

```
1 void F_RES(int* restrict p, int* restrict q, int a, int b){
2   p[a] = p[a] + q[b];
3   p[a] = p[a] + q[b];
4 }
```

(b) Procedure F of Figure I.2.17a with restrict keyword.

Figure I.2.17 – Example to study the *restrict* keyword inside LLVM.

Experimental evaluation of pointer analyses inside state-of-the art compilers GCC, LLVM and ICC shows that these compilers fails to carry out the optimizations based on pointer analyses for simple benchmarks. Some published techniques may had tried to improve these results and might theoretically succeed in optimizing them, but the fact that they are not added to compilers and not available in the distributed version of these compilers can be seen as exemplifying a certain lack of efficiency.

[9] Inlining a function means making a working copy of the callee within the caller.

```
1  define void @F(i32* nocapture %p, i32* nocapture readonly %q, i32 %a, i32 %b) #0 {
2  entry:
3    %idxprom = sext i32 %a to i64
4    %arrayidx = getelementptr inbounds i32, i32* %p, i64 %idxprom
5    %0 = load i32, i32* %arrayidx, align 4
6    %idxprom1 = sext i32 %b to i64
7    %arrayidx2 = getelementptr inbounds i32, i32* %q, i64 %idxprom1
8    %1 = load i32, i32* %arrayidx2, align 4
9    %add = add nsw i32 %1, %0
10   store i32 %add, i32* %arrayidx, align 4
11   %2 = load i32, i32* %arrayidx2, align 4
12   %add9 = add nsw i32 %2, %add
13   store i32 %add9, i32* %arrayidx, align 4
14   ret void
15 }
```

(a) Procedure without *restrict* keyword.

```
1  define void @F_RES(i32* noalias nocapture %p, i32* noalias nocapture readonly %q, i32 %a, i32 %b) #0 {
2  entry:
3    %idxprom = sext i32 %a to i64
4    %arrayidx = getelementptr inbounds i32, i32* %p, i64 %idxprom
5    %0 = load i32, i32* %arrayidx, align 4
6    %idxprom1 = sext i32 %b to i64
7    %arrayidx2 = getelementptr inbounds i32, i32* %q, i64 %idxprom1
8    %1 = load i32, i32* %arrayidx2, align 4
9    %factor = shl i32 %1, 1
10   %add9 = add i32 %factor, %0
11   store i32 %add9, i32* %arrayidx, align 4
12   ret void
13 }
```

(b) Procedure with *restrict* keyword.

Figure I.2.18 – Impact of using the *restrict* keyword for function F. The code inside boxes in Figure I.2.18a is unnecessary if we know that `p` and `q` do not alias.

I.2.6 Conclusion

As we saw from the simple but representative examples, mainstream compilers still struggle to distinguish intervals within the same array. Our study confirms that pointer analyses often fail to disambiguate regions addressed from a common base pointer via different offsets, as stated by Yong and Horwitz [YH04]. Field-sensitive pointer analysis could provide a partial solution to this problem. These analyses can distinguish different fields within a record, such as a `struct` in C [PKH04], or a `class` in Java [YXR11]. However, they rely on syntax that is usually absent in the low level program representations adopted by compilers. We believe that actually implementing such analyses inside production compilers is still a challenge. In Part II of this thesis, we shall propose scalable and efficient algorithms to deal with this challenge. But before, we study in the next chapter some state-of-the-art pointer analysis techniques embedded into compilers. We will start by adopting a classification of possible relations between pointers and then show the different approaches to deal with the pointer disambiguation problem.

Chapter I.3

State-of-the-Art of Pointer Analysis Techniques

Contents

Alias analyses techniques are widely used in verification and compilation domains to prove program correctness or provide optimizing program with information needed to perform optimizations. In this thesis, we are particularly interested in techniques designed and developed to be embedded into production compilers. We study in this section state-of-the-art techniques introduced to disambiguate pointers for program performance purposes. We shall give an overview of algorithms we found the most relevant and the most related to our line of research. We will focus on *static* alias analyses with a little survey of hybrid ones.

Pointers in a given program may be classified into two major classes: related and non related pointers. Related pointers are those who share a base pointer. In other words, they are both defined from the same pointer with offsets which may be the same. This is what we called *pointer arithmetic* (Section I.1.3.1). Pointers $\mathtt{p_1} = \mathtt{p_0} + \mathtt{x}$ and $\mathtt{p_2} = \mathtt{p_0} + \mathtt{y}$, $\mathtt{x}$ and $\mathtt{y}$ integers, are related by pointer $\mathtt{p_0}$. Pointers which are not related are said non related pointers or pointers with different base pointers. In this section, we shall adopt this classification to study state-of-the-art alias analyses algorithms and comment on their characteristics.

I.3.1 Different Base Pointers

Andersen's (Section I.2.2.2.1) and Steensgaard's (Section I.2.2.2.2) algorithms are the best known algorithms for the inclusion and unification based approaches. The output of such analysis style is a *points-to* set for each program pointer. Both algorithms are flow- and context-insensitive. Andersen's analysis has a cubic worst-case running time while Steensgaard's one is almost linear. In the contrary, the former may be much more precise than the latter. Recall that the cubic complexity of Andersen's algorithm is due to computing the transitive closure of the constraint graph where nodes represent points-to sets and an edge from `a` to `b` means the points-to set of `a` is a subset of the points-to set of `b`. Many techniques have been proposed in the literature to improve Andesen's analysis. In [Sta09] Staiger attempted to improve the precision by implementing a sparse flow-sensitive version run on programs in SSA representation. A context-sensitive approach for java programs has also been proposed in [LPH05]. Shapiro and Horwitz proposed in [SH97] a modification for Andersen's and Steensgaard's algorithms to increase the precision of Steensgard's analysis while keeping an almost linear runtime. The key insight of this work is to limit the number of outgoing edges of each node in the constraint graph to a degree `k` ranging from 1 to `n` with `n` the number of nodes in the graph (number of program variables). If $\mathtt{k} = \mathtt{1}$, we have the Steensgaard's analysis, if $\mathtt{k} = \mathtt{n}$, it is Andesen's style. Based on fixed `k`'s value, variables are assigned to the different `k` categories. Nodes within the same category may be collapsed à la Steensgaard. The main contribution of [SH97] is thus a *tuned* algorithm with a complexity and accuracy that range from those of Steensgard to those of Andersen. Some other works have been proposed to keep Andersen's analysis precision while speeding up the algorithms used to solve constraints. These techniques are mainly based on the idea of collapsing nodes in the constraint graph when cycles are detected. This is because all nodes in cycles are guaranteed to have the same points-to set. In [PB09], Pereira and Berlin introduced two techniques called *Wave Propagation* and *Deep Propagation* to deal with points-to set propagation and new edges related to complex constraints inserted in the graph. Although their cubic time on the number of nodes, depending on the running environment regarding memory availability and how big benchmarks are, wave and deep propagation methods may show in practice better runtime performance comparing to state-of-the-art resolution algorithms.

Despite all the work done to make Andesens's and Steensgaard's algorithms more precise with better scalability, these techniques are still limited to disambiguating pointers with different base pointers. We recall that for both algorithms, statements that generate constraints are the following: $\mathtt{a} = \&\mathtt{b}$, $\mathtt{a} = \mathtt{b}$, $\mathtt{a} = {}^*\mathtt{b}$, and ${}^*\mathtt{a} = \mathtt{b}$. This means that only non related pointers may

be disambiguated when running alias queries on points-to sets. Arrays are treated as single big objects. Hence, a pointer arithmetic instruction $\mathtt{q} = \mathtt{p} + \mathtt{1}$ is treated as $\mathtt{q} = \mathtt{p}$.

In [Alv+15], Alves *et al.* introduce a dynamic and a static with runtime check approaches to disambiguate pointers in order to enable optimizations. Recall that a hybrid alias analysis builds guards checked at runtime to determine whether two memory locations can overlap (Section I.2.1.2) while a dynamic analysis gathers pointer values at runtime. In the purely dynamic technique, they are able to diambiguate pointers with different allocation sites using a *red-black tree* `T` [Bay72]. Each pointer allocation is a new insert associated with the allocated range. Bounds of these ranges are integers determined at runtime and are used for debug purposes. Given a pointer `p`, $\mathtt{T}(\mathtt{p})$ returns a *unique identifier* of the allocated memory block pointed by `p`. Two pointers `p` and `q` do not alias if $\mathtt{T}(\mathtt{p}) \neq \mathtt{T}(\mathtt{q})$. However, if $\mathtt{q} = \mathtt{p} + \mathtt{c}; \mathtt{c} \in \mathbb{N}$, the analysis assumes that $\mathtt{T}(\mathtt{p}) = \mathtt{T}(\mathtt{q})$ and `p` and `q` may alias. The hybrid analysis shall be discussed in Section I.3.2.1.1.

I.3.2 Solving Pointer Arithmetic

I.3.2.1 Range-based alias analyses

Range-based alias analyses associate ranges with pointers to disambiguate memory locations. These techniques share a common idea: two memory addresses $\mathtt{p}_1 + [l_1, u_1]$ and $\mathtt{p}_2 + [l_2, u_2]$ do not alias if the intervals $[\mathtt{p}_1 + l_1, \mathtt{p}_1 + u_1]$ and $[\mathtt{p}_2 + l_2, \mathtt{p}_2 + u_2]$ do not overlap. These analyses differ in the way they represent intervals: with numeric or symbolic bounds, and in the way they solve them.

I.3.2.1.1 Numeric interval bounds

Integer intervals have being used by Yong *et al.* [YH04] to detect potential out-of-bounds array accesses. In [YH04], authors handle pointer arithmetic for both well typed and mismatched typed arithmetic. Each pointer is abstracted as $\langle \mathtt{location}, \mathtt{range} \rangle$ in the location-offset representation and as $\langle \mathtt{location} : \mathtt{type}, \mathtt{range} \rangle$ in the description-offset representation that handles types; `type` is the type of elements in `range`. Actually, C standard allows casts between pointers of different types (undefined behavior if the resulting pointer is not correctly aligned [ISO11, §6.3.2p7]) while it supposes that pointers of different types do not alias. Hence, in the context of pointer analysis for program verification, and when the goal is to check array out-of-bound accesses, description-offset representation is more precise. This is in case the analyzed programs use mismatched typed arithmetic.

The technique of coupling memory locations with numeric intervals to track memory accesses was also proposed by Balakrishnan and Reps in [BR04] where they introduced the notion of *Value Set* Analysis. In fact, they define a set of *abstract locations* called a-locs which can be the whole or just one segment of scalars, structs, arrays, etc. Locations a-locs are used to track an overapproximation of memory location's values accessed through these locs. They roughly correspond to global and local variables in the program. Adresses that belong to an a-loc are denoted by `rng`, $[\mathtt{offset}, \mathtt{offset} + \mathtt{size} - \mathtt{1}]$, and correspond to offsets they may access within their definition region. Note that used offsets are integers. In Chapter II.1 however, we shall be inspired by this work to design a new alias analysis technique which is more precise.

Integer intervals have also being more recently used by Oh *et al.* [Oh+14]. They use pointer disambiguation incidentally, to demonstrate their ability to implement efficiently static analyses in a context-sensitive way.

Based on the line of work of Pearce *et al.* [PKH04], Balatsouras *et al.* propose in [BS16] a structure-sensitive points-to analysis developped on top of the LLVM (strongly typed IR). Abstract objects are defined regarding to variable types in both cases: known and unkown types. The analysis maps each allocated pointer to an allocation site $\hat{o}$ and each program pointer to an allocation site $\hat{o}$ plus an offset. Note that for the sake of simplicity we are representing allocation sites without related types. As we discussed in Section I.2.2.3.4, this analysis is designed to maintain a better level of precision in fields and arrays. However, only elements of *constant* array indices `c` are precisely handled as $\widehat{o[c]}$ where $\hat{o}$ is the array base pointer; other array elements are considered as unknown offsets of $\hat{o}$ and are denoted by $\widehat{o[*]}$.

Unlike Balatsouras *et al.* [BS16], Sui *et al* [Sui+16] consider non constant array indices as intervals of numeric ranges $[l, u]$. An interval does not represent consecutive accesses from an allocation site; it contains, in fact, holes represented by trip counters `t` and strides `s`. Designed for loop alias analysis, the trip counter stores the number of loop iterations. The stride `s` is computed based on the loop access step and the size of the accessed element. Both `t` and `s` are then used to compute the *set* of memory accesses from the related memory object. If two memory locations are generated from the same memory object (same allocation site) and the memory accesses sets overlap, then the pointers alias, otherwise they are disjoint.

I.3.2.1.2 Symbolic interval bounds

In a symbolic range analysis, ranges are defined as expressions of the program symbols, a symbol being either a constant or the name of a variable. It represents any name in the program syntax that cannot be built as an expression of other names.

Integer linear programming Much of the work on automatic parallelization has some way to associate symbolic offsets, usually loop bounds, with pointers. Michael Wolfe [Wol96, Ch.7] and Aho et al. [Aho+06, Ch.11] have entire chapters devoted to this issue. State-of-the-art approaches resort to integer linear programming (ILP) or the Greatest Common Divisor test to solve diophantine equations.

Rugina and Rinard [RR05] developed a symbolic range analysis to solve pointer arithmetic in programs with recursive calls and intensive dynamic memory allocations like divide and conquer programs. For each procedure, their analysis is able to determine a contiguous set of symbolic memory regions a procedure reads or writes within an allocation block. For instance, given `a` an allocation block, the analysis result for a procedure `f` is an interval $[l, u]$ at each program point such that each call to `f` reads or writes regions between l and u within `a`. Bounds l and u are symbolic polynomials with rational coeficients. The constraint-based algorithm aims to maximize the lower bound and minimize the upper bound of each variable `v`. To that end, `v` is expressed as a linear combination of program symbols (`f`'s parameters). The constraint system is finally solved by reducing it to a linear program.

Abstract interpretation Like [YH04], [Alv+15] represent intervals contiguously yet in hybrid approaches. In their hybrid approach, Alves *et al.* [Alv+15] analyze pairs of pointers which have not been disambiguated by an off-the-shelf static analysis and need extra runtime information. Checks are generated statically for each Single Entry Single Exit block (SESE) and used at runtime to disambiguate pointers. Given a base pointer `p` used to access memory for a load or a store within a loop (or nested loops), the analysis computes the region `M` covered by all accesses. In other words, an `M` for each offset $\texttt{v}_\texttt{i}$ accessed from `p`. To that end, they consider the memory accessed size $\texttt{s}_\texttt{i}$ which gives the concrete access when combined with the lower and upper

bounds of v_i, and with p. For instance, given a pair of pointers p_1 and p_2 with estimated regions dereferenced M_1 and M_2, pointers do not overlap if $p_1 + M_1 \leq p_2$ or $p_2 + M_2 \leq p_1$ which is checked at runtime. To compute the range (lower and upper bounds) of offsets v_i, two different techniques are used: a *polyhedral based range analysis* and a *symbolic based range analysis*. The polyhedral based approach creates relations between integer variables and linear constraints. It tries for instance to generate the relation $N \leq v_i \leq N + k$ such that N and k are program symbols. The symbolic based range analysis builds range values by mapping each v_i with its estimated lower and upper symbolic bounds. Since the presented analysis is hybrid, it introduces an execution time overhead related to runtime check. However, it allows at the same time more precise pointer disambiguation which gives the compiler the opportunity to perform more optimizations. In [Alv+15], Alves *et al.* compare program runtimes when compiled with LLVM highest level of optimization (O3) to those when compiled with LLVM O3 augmented with their pass. Although the overall speedup shown, this technique may face, for certain programs, scalability and space problems due to important overheads introduced and the code size expansion related to inserted checks and duplicated code.

I.3.2.2 Relational pointer analyses

In general, a relational analysis associates a single or multiple program variables with an abstract information or sets of other program variables. In the field of relational analyses we distinguish two types: the *semi-relational* and the *fully-relational*. Semi-relational analyses associate single program variables with sets of other variables, fully-relational ones associate tuples of variables with abstract information. In pointer analysis line of research, state-of-the-art relational analyses show a limited use. In [BGS00], Bodik *et al.* proposed a pointer semi-relational analysis to eliminate array-bound-checks in *just-in-time* java compilers. Their algorithm called ABCD (short for Array Bounds Checks on Demand) builds a new program representation to achieve a sparse *less-than* analysis. To that end, they build an *inequality graph* where nodes are program variables (including pointers) and weighted edges represent the difference between variable nodes whenever this weight is known statically. As we introduced, the ABCD algorithm was initially designed to eliminate array-bound-checks which is asserted by a negative length between an offset x and the length of the accessed array A.length in the shorted path between both variables. However, and as pointed by authors, this technique correctly handles pointer aliasing and can be used to disambiguate pointers. In Chapter II.2 we shall introduce a new but alike algorithm for alias analysis and discuss the differences between both algorithms in terms of relational analysis resolving.

Fully-relational abstract interpretation-based analyses, such as Octogons [Min06] or Polyhedrons [CH78], associate tuples of variables with abstract information. Miné's Octogons [Min06] build relations such as $x_1 + x_2 \leq 1$, where x_1 and x_2 are variables in the target program. Cousot and Halbwachs' Polyhedrons build more general relations between program variables such as $x_1 + 2x_2 \leq N$ where N is a constant in the program. All these techniques can be used to discover relationships between integer variables as well as pointers. If the projection of the given abstract value on the pointers p_1 and p_2 is empty, then we can conclude that they do not alias. This technique is very expressive but clearly too costly for the programs we want to analyze.

Another example of relational analysis is the preprocessing phase of the polyhedral model-based optimizations [FL11]. The polly-LLVM[1] implementation computes a representation of the loop iterations as a polyhedron on the iteration variables. In some cases the analysis is capable of disambiguating pointers that differ inside the loops it analyzes. However, the framework is not capable of propagating information from tests, which is a severe limitation if we want to use it

[1] Available at http://polly.llvm.org/

for our disambiguation purpose.

I.3.3 Conclusion

In this chapter, we gave a survey of state-of-the-art pointer analysis techniques used to disambiguate pointers inside compilers. We saw that for both pointer classes we considered, although alias analysis is an old problem, it still draws the attention of researchers. This is mainly due to the lack of precision or scalability noticed by some, or also the aim to make it useful for optimizations for others. In the next part of this thesis, we shall be interested in pointer arithmetic. We develop and implement new alias analysis techniques to join precision and efficiency. Putting pointer analysis into work for program optimizations shall be discussed in Chapter II.4.

Part II

Static Alias Analyses for Pointer Arithmetic

Chapter II.1

Symbolic Range Analysis of Pointers

Contents

In this chapter, we present a new alias analysis technique to deal with pointer arithmetic in C-like languages. This analysis is based on an off-the-shelf symbolic range analysis as we shall discuss in the sequel. We start the chapter by motivating the need to such pointer analysis through a pattern found in distributed systems. We give after an overview on how our algorithm works to disambiguate pointers. In the rest of the chapter, we deeply detail the design and execution of our analysis implemented on top of the LLVM compiler. We show and discuss the experiments we have run to evaluate the complexity and precision of our algorithm.

II.1.1 Context and Motivation

Figure II.1.1 shows a pattern typically found in distributed systems implemented in C. Messages are represented as arrays of bytes. In this particular example, messages have two parts: an identifier, which is stored in the beginning of the array, and a payload, which is stored right after. The loops in lines 3-6 and 7-10 fill up each of these parts with data. If a compiler can prove that the stores at lines 4 and 8 are always independent, then it can perform optimizations that would not be otherwise possible. For instance, it can parallelize the loops, or switch them, or merge them into a single body.

```
1  void prepare(char* p, int N, char* m) {
2    char *i, *e, *f;
3    for (i = p, e = p + N; i < e; i += 2) {
4      *i = 0;
5      *(i + 1) = 0xFF;
6    }
7    for (f = e + strlen(m); i < f; i++) {
8      *i = *m;
9      m++;
10   }
11 }
12
13 int main(int argc, char** argv) {
14   int Z = atoi(argv[1]);
15   char* b = (char*)malloc(Z + strlen(argv[2]
16   char* s = (char*)malloc(strlen(argv[2]));
17   strcpy(s, argv[2]);
18   prepare(b, Z, s);
19   //...
20 return 0;
21 }
```

Figure II.1.1 – Example of program that builds messages as sequences of serialized bytes. We are interested in disambiguating the locations accessed at lines 4 and 8.

No alias analysis currently available in distributed version of either gcc or LLVM is able to disambiguate the stores at lines 4 and 8. These analyses are limited because they do not contain *range information*. The range interval $[l, u]$ associated with a variable `i` is an estimate of the lowest (l) and highest (u) values that `i` can assume throughout the execution of the program.

Note that in this case, pointers at lines 4 and 8 are both defined from pointer `p` and numeric range information (with integer bounds) do not help to disambiguate them. In fact, variable `i` in the loop line 4 for instance grows up until $\texttt{p} + \texttt{N} - 1$ while `N` is a function parameter whose value still unknown before runtime, even when using a context-sensitive analysis. Therefore, the state-of-the-art techniques presented in Section I.3.1 and Section I.3.2.1.1 are not able to prove the disjointness of accesses.

In this chapter, we propose an alias analysis, based on symbolic intervals, that solves this problem. We call this analysis *Global Range Analysis of pointers*.

Figure II.1.2 shows a program in which the simple intersection of ranges using the global range analysis would not let us disambiguate pointers $\texttt{tmp}_0$ and $\texttt{tmp}_1$. We shall go back to this example later to explain how the global range analysis technique is not able to disambiguate these pointers. Notwithstanding this, we know that $\texttt{tmp}_0$ and $\texttt{tmp}_1$ will never point to a common location.

In fact, these pointers constitute different offsets from the same base address. To deal with this imprecision of the global check, we will also be discussing a *local disambiguation criterion*.

```
1 void accelerate(float* p, float X, float Y, int N) {
2   int i = 0;
3   while (i < N) {
4     p[i] += X; // float* tmp 0 = p + i; *tmp 0 = ...;
5     p[i + 1] += Y; // float* tmp 1 = p + i + 1;
6     i += 2; // *tmp 1 = ...;
7
8   }
9 }
```

Figure II.1.2 – Program that shows the need to assign common names to addresses that spring from the same base pointer.

II.1.2 Overview

We introduce a new alias analysis that defines an abstract domain that associates pointers with symbolic ranges. In other words, for each pointer `p` we conservatively estimate the range of memory slots that can be addressed as an offset of `p`. We let GR(`p`) be the global abstract address set associated with pointer `p`, such that if $\mathsf{loc}_i + [l, u] \in \text{GR}(\mathtt{p})$, then `p` may dereference any address from $@(\mathsf{loc}_i) + l$ to $@(\mathsf{loc}_i) + u$, where loc_i is a program site that contains a memory allocation call, and $@(\mathsf{loc}_i)$ is the actual return address of the `malloc` at runtime. We let $\{l, u\}$ be two *symbols* defined within the program code. Like the vast majority of pointer analyses available in the compiler literature, from Andersen's work [And94] to the more recent technique of Zhang *et al.* [Zha+14], our method is "correct" if the underlying program is also "correct". In other words, our results are sound with respect to the semantics of the program if this program has no undefined behavior, such as out-of-bounds accesses.

The key insight of this technique is the combination of pointer analysis with range analysis on the symbolic interval lattice. The state-ot-the-art of symbolic range analysis has been discussed in Section I.3.2.1.2. We here recall that, in such analysis, ranges are defined as expressions of the program symbols, a symbol (formalized later in Section II.1.3.3) being either a constant or the name of a variable. The algorithms that we present in the rest of this chapter do not depend on any particular implementation. Nevertheless, the more precise the range analysis that we use, the more precise the analysis results that we produce. In this work we have adopted the symbolic range analysis proposed in 1994 by William Blume and Rudolf Eigenmann [BE94].

To validate our ideas, we have implemented them in the LLVM compilation infra-structure [LA04] (Section I.2.4). We have tested our pointer analysis onto three different benchmarks used in previous work related to pointer disambiguation: Prolangs [Ryd+01], PtrDist [ZRW05] and MallocBench [GZH93]. As we will show in Section II.1.6.2, our analysis is linear on the size of programs. It can go over one-million assembly instructions in approximately 10 seconds. Furthermore, we can disambiguate 1.35x more queries than the alias analysis currently available in LLVM.

As was introduced in Section II.1.1, we have two different ways to answer the following question: "do pointers $\mathtt{tmp}_\mathtt{i}$ and $\mathtt{tmp}_\mathtt{j}$ alias?" These tests are called *global* and *local*; two distinct but complementary strategies: one is not a superset of the other. In the following, we go back to our running examples in Figure II.1.1 and Figure II.1.2 to illustrate situations in which each query is more effective.

II.1.2.1 Global pointer disambiguation

In Figure II.1.1 our goal is to disambiguate memory accesses at lines 4 and 8. To achieve this goal, we couple alias analysis with range analysis on symbolic intervals [BE94]. Thus, we will say that the store at line 4 might modify any address from $\mathtt{p}+\mathtt{0}$ to $\mathtt{p}+\mathtt{N}-\mathtt{1}$, and that the store at line 8 might write on any address from $\mathtt{p}+\mathtt{N}$ to $\mathtt{p}+\mathtt{N}+\mathtt{strlen(m)}-\mathtt{1}$. For this purpose, we will use an *abstract address* that encodes the actual value(s) of `p` inside the `prepare` function. These memory addresses are depicted in Figure II.1.3, where each □ represents a memory slot.

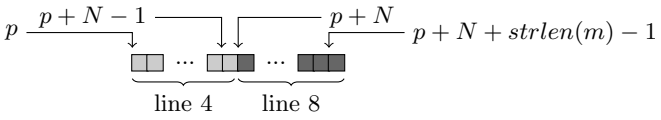


Figure II.1.3 – Array `p` in the routine `prepare` seen in Figure II.1.1. Lines 4 and 8 represent the different stores in the figure.

Whole program analysis reveals that there are two candidate locations that any pointer in the program may refer to. These locations have been created at lines 15 and 16 of Figure II.1.1, and we represent them abstractly as loc_{15} and loc_{16}. These names are unique across the entire program. After running our analysis, we find out that the abstract state (GR) of `i` at line 4 is $\mathrm{GR}(\mathtt{i}_{\ell_{\mathtt{n}.4}}) = \{\mathsf{loc}_{15} + [\mathtt{0}, \mathtt{N}-\mathtt{1}]\}$, and that the abstract state of `i` at line 8 is $\mathrm{GR}(\mathtt{i}_{\ell_{\mathtt{n}.8}}) = \{\mathsf{loc}_{15} + [\mathtt{N}, \mathtt{N}+\mathtt{strlen(m)}-\mathtt{1}]\}$. Given that these two abstract ranges do not intersect, we know that the two stores always update different locations. We call this check the *global disambiguation criterion*.

II.1.2.2 Local pointer disambiguation

After performing the global range analysis for program in Figure II.1.2, we have that $\mathrm{GR}(\mathtt{tmp}_0) = \{\mathsf{loc}_0 + [\mathtt{0}, \mathtt{N}+\mathtt{1}]\}$ and that $\mathrm{GR}(\mathtt{tmp}_1) = \{\mathsf{loc}_0 + [\mathtt{1}, \mathtt{N}+\mathtt{2}]\}$, where loc_0 defines the abstract address of the function parameter `p`. The intersection of these ranges is non-empty for $\mathtt{N} \geq \mathtt{1}$. Thus, the global check that we have used to disambiguate locations in Figure II.1.1 does not work in Figure II.1.2. In this case, we rename every pointer `p` that is alive at the beginning of a single entry region to a fresh name $\mathtt{new}_\mathtt{p}$. A region being a block in the control flow graph. Whereas we use the global test for pointers in different regions, the local test is applied onto pointers within the same single entry region. After renaming, we update the table of pointer pairs, so that $\mathrm{LR}(\mathtt{new}_\mathtt{p}) = \mathsf{loc}_{new} + [\mathtt{0}, \mathtt{0}]$, regardless of the old ranges assigned to the original pointer p. In Figure II.1.4 we would have that $\mathrm{LR}(\mathtt{tmp}_2) = \mathsf{loc}_{new} + [\mathtt{0}, \mathtt{0}]$ and $\mathrm{LR}(\mathtt{tmp}_3) = \mathsf{loc}_{new} + [\mathtt{1}, \mathtt{1}]$, where $\mathtt{tmp}_2$ is the name of the address $\mathtt{new}_{\mathtt{p}[\mathtt{0}]}$, and $\mathtt{tmp}_3$ is the name of the address $\mathtt{new}_{\mathtt{p}[\mathtt{1}]}$. This new binding of intervals to pointers gives us empty intersections between similar locations in $\mathrm{LR}(\mathtt{tmp}_2)$ and $\mathrm{LR}(\mathtt{tmp}_3)$. Consequently, the local check is able to distinguish addresses referenced by $\mathtt{tmp}_2$ and $\mathtt{tmp}_3$.

II.1.3 Combining Range and Pointer Analyses

We perform our pointer analysis in several steps. Figure II.1.5 shows how each of these phases relates to others. Our final product is a function that, given two pointers, $\mathtt{p}_0$ and $\mathtt{p}_1$, tells if they may point to overlapping areas or not. An invocation of this function is called a *query*. We use an off-the-shelf symbolic range analysis, e.g., à la Blume [BE94], to bootstrap our pointer analysis. By inferring the symbolic ranges of pointers, we have two alias tests: the global and the local approach. In the rest of this section we describe each one of these contributions.

```
1 void accelerate(float* p, float X, float Y, int N) {
2   int i = 0;
3   while (i < N) {
4     float* new_p = p+i; // LR(new_p ) = loc_new + [0, 0]
5     new_p [0] += X; // float* tmp2 = new_p ; *tmp2 = ...;
6     new_p [1] += Y; // float* tmp3 = new_p + 1; *tmp3 = ...;
7     i += 2;
8   }
9 }
```

Figure II.1.4 – Program from Figure II.1.2, after the pointer is renamed within the loop.

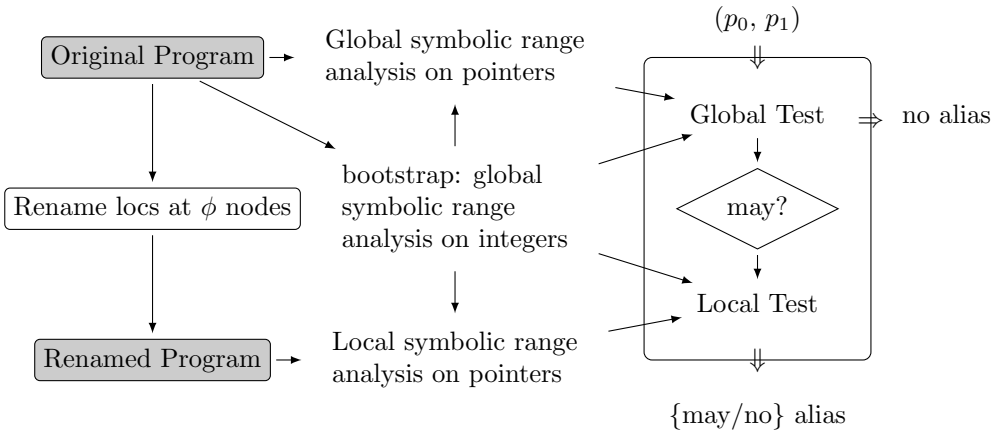


Figure II.1.5 – Overview of our pointer analysis.

II.1.3.1 A core language

We solve range analysis through abstract interpretation. To explain how we abstract each instruction in our intermediate representation, we shall use the language seen in Figure II.1.6; henceforth, we shall call this syntax our *core language*. We shall be working on programs in extended Static Single Assignment (e-SSA) form. The reader may refer to Section I.1.2.1 where we discussed the e-SSA form. We here recall that it is a flavor of Static Single Assignment (SSA) form, with variable renaming after inequalities. Thus, our core language contains ϕ-functions to ensure the single definition (SSA) property, and intersections to rename variables after conditionals. We refer the reader to Section I.1.2 for a detailed description of the SSA form and of its extensions. We here assume that ϕ-functions have only two arguments. Generalizing this notation to `n`-ary functions is immediate.

Figure II.1.7 shows the control flow graph of the program seen in Figure II.1.1. The implementation of the analysis that we shall present in this chapter is interprocedural, albeit not context-sensitive (Section I.2.2). To achieve interprocedurality, we associate actual parameters with formal parameters of functions. In the example of Figure II.1.1, pointer `b` – an actual parameter – is linked with `p` – a formal parameter – through a ϕ-function.

The e-SSA format lets us implement our analysis sparsely, e.g., we can assign information directly to variables, instead of having to assign it to pairs of a variable and a program point (Section I.2.2.3.2). As demonstrated by Choi *et al.* [CCF91], the main advantage of a sparse analysis is efficiency: the product of the analysis - the information that is bound to each variable - requires $O(N)$ space, where `N` is the number of variable names in the program. Furthermore, as we shall explain in the rest of this section, our analysis can be computed in $O(N)$ time.

Integer constants	::=	$\{c_1, c_2, \ldots\}$
Integer variables	::=	$\{i_1, i_2, \ldots\}$
Pointer variables	::=	$\{p_1, p_2, \ldots\}$
Instructions (I)	::=	
– Allocate memory	|	$p_0 = \text{malloc}(i_0)$
– Free memory	|	$p_0 = \text{free}(p_1)$
– Pointer plus int	|	$p_0 = p_1 + i_0$
– Pointer plus const	|	$p_0 = p_1 + c_0$
– Bound intersection	|	$p_0 = p_1 \cap [l, u]$
– Load into pointer	|	$p_0 = *p_1$
– Store from pointer	|	$*p_0 = p_1$
– ϕ-function	|	$p_0 = \phi(p_1 : \ell_1, p_2 : \ell_2)$
– Branch if not zero	|	$\text{bnz}(v, \ell)$
– Unconditional jump	|	$\text{jump}(\ell)$

Figure II.1.6 – The syntax of our language of pointers.

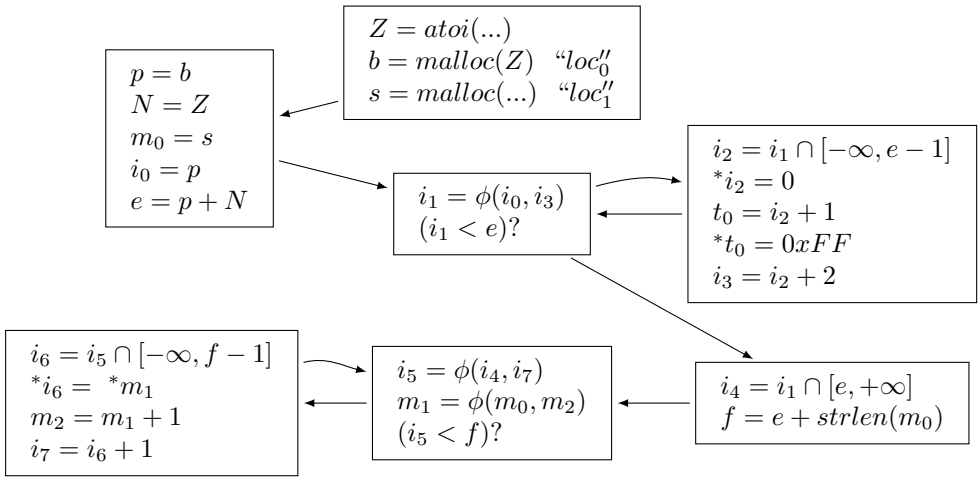


Figure II.1.7 – Control flow graph of program seen in Figure II.1.1

Key to these good properties is the fact that we create new variable names at each program point where our analysis can infer new information. This knowledge appears due to memory allocation (malloc), deallocation (free), pointer arithmetic, intersections and ϕ-functions. Each of these instructions defines new variables, whose names are associated with information. For instance, the instruction $\mathtt{p_0} = \text{free}(\mathtt{p_1})$ copies $\mathtt{p_1}$ to $\mathtt{p_0}$, and binds $\mathtt{p_0}$ to a memory chunk of size 0. As we will show in Section II.1.4, our abstract interpreter associates to $\mathtt{p_0}$ a new abstract state that indicates that $\mathtt{p_0}$ is not a valid reference to any location.

II.1.3.2 Program locations

Our analysis binds variable names to sets of *locations* and *ranges*. We denote the set of locations in a program by $\mathcal{L}oc = \{\mathsf{loc}_0, \mathsf{loc}_1 \ldots, \mathsf{loc}_{n-1}\}$ where n is the number of allocation sites. In our representation, i.e., Figure II.1.6, new locations are created by *malloc* operations.

Example 5 *Figure II.1.7 shows the control flow graph of the program seen in Figure II.1.1. The two allocations at lines 15 and 16 are associated respectively with* loc_0 *and* loc_1.

II.1.3.3 Symbolic range analysis

We start our pointer analysis by running an off-the-shelf *range analysis* parameterized on *symbols*. For the sake of completeness, we shall revisit the main notions associated with range analysis, which we borrow from Nazaré *et al.* [Naz+14]. We say that E is a symbolic *expression*, if and only if, E is defined by the grammar below. In this definition, s is a symbol and $n \in \mathbb{N}$. The set of symbols s in a program forms its *symbolic kernel*. The symbolic kernel is formed by names that cannot be represented as functions of other names in the program text. Concretely, this set contains the names of global variables and variables assigned with values returned from the library functions.

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

We shall be performing arithmetic operations over the partially ordered set $S = S_E \cup \{-\infty, +\infty\}$, where S_E is the set of symbolic expressions. The partial ordering of expressions is given by $-\infty < \ldots < -2 < -1 < 0 < 1 < 2 < \ldots + \infty$. There exists no ordering between two distinct elements of the symbolic kernel of a program. For instance, $N < N+1$ but there is no relationship between an expression containing N and another expression containing M.

A *symbolic interval* is a pair $R = [l, u]$, where l and u are symbolic expressions. We denote by $R_\downarrow$ the lower bound l and $R_\uparrow$ the upper bound u. We define the partially ordered set of (symbolic) intervals $S^2 = (S \times S, \sqsubseteq)$, where the ordering operator is defined as:

$$[l_0, u_0] \sqsubseteq [l_1, u_1], \text{ if } l_1 \leq l_0 \wedge u_1 \geq u_0$$

We define the semi-lattice SymbRangesof symbolic intervals as $(S^2, \sqsubseteq, \sqcup, \emptyset, [-\infty, +\infty])$, where the join operator "$\sqcup$" is defined as:

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

Our lattice has a smallest element $\emptyset$, such that:

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

and a greatest element $[-\infty, +\infty]$, such that:

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

For sake of clarity, we also define the intersection operator "$\sqcap$":

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

$[-\infty, +\infty]$ is absorbant and $\emptyset$ is neutral for $\sqcap$.

The result of range analysis is a function $\mathtt{R} : V \mapsto S^2$, that maps each integer variable i in a program to an interval $[l, u], l \leq u$, i.e., $\mathtt{R}(i) = [l, u]$. The more precise the technique we use to produce this result, the more precise our results will be. Nevertheless, the exact implementation of the range analysis is immaterial for the formalization that follows where we use the widening operand on SymbRanges. In the following, we recall the widening for intervals defined in Section I.1.1:

$$[l, u] \nabla [l', u'] = \begin{cases} [l, u] & \text{if } l = l' \text{ and } u = u' \\ [l, +\infty] & \text{if } l = l' \text{ and } u' > u \\ [-\infty, u] & \text{if } l' < l \text{ and } u' = u \\ [-\infty, +\infty] & \text{if } l' < l \text{ and } u' > u \end{cases}$$

The only requirement that we impose on the implementation of range analysis is that it exists over SymbRanges, our lattice of symbolic intervals.
We denote by $(\alpha_{\mathsf{SymbRanges}}, \gamma_{\mathsf{SymbRanges}})$ the underlying Galois connection.

Example 6 *A range analysis such as Nazaré* et al.*'s [Naz+14], if applied onto the program seen in Figure II.1.2 (that we recall below), will give us that*

- $\mathtt{R}(i_{\ell n.2}) = [0, 0]$,
- $\mathtt{R}(i_{\ell n.4}) = [0, N-1]$,
- $\mathtt{R}(i_{\ell n.6}) = [0, N+1]$.

```
1 void accelerate(float* p, float X, float Y, int N) {
2   int i = 0;
3   while (i < N) {
4     p[i] += X; // float* tmp 0 = p + i; *tmp 0 = ...;
5     p[i + 1] += Y; // float* tmp 1 = p + i + 1;
6     i += 2; // *tmp 1 = ...;
7
8   }
9 }
```

Figure II.1.2 – Program that shows the need to assign common names to addresses that spring from the same base pointer.

II.1.4 Global Range Analysis

As we have mentioned in Section II.1.2, we use two different strategies to disambiguate pointers: the global and the local test. Our global pointer analysis goes over the entire code of the program, associating pointer variables with elements of an abstract domain that we will define soon. The local analysis, on the other hand, works only for small regions of the program text. We shall discuss the local test in Section II.1.5. In this section, we focus on the global test, which is an abstract-interpretation based algorithm.

II.1.4.1 An abstract domain of pointer locations

We associate pointers with tuples of size n: $(\mathsf{SymbRanges} \cup \bot)^n$; n being the number of program sites where memory is allocated (the cardinal of $\mathcal{L}oc$) and $\cup$ is the disjoint union.

Let $@(\mathsf{loc}_i)$ denotes the actual address value returned by the i^{th} malloc of the program. By construction, all actual addresses are supposed to be offsets of a given $@(\mathsf{loc}_i)$. The abstract value $\mathrm{GR}(p) = (p_0, \ldots, p_{n-1})$ represents (an abstract version) of the set of memory locations that pointer variable p can address throughout the execution of a program:

Definition 6 (Abstraction) *A set of actual addresses, $S = \{s \mid \exists i \in \mathbb{N}, d \in \mathbb{N}, s = @(\mathsf{loc}_i)+d\}$ is abstracted by $\alpha(S) = (p_0, p_1 \ldots, p_{n-1})$ where :*

- $p_i = \bot$ *if there is no address in S which is an offset of $@(\mathsf{loc}_i)$*
- $p_i = \alpha_{\mathsf{SymbRanges}}(\{d \in \mathbb{Z} \mid s = @(\mathsf{loc}_i) + d \text{ and } s \in S\})$, *otherwise. The offsets from a given pointer are abstracted alltogether in the SymbRangeslattice.*

The goal of our global range analysis (GR) is to compute such an abstract value for each pointer of the program. Some elements in a tuple $\mathrm{GR}(p)$ are bound to the undefined location, i.e., $\bot$. These elements are not interesting to us, as they do not encode any useful information. Thus, to avoid keeping track of them, we rely on the concept of *support*, which we state in Definition 7.

Definition 7 (Support) *We denote by $supp_{GR}(p)$ the set of indexes for which p_i is not $\bot$:*

$$supp_{GR}(p) = \{i \mid p_i \neq \bot\}.$$

For the sake of readability, let us denote the instance $\mathrm{GR}(p) = (\bot, [l_1, u_1], \bot, [l_3, u_3], \bot)$ by the set $\mathrm{GR}(p) = \{\mathsf{loc}_1 + [l_1, u_1], \mathsf{loc}_3 + [l_3, u_3]\}$. In the concrete world, this notation will mean that pointer p can address any memory location from $@(\mathsf{loc}_1)+l_1$ to $@(\mathsf{loc}_1)+u_1$, and from $@(\mathsf{loc}_3)+l_3$ to $@(\mathsf{loc}_3) + u_3$.

For instance, consider that $l_1 = 3$, $u_1 = 5$, $l_3 = 3$ and $u_3 = 8$. $\mathrm{GR}(p) = \{\mathsf{loc}_1 + [3, 5], \mathsf{loc}_3 + [3, 8]\}$ is then depicted in Figure II.1.8.

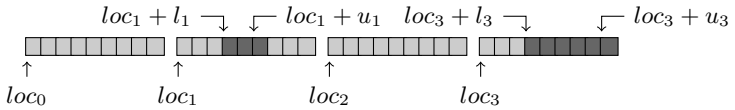


Figure II.1.8 – The concrete semantics of $\mathrm{GR}(p) = \{\mathsf{loc}_1 + [3, 5], \mathsf{loc}_3 + [3, 8]\}$. Dark grey cells denote possible (concrete) values of p.

Now for the abstract operations: $(\bot, \ldots, \bot)$ is the least element of our lattice, and $([-\infty, \infty], \ldots, [-\infty, \infty])$ the greatest one.

Given the two abstract values $\mathrm{GR}(p^1) = (p^1_0, \ldots, p^1_{n-1})$ and $\mathrm{GR}(p^2) = (p^2_0, \ldots, p^2_{n-1})$, the union $\mathrm{GR}(p^1) \sqcup \mathrm{GR}(p^2)$ is the tuple $=$

$$\begin{cases} \bot & \text{if } p^1_i = p^2_i = \bot \\ p_1 \sqcup p_2 & \text{otherwise} \end{cases}$$

and $\mathrm{GR}(p^1) \sqsubseteq \mathrm{GR}(p^2)$ if an only if all involved (symbolic) intervals of p^1 are included in the ones of p^2: $\forall i \in [0..(n-1)], p^1_i \sqsubseteq p^2_i$ (considering $\bot \sqsubseteq R$ and $\bot \sqcup R = R$ for all non-empty intervals R). We call $\mathsf{MemLocs}$ the lattice formed by $(\mathsf{SymbRanges} \uplus \bot)^n$ with its partial ordering.

Example 7 *For the example depicted in Figure II.1.7 where we only have two malloc sites denoted by loc_0 and loc_1, we obtain the following results:*

- $GR(p) = GR(b) = \{\mathsf{loc}_0 + [0, 0]\}$,
- $GR(m_0) = GR(s) = \{\mathsf{loc}_1 + [0, 0]\}$,
- $GR(e) = \mathsf{loc}_0 + [N, N]$,
- $GR(m_1) = \mathsf{loc}_1 + [1, +\infty]$,
- $GR(i_7) = \mathsf{loc}_0 + [N + strlen(m_0), N + strlen(m_0) + 1]$.

We discuss in the rest of this section how to find such a mapping.

II.1.4.2 Abstract semantics for GR, and concretization

The abstract semantics of each instruction in our core language is given by Figure II.1.9. Figure II.1.9 defines a system of equations whose fixpoint gives us an approximation on the locations that each pointer may dereference. We remind the reader of our notation: $[l, u]_{\downarrow} = l$, and

$[l,u]_{\uparrow} = u$. In Figure II.1.9, this notation surfaces in the semantics of intersections created at σ-nodes. Constraints are simple. Whenever a new memory allocation at site j is performed, we associate the range [0, 0] to loc_j. We use the symbolic range analysis of integer variables for pointer arithmetic instructions and since we do not handle second order pointers (pointers to pointers), the load constraint yields the range $[-\infty,+\infty]$ for each allocation site of the loaded variable. The abstract interpretation of the pointer-related instructions in Figure II.1.7 yields the results discussed in Example 7.

$$
\begin{array}{rcl}
\begin{array}{r} j : p = \text{ malloc } (v) \\ \text{with } v \text{ scalar} \end{array} & \Rightarrow & \mathrm{GR}(p) = (\bot, \ldots, \underbrace{[0,0]}_{j^{th}component}, \ldots, \bot) \\
\begin{array}{r} p = \text{ free } (v) \\ \text{with } v \text{ scalar} \end{array} & \Rightarrow & \mathrm{GR}(p) = (\bot, \ldots, \bot) \\
v = v_1 & \Rightarrow & \mathrm{GR}(v) = \mathrm{GR}(v_1) \\
\begin{array}{r} q = p + c \\ \text{with } c \text{ numeric variable} \end{array} & \Rightarrow & \begin{cases} \mathrm{GR}(q) = (q_0, \ldots, q_{n-1}) \text{ with} \\ q_i = \begin{cases} \bot \text{ if } p_i = \bot \\ p_i + R(c) \text{ otherwise} \end{cases} \end{cases} \\
q = \phi(p^1, p^2) & \Rightarrow & \mathrm{GR}(q) = \mathrm{GR}(p^1) \sqcup \mathrm{GR}(p^2) \\
q = p^1 \cap [-\infty, p^2] & \Rightarrow & \begin{cases} \mathrm{GR}(q) = (q_0, \ldots, q_{n-1}) \text{ with} \\ q_i = \begin{cases} \bot \text{ if } (p_i^1 = \bot \text{ or } p_i^2 = \bot) \\ p_i^1 \sqcap [-\infty, p_{i\uparrow}^2] \text{ otherwise} \end{cases} \end{cases} \\
q = p^1 \cap [p^2, +\infty] & \Rightarrow & \begin{cases} \mathrm{GR}(q) = (q_0, \ldots, q_{n-1}) \text{ with} \\ q_i = \begin{cases} \bot \text{ if } (p_i^1 = \bot \text{ or } p_i^2 = \bot) \\ p_i^1 \sqcap [p_{i\downarrow}^2, +\infty] \text{ otherwise} \end{cases} \end{cases} \\
q = *p & \Rightarrow & \mathrm{GR}(q) = ([-\infty, \infty], \ldots, [-\infty, \infty]) \\
*q = p & \Rightarrow & \mathsf{Nothing}
\end{array}
$$

Figure II.1.9 – Constraint generation for GR with $\mathrm{GR}(p) = (p_0, \ldots, p_{n-1})$ given p in the right-hand side of rules

There remains to define how the abstract states will be concretized ($@(\mathsf{loc}_i)$ is the actual address returned by the i^{th} malloc):

Definition 8 (Concretization) *Let $GR(p) = (p_0, \ldots, p_{n-1})$ be an abstract value (a set of "abstract addresses for p"), we define its concretization as follows:*

$$\gamma(GR(p)) = \bigcup_{i \in supp_{GR}(p)} \{@(loc_i) + o, p_{i\downarrow} \leq o \leq p_{i\uparrow}\}$$

The concretization function of this abstract value is thus a set of (concrete) addresses, obtained by shifting a set of base addresses by a certain value in SymbRanges[1].

Proposition 1 *(α, γ) is a Galois connection.*

Proof 1 *Immediate since $(\alpha_{SymbRanges}, \gamma_{SymbRanges})$ is a Galois connection.*

[1] While speaking about symbolic ranges, we also have to concretize the values involved in the bounds of p_i, that is we shall use the actual values between $S(p_{i\downarrow})$ and $S(p_{i\uparrow})$.

II.1.4.2.1 Solving the abstract system of contraints

Following the abstract interpretation framework, we solve our system of constraints by computing for each pointer a growing set of abstract values until convergence.

However, as the underlying lattice SymbRanges has infinite height, widening is necessary to ensure that these sequences of iterations actually terminate. Example 8 illustrates a situation in which widening is necessary for the analysis to terminate.

Example 8 *Consider the program in Figure II.1.10. Variable* x *is incremented within the infinite loop. Using the widening we shall conclude that* $R(x) = [0, +\infty]$. *Otherwise, the analysis would not terminate since the upper bound of* $R(x)$ *is infinitely incremented.*

```
1 int widen_conv(int x){
2   x = 0;
3   while(true)
4       x++;
5   return x;
6 }
```

Figure II.1.10 – Program to illustrate the need for widening in the range analysis for integer variables.

Our widening operation on pointers generalizes the widening operation on ranges. It is defined as follows:

Definition 9 *Given* $GR(p)$ *and* $GR(p')$ *with* $GR(p) \sqsubseteq GR(p')$, *we define the widening operator:*

$$GR(p) \nabla GR(p') = (p_0 \nabla p'_0, \ldots, p_{n-1} \nabla p'_{n-1}),$$

where ∇ *denotes the widening on SymbRanges, extended with* $\bot \nabla \bot = \bot$ *and* $\bot \nabla [l, u] = [l, u]$.

As usual, we only apply the widening operator on a cut set of the control flow graphs (here, only on ϕ functions).

Widening may lead our interpreter to produce very imprecise results. To recover part of this imprecision, we use a descending sequence of finite size: after convergence, we redo a step of symbolic evaluation of the program, starting from the value obtained after convergence. One example of analysis will be detailed later, in Section II.1.4.4.

II.1.4.2.2 The abstract interpretation of loads and stores

In Figure II.1.9, we chose not to track precisely the intervals associated with pointers stored in memory. In other words, when interpreting stores, e.g., $q = *p$, we assign the top value of our lattice to q. This decision is pragmatic. As we shall explain in Section II.1.6.2, a typical compilation infra-structure already contains analyses that are able to track the propagation of pointer information throughout memory. Our goal is not to solve this problem. We want to deliver a fast analysis that is precise enough to handle C-style pointer arithmetic.

II.1.4.3 Answering GR queries

Our queries are based on the following result, that is an immediate consequence of the fact that our analysis is an abstract interpretation:

Proposition 2 (Correctness) *Let p and p' be two pointers in a given program then:*

if $supp_{GR}(p) \cap supp_{GR}(p') = \emptyset$ *or* $\forall i \in supp_{GR}(p) \cap supp_{GR}(p'), p_i \sqcap p'_i = \emptyset$

then $\gamma(GR(p)) \cap \gamma(GR(p')) = \emptyset$.

In other words, if the abstract values of two different pointers of the program have a null intersection, then the two *concrete pointers* do not alias. This result is directly implied by the abstract interpretation framework. Thanks to this result, we implement the query $Q_{\mathrm{GR}}(p, p')$ as:

- If GR(p) and GR(p') have an empty intersection, then "they do not alias".
- Else "they may alias".

II.1.4.4 A complete example

Example 9 shows how our analysis works on the program seen in Figure II.1.1.

Example 9 *Figure II.1.7 shows the control flow graph (cfg) of the program in Figure II.1.1. Our graph is in e-SSA form [BGS00]. Figure II.1.11 shows the result of widening ranges after one round of abstract interpretation (stabilization achieved), and a descending sequence of size two. Our system stabilizes after each instruction is visited four times. The first visit does initialization, the second widening (and stabilization check), and the last two build the descending sequence.*

This example illustrates the need of widening to ensure termination. Our program has a cycle of dependencies between pointers i_1, i_2 and i_3. If not for widening, pointer i_3, incremented in line 5 of Figure II.1.1 would grow forever. Thus, as in Abstract Interpretation, we must break the cyclic dependences between our pointers under analysis, by inserting widening points (identify points in the cfg where widening should be applied to insure convergence).

Returning to our example of Figure II.1.1, we are interested in knowing, for instance, that the memory access at line 4 is independent of the accesses that happen at line 8. To achieve this goal, we must bound the memory regions covered by pointers i_3 and i_7. A cyclic dependence happens at the operation `i++`, because in this case, we have a pointer being used as both source and destination of the update. Thus, we should have inserted a widening point at store and load instructions. However, in the Abstract Interpreter depicted in Figure II.1.9, it was sufficient to insert widening points at ϕ functions (as we have already said before) because:

- heads of loops are ϕ functions (thus dependencies between variables of different iterations of loops are broken).
- we are working on (e-)SSA form programs; thus, the only inter-loop dependencies are successive stores to the same variable : `*q=..., *q=...`. The value GR(q) is the union of all information gathered inside the loop. (In essence, memory addresses *are not* in static single assignment form, i.e., we could have the same address being used as the target of a store multiple times). This information might grow forever; hence, we would have inserted a widening point on the last write. In our case, the information we store is already the top of our lattice; hence, there is no need for widening.

	Var	GR	LR
Starting state	b, p, i_0	$([0,0], \bot)$	$\mathsf{loc}_0 + [0,0]$
	m_0, s	$(\bot, [0,0])$	$\mathsf{loc}_1 + [0,0]$
	i_1	$([0,0], \bot)$	$\mathsf{loc}_2 + [0,0]$
	i_2	$([0,0], \bot)$	$\mathsf{loc}_2 + [0,0]$
	t_0	$([1,1], \bot)$	$\mathsf{loc}_2 + [1,1]$
	e	$([N,N], \bot)$	$\mathsf{loc}_0 + [N,N]$
	i_3	$([2,2], \bot)$	$\mathsf{loc}_2 + [2,2]$
	i_4	$(\bot, \bot)$	$\mathsf{loc}_2 + [0,0]$
	f	$([k,k], \bot)$	$\mathsf{loc}_0 + [k,k]$
	m_1	$(\bot, [0,0])$	$\mathsf{loc}_3 + [0,0]$
	m_2	$(\bot, [1,1])$	$\mathsf{loc}_3 + [1,1]$
	i_5	$(\bot, \bot)$	$\mathsf{loc}_4 + [0,0]$
	i_6	$(\bot, \bot)$	$\mathsf{loc}_4 + [0,0]$
	i_7	$(\bot, \bot)$	$\mathsf{loc}_4 + [1,1]$
Growing iterations + widening	i_1	$([0,+\infty], \bot)$	
	i_2	$([0,+\infty], \bot)$	
	t_0	$([1,+\infty], \bot)$	
	i_3	$([2,+\infty], \bot)$	
	i_4	$([N,+\infty], \bot)$	
	m_1	$(\bot, [0,+\infty])$	
	m_2	$(\bot, [1,+\infty])$	
	i_5	$([N,+\infty], \bot)$	
	i_6	$([N,k-1], \bot)$	
	i_7	$([N+1,k], \bot)$	
After one descending step	i_2	$([0,N-1], \bot)$	
	t_0	$([1,N], \bot)$	
	i_3	$([2,N+1], \bot)$	
	m_1	$(\bot, [0,+\infty])$	
	m_2	$(\bot, [1,+\infty])$	
After two descending steps	i_1	$([0,N+1], \bot)$	
	i_4	$([N,N+1], \bot)$	
	i_5	$([N,k], \bot)$	
	i_6	$([N,k-1], \bot)$	
	i_7	$([N+1,k], \bot)$	

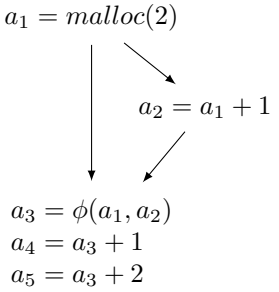
Figure II.1.11 – Abstract interpretation of the cfg seen in Figure II.1.7 (program in Figure II.1.1). For GR, we associate loc_0 with the `malloc` at line 15 and loc_1 with the `malloc` at line 16 (of the program). Only changes in GR and LR are rewritten after the growing and descending iterations. We let k = N+`strlen`(m_0).

To conclude this example, we recall that our goal is to disambiguate pointers i_2 and i_6 in the control flow graph of Figure II.1.7 used in store instructions at lines ℓ_4 and ℓ_8 in Figure II.1.1. After solving the constraints in Figure II.1.11, we get $\text{GR}(i_2) = \mathsf{loc}_0 + [0, N-1]$ and $\text{GR}(i_6) = \mathsf{loc}_0 + [N, N + strlen(m_0) - 1]$. $Q_{\text{GR}}(i_2, i_6) = \text{GR}(i_2) \cap \text{GR}(i_6) = \emptyset$ then i_2 and i_6 do not alias and memory accesses at lines ℓ_4 and ℓ_8 are disjoint (assuming that $strlen(m_0) \geq 1$).

II.1.5 Local Range Analysis

II.1.5.1 Abstract semantics for LR

The global pointer analysis is not path sensitive. As a consequence, this analysis cannot, for instance, distinguish the effects of different iterations of a loop upon the actual value of a pointer, or the effects of different branches of a conditional test on that very pointer. The program in Figure II.1.12 illustrates this issue. Pointers a_4 and a_5 clearly must not alias. Yet, their abstract states have non-empty intersections for loc_1. Therefore, the query mechanism of Section II.1.4.3 would return a "may-alias" result in this case.



$$\begin{array}{ll} \text{GR}\ (a_1) = \{\mathsf{loc}_1 + [0,0]\} & \text{LR}\ (a_1) = \mathsf{loc}_1 + [0,0] \\ \text{GR}\ (a_2) = \{\mathsf{loc}_1 + [1,1]\} & \text{LR}\ (a_2) = \mathsf{loc}_1 + [1,1] \\ \text{GR}\ (a_3) = \{\mathsf{loc}_1 + [0,1]\} & \text{LR}\ (a_3) = \mathsf{loc}_2 + [0,0] \\ \text{GR}\ (a_4) = \{\mathsf{loc}_1 + [1,2]\} & \text{LR}\ (a_4) = \mathsf{loc}_2 + [1,1] \\ \text{GR}\ (a_5) = \{\mathsf{loc}_1 + [2,3]\} & \text{LR}\ (a_5) = \mathsf{loc}_2 + [2,2] \end{array}$$

Figure II.1.12 – Example that illustrates the imprecision of the global analysis due to its lack of path-sensitiveness.

To solve this problem, we have developed a *local* version of our pointer analysis. We call it local because it creates new locations for every ϕ-function. Our local range analysis is simpler than its global counterpart. We solve it in a single iteration of abstract interpretation applied on the instructions of our core language. Instructions are evaluated abstractly in the order given by the program dominance tree. Figure II.1.13 gives the abstract semantics of each instruction. The abstract value $\text{LR}(p)$ exists in $(\mathcal{L}oc \cup \mathcal{N}ewLocs) \times \mathsf{SymbRanges}$ where $\mathcal{N}ewlocs$ denotes a set of "fresh location variables", that are computed by invocation of the function $\mathcal{N}ewLocs()$. As before, we write $\mathsf{loc} + \mathtt{R}$ instead of $(\mathsf{loc}, \mathtt{R})$. Similar to γ_{GR}, γ_{LR} denotes the set of abstract addresses from $@(\mathsf{loc}) + \mathtt{R}_{\downarrow}$ to $@(\mathsf{loc}) + \mathtt{R}_{\uparrow}$.

$$\begin{array}{rcl} p = \text{ malloc } (v) \\ \text{with } v \text{ scalar} & \Rightarrow & \text{LR}(p) = \mathcal{N}ewLocs() + [0,0] \\ v = v_1 & \Rightarrow & \text{LR}(v) = \text{LR}(v_1) \\ i : q = p + c \\ \text{with } c \text{ integer variable and} \\ \text{LR}(q) = \mathsf{loc}_i + [l,u] & \Rightarrow & \text{LR}(q) = \mathsf{loc}_i + ([l,u] + R(c)) \\ j : q = \phi(p_1, p_2) & \Rightarrow & \text{LR}(q) = \mathcal{N}ewLocs() + [0,0] \\ q = p_1 \cap [-\infty, p_2] \\ q = p_1 \cap [p_2, +\infty] & \Rightarrow & \text{LR}(q) = \text{LR}(p_1) \\ q = *p_1 & \Rightarrow & \text{LR}(q) = \mathcal{N}ewLocs() + [0,0] \\ *q = p_1 & \Rightarrow & \mathsf{Nothing} \end{array}$$

Figure II.1.13 – Constraint generation for LR.

To find a solution to the local analysis, we solve the system provided by the abstract rules seen in Figure II.1.13. This resolution process involves computing an increasing sequence of abstract values for each pointer p of the program. Contrary to the global analysis, this analysis is based on a finite lattice, we do not need any widening operator. Figure II.1.12 (Right) shows the result of the local analysis. Contrary to the global analysis, we have a new location bound to variable a_3, which is defined by a ϕ operator. The range of this new location is $[0,0]$. The other variables that are functions of a_3, e.g., a_4 and a_5, have now non-intersecting ranges associated with this new memory name.

II.1.5.2 Answering LR queries

The correction for the local analysis is stated by the following proposition, which is a direct consequence of the abstract interpretation framework usage:

Proposition 3 (Correctness) *Let p and p' be two pointers in a given program, and γ_{LR} be the concretization of the abstract map LR, which we state like in Definition 8. If $LR(p) = \mathit{loc} + \mathtt{R}$ and $LR(p') = \mathit{loc}' + \mathtt{R}'$, then if $\mathit{loc} = \mathit{loc}'$ and $\mathtt{R} \sqcap \mathtt{R}' = \emptyset$ then $\gamma(LR(p)) \cap \gamma(LR(p') = \emptyset$. In other words, p and p' never alias.*

In other words, if two pointer variables have the same local site in the abstract state such that their offset ranges do not intersect, then the concrete pointers reference non overlapping memory regions. Thanks to the concretization function, concrete ranges do not intersect if abstract ones do not.

Thanks to this result, we implement the query $Q_{\mathrm{LR}}(p, p')$ as:

- If $\mathrm{LR}(p)$ and $\mathrm{LR}(p')$ have a common base pointer with ranges that do not intersect, then "they do not alias".
- Else "they may alias".

II.1.6 Evaluation and Experiments

II.1.6.1 Complexity

The e-SSA representation ensures that we can implement our analysis sparsely. Sparsity is possible because the e-SSA form renames variables at each program point where new abstract information, e.g., ranges of integers and pointers, arises. According to Tavares *et al.* [Tav+14], this property – single information – is sufficient to enable sparse implementation of non-relational static analyses [Tav+14]. Therefore, the abstract state of each variable is invariant along the entire live range of that variable. Consequently, the space complexity of our static analysis is $O(|V| \times I)$, where V is the set of names of variables in the program in e-SSA form, and I is a measure of the size of the information that can be bound to each variable.

We apply widening after one iteration of abstract interpretation. Thus, we let the state of a variable change first from $[\bot, \bot]$ to $[s_l, s_u]$, where $s_l \neq -\infty$, and $s_u \neq +\infty$. From there, we can reach either $[-\infty, s_u]$ or $[s_l, +\infty]$. And, finally, this abstract state can jump to $[-\infty, +\infty]$. Hence, our time complexity is $O(3 \times |V|) = O(|V|)$. This observation also prevents our algorithm from generating expressions with very long chains of "min" and "max" expressions. Therefore, I, the amount of information associated with a variable, can be represented with $O(1)$ space. As a consequence of this frugality, our static analysis runs in $O(|V|)$ time, and requires $O(|V|)$ space.

II.1.6.2 Experiments

We have implemented our range analysis in the LLVM compiler, version 3.5. In this section, we show numbers that we have obtained with this implementation. All our experiments have been performed on an Intel i7-4770K, with 8GB of memory, running Ubuntu 14.04.2. Our goal with these experiments is to show: (i) that our alias analysis is more precise than other alternatives of practical runtime; and (ii) that it scales up to large programs.

On the Precision of our Analysis. In this section, we compare our analysis against the other pointer analyses that are available in LLVM 3.5, namely *basic* and *SCEV*. The first of them, although called "basic", is currently the most effective alias analysis in LLVM, and is the default choice at the -O3 optimization level. To disambiguate pointers, it relies on the heuristics we presented in Section I.2.4.2.1. As we saw, the *basic* alias analysis has some of the capabilities of the technique that we present in this chapter, namely the ability to distinguish fields and indices within aggregate types. In this case, such disambiguation is only possible when the aggregates are indexed with constants known at compilation time. For situations when these indices are symbols, LLVM relies on a second kind of analysis to perform the disambiguation: the "scalar-evolution-based" (SCEV) alias analysis. This analysis tries to infer closed-form expressions to the induction variables used in loops. For each loop such as:

$$\texttt{for}\ (\texttt{i} = \texttt{B}; \texttt{i} < \texttt{N}; \texttt{i} += \texttt{S})\{...\texttt{a}[\texttt{i}]...\}$$

this analysis associates variable `i` with the expression $\texttt{i} = \texttt{B} + \mathit{iter} \times S, \texttt{i} \leq \texttt{N}$. The parameter *iter* represents the current iteration of the loop. With this information, SCEV can track the ranges of indices which dereference array `a` within the loop. Contrary to our analysis, SCEV is only effective to disambiguate pointers accessed within loops and indexed by variables in the expected closed-form.

Program	#Queries	%scev	%basic	%rbaa	%(rbaa + basic)
cfrac	89,255	0.87	9.70	16.65	21.03
espresso	787,223	2.39	12.62	28.16	33.04
gs	608,374	15.56	40.67	56.18	59.99
allroots	974	16.32	64.37	79.77	79.88
archie	159,051	0.98	20.57	16.44	28.04
assembler	35,474	2.16	40.31	47.86	55.61
bison	114,025	0.74	10.95	9.56	14.74
cdecl	301,817	13.74	24.80	49.72	50.73
compiler	9,515	0.49	67.27	67.27	69.20
fixoutput	3,778	0.11	88.30	83.17	90.37
football	495,119	3.58	59.20	60.08	65.08
gnugo	13,519	9.23	60.89	78.21	79.29
loader	13,782	2.32	29.55	36.47	46.09
plot2fig	27,372	2.90	24.09	46.45	49.54
simulator	25,591	3.56	46.32	41.25	52.27
unix-smail	61,246	1.22	37.36	42.92	48.95
unix-tbl	85,339	7.30	44.38	33.92	48.83
anagram	3,114	2.18	32.85	53.31	59.54
bc	198,674	14.14	30.95	47.86	50.01
ft	7,660	2.73	5.23	24.65	25.91
ks	14,377	0.61	22.98	21.60	27.70
yacr2	38,262	0.20	7.22	12.83	14.48
Total	3,093,541	6.97	30.83	41.73	46.53

Figure II.1.14 – Comparison between three different alias analyses. Our analysis **rbaa** includes the global and local tests. Numbers in **scev**, **basic**, **rbaa** and **rbaa+basic** show percentage of queries that answer "no-alias".

Figure II.1.14 shows how the three different analyses fare when applied on larger benchmarks. For this experiment we have chosen three benchmarks that have been used in previous work that compares pointer analyses: Prolangs [Ryd+01], PtrDist [ZRW05] and MallocBench [GZH93]. We first notice that in general all the pointer analyses in LLVM disambiguate a relatively low

number of pointers in all programs of the three benchmarks. This happens because many pointers are passed as arguments of functions, and, not knowing if these functions will be called from outside the program, the analyses must, conservatively, assume that these parameters may alias. Second, we notice that our pointer analysis is one order of magnitude more precise than the scalar-evolution based implementation available in LLVM. Finally, we notice that we are able to disambiguate more queries than the *basic* analysis. Furthermore, our results complement it in non-trivial ways. In total, we tried to disambiguate 3.093 million pairs of pointers. Our analysis found out that 1.29 million pairs reference non-overlapping regions. The *basic* analysis has been able to distinguish 953 thousand pairs. By using our analysis in conjunction with *basic* analysis, we are able to find that 1.439 million pairs reference non-overlapping regions. In other words, our analysis disambiguates 486 thousand pairs not disambiguated by *basic* and *basic* disambiguates 149 thousand pairs that we are not able to disambiguate. All pairs disambiguated by SCEV are disambiguated by either basic or our method.

Prog	noalias	global	Prog	noalias	global
cfrac	14,865	1,102	gnugo	10,573	1,851
espresso	221,416	20,791	loader	5,026	433
gs	341,532	106,859	plot2fig	12,713	861
allroots	777	182	simulator	10,557	1,092
archie	26,142	2,034	unix-smail	26,289	771
assembler	16,977	905	unix-tbl	28,948	1,136
mybison	10,905	1,417	anagram	1,660	88
cdecl	150,050	43,619	bc	95,091	32,498
compiler	6,401	156	ft	1,888	452
fixoutput	3,142	4	ks	3,105	218
football	297,491	22,052	yacr2	4,909	487

Figure II.1.15 – Number of queries solved with the global test of Section II.1.4. Column **noalias** gives the number of queries that we have been able to disambiguate, and column **global** shows how many queries were solved with the global test.

Figure II.1.15 shows the proportion of queries that we have been able to disambiguate with the global test of Section II.1.4. The two columns **noalias** of Figure II.1.15 correspond to the percentage in column **%rbaa** applied on the column #**Queries** of Figure II.1.14. Overall, the global test has given us 239,008, out of 1,290,457 "no-alias" answers. This corresponds to 18.52% of all the pairs of pointers that we have disambiguated. We did not show the local test in this table because these two tests are not directly comparable. The global test disambiguates pointers in the whole program, and the local test disambiguates pointers defined in the Single Entry Single Exit Blocks. Therefore, run separately, these two tests may disambiguate the same pair of pointers: globally and locally different. For instance, when ϕ-operands have different allocation sites, the ϕ-function join does not make the global analysis lose precision. The two experiments of Figure II.1.14 and Figure II.1.15 show that even though our local and global analyses overlap in some few cases, they are both important and complementary.

On the Scalability of our Analysis. The chart in Figure II.1.16 shows how our analysis scales when applied on programs of different sizes. We have used the 50 largest programs in the LLVM benchmark suite. These programs gave us a total of 800,720 instructions in the LLVM intermediate representation, and a total of 241,658 different pointer variables. We analyzed all these 50 programs in 8.36 seconds. We can – effectively – analyze 100,000 instructions in about one second. In this case, we are counting only the time to map variables to values in SymbRanges. We do not count the time to query each pair of pointers, because usually compiler optimizations perform these queries selectively, for instance, only for pairs of pointers within a loop. Also, we

do not count the time to run the out-of-the-box implementation of range analysis mentioned in Section II.1.3.3, because our version of it is not implemented within LLVM. It runs only once, and we query it afterwards, never having to re-execute it.

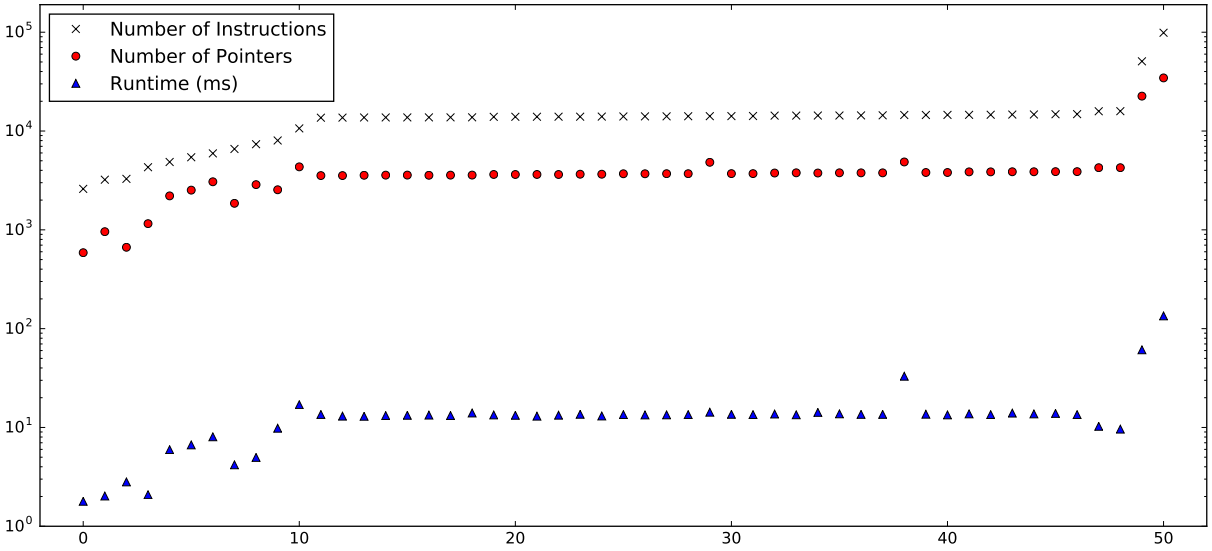


Figure II.1.16 – Runtime of our analysis for the 50 largest benchmarks in the LLVM test suite. Each point on the X-axis represents a different benchmark. Benchmarks are ordered by size. This experiment took less than 10 seconds.

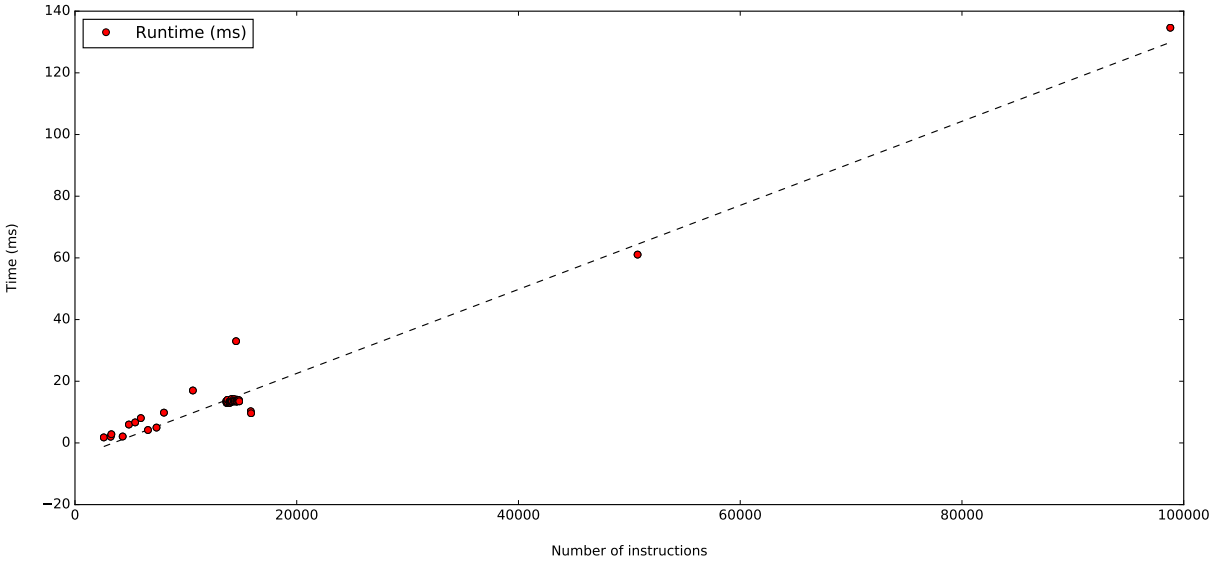


Figure II.1.17 – Linear correlation between time and number of instructions for the programs seen in Figure II.1.16.

Charts in Figure II.1.16 and Figure II.1.17 provide strong visual indication of the linear behavior of our algorithm. We have found, indeed, cogent evidence pointing in this direction. The linear correlation coefficient (R) indicates how strong is a linear relationship between two variables. The closer to one, the more linear is the correlation. In Figure II.1.17 we illustrate the linear correlation between time and number of instructions for the programs seen in Figure II.1.16 which value is 0.982. The correlation between time and number of pointers is 0.975.

II.1.7 Discussion

The contribution of this work is a new representation of pointers, based on the SymbRangeslattice, and an algorithm to reach a fixpoint in this lattice, based on abstract interpretation. This contribution complements classic work on pointer analysis. In other words, our representation of pointers can be used to enhance the precision of algorithms such as Steensgard's [Ste96], Andersen's [And94], or even the state-of-the-art technique of Hardekopf and Lin [HL11]. These techniques map pointers to sets of locations, but they could be augmented to map pointers to sets of locations plus ranges. Furthermore, the use of our approach does not prevent the employment of acceleration techniques such as lazy cycle detection [HL07], or wave propagation [PB09].

In Chapter I.3, we presented previous work that use lattices similar to ours but use different resolution algorithms. For instance, much of the work on automatic parallelization has some way to associate symbolic offsets, usually loop bounds, with pointers. The key difference between our work and this line of research is the algorithm to solve pointer relations: they resort to integer linear programming (ILP) or the Greatest Common Divisor test to solve diophantine equations, whereas we do abstract interpretation. Even Rugina and Rinard [RR05], who we believe is the state-of-the-art approach in the field today, use integer linear programming to solve symbolic relations between variables. We speculate that the ILP approach is too expensive to be used in large programs; hence, concessions must be made for the sake of speed. For instance, whereas the previous literature that we know restricts their work to pointers within loops, we can analyze programs with over one million assembly instructions in a few seconds.

As we presented in Section I.3.2, there exists work that, like ours, also associates intervals with pointers, and solves static analysis via abstract interpretation techniques. However, to the best of our knowledge, these approaches have a fundamental difference with our work: they use integer intervals à la Cousot [CC77], whereas we use symbolic intervals. The inspiration for much of this work springs from Balakrishnan and Reps notion of *Value Set* Analysis [BR04]. Integer intervals have also being used by Yong *et al.* [YH04] and, more recently, by Oh *et al.* [Oh+14]. In the latter case, Oh *et al.* use pointer disambiguation incidentally, to demonstrate their ability to implement efficiently static analyses in a context-sensitive way.

We have not implemented value set analysis [BR04] to evaluate the additional precision induced by the use of *symbolic ranges* instead of numeric ranges, but we have tried a simple experiment: in our final results, we counted the number of pointers that have integer ranges, and compared this number against the quantity of pointers that have symbolic ranges. We found out that 20.47% of the pointers in our three benchmark suites have exclusively symbolic ranges. Classic range analysis would not be able to distinguish them. Notice that numeric ranges are more common among pointer variables than among integers, because fields within `structs` – a very common construct in C – are indexed through integers.

II.1.8 Conclusion

In this chapter, we use an abstract interpretation-based algorithm where we compute offsets from *abstract* program locations having in mind both the expressivity and the efficient computation of the fixpoint itself.

Our analysis strongly relies on a preprocessing that computes *symbolic ranges* of numerical variables that lets us achieve the expressivity that also comes from the design of an efficient splitting strategy tailored to our particular abstract domain. The solution that we have proposed is sparse and relies on Extended Static Single Assignment e-SSA form [BGS00] to achieve efficiency. The fact that we use Bodik's e-SSA form distinguishes our abstract interpretation algorithm from

previous work. This representation lets us solve our analysis sparsely, whereas Balakrishnan's algorithm works on a dense representation that associates facts with pairs formed by variables and program points. We demonstrated indeed, that working on the LLVM IR is the key for designing analyses that scale.

In the next chapter, we develop a second novel alias analysis algorithm to solve pointer arithmetic that this range-based analysis do not handle.

Chapter II.2

Pointer Disambiguation via Strict Inequalities

Contents

In Chapter II.1 we designed an alias analysis algorithm based on the symbolic range analysis of integer variables. In this chapter, we introduce a second technique to solve pointer arithmetic. In the first section, we motivate this analysis and show how it handles another pointer relation class. After presenting an overview of this analysis, we recall the program representation we shall use. The algorithm of our analysis, called "less-than check", is detailed in Section II.2.4. We close this chapter by an evaluation section that shows our experiments in the LLVM compiler and discuss its performance and implementation compared to state-of-the-art techniques.

II.2.1 Context and Motivation

Pointer arithmetics come from the ability to associate pointers with offsets. As we saw in Chapter I.3, much of the work on automatic parallelization and loop vectorization consists in the design of techniques to distinguish offsets from the same base pointer. The abundant state-of-the-art approaches as well as the new alias analysis technique we introduced in Chapter II.1 try to associate intervals, numeric or symbolic, with pointers. They present indeed different ways to build Balakrishnan and Reps' notion of *value sets* [BR04]. And yet, as expressive and powerful as such approaches are, they fail to disambiguate locations that are obviously different, as `v[i]` and `v[j]`, in the loop:

$$\texttt{for}(\texttt{i} = 0, \texttt{j} = \texttt{N}; \texttt{i} < \texttt{j}; \texttt{i}++, \texttt{j}--)\ \texttt{v[i]} = \texttt{v[j]};$$

To motivate the need for a new pointer analysis we show its application on the widely used program seen in Figure II.2.1. The figure displays the C implementation of a sorting routine that makes heavy use of pointers. In this case, we know that memory positions `v[i]` and `v[j]` can never alias within the same loop iteration. However, traditional points-to analyses cannot prove this fact. Typical implementations of these analyses, built on top of the work of Andersen [And94] or Steensgaard [Ste96], can distinguish pointers that dereference different memory blocks; however, they do not say much about references ranging on the same array.

```
1  void partition(int *v, int N) {
2    int i, j, p, tmp;
3    p = v[N/2];
4    for (i = 0, j = N - 1;; i++, j--) {
5      while (v[i] < p) i++;
6      while (p < v[j]) j--;
7      if (i >= j)
8        break;
9      tmp = v[i];
10     v[i] = v[j];
11     v[j] = tmp;
12   }
13 }
```

Figure II.2.1 – A snippet of C code that challenges typical pointer disambiguation approaches.

Still, none of the pointer analyses specifically designed to deal with pointer arithmetics work satisfactorily for the example seen in Figure II.2.1. The reason for this ineffectiveness lies on the fact that these analyses use range intervals to disambiguate pointers. In our example, the ranges of integer variables `i` and `j` overlap. Consequently, any conservative range analysis, à la Cousot [CC77], once applied on Figure II.2.1, will conclude that `i` exists on the interval

$[0, N-2]$, and that j exists on the interval $[1, N-1]$. Because these two intervals have non-empty intersection, points-to analyses based on the interval lattice will not be able to disambiguate the memory accesses at lines 9-11 of Figure II.2.1.

In this chapter, we present a simple and efficient solution to this shortcoming. The technique that we introduce can disambiguate every use of v[i] and v[j] in the example.

We say that v[i] and v[j] are obviously different locations because $i < j$. There are techniques to compute *less-than* relations between integer variables in programs [BGS00; LF08; LF10; Min06]. Nevertheless, so far, they have not been used to disambiguate pointer locations. The insight that such approaches are effective and useful to such purpose is the key contribution of the technique we present in this chapter. However, we go beyond: we rely on recent advances on the construction of sparse dataflow analyses [Tav+14] to design an efficient way to solve less-than inequalities. The sparse implementation lets us view this problem as an instance of the abstract interpretation framework; hence, we get correctness for free. The end result of our tool is a less-than analysis that can be augmented to handle different program representations. This analysis can increase in non-trivial ways the ability of compilers to distinguish pointers.

To demonstrate this last statement, we have implemented our static analysis in the LLVM compiler [LA04]. We show empirically that industrial-quality alias analyses still leave unresolved pointers that our simple technique can disambiguate. As an example, we distinguish 11,881 pairs of pointers in SPEC's lbm, whereas LLVM's analyses distinguish only 1,888 pairs. Furthermore, by combining our approach with basic heuristics, we obtain even more impressive results. For instance, our less-than check increases the success rate of LLVM's basic disambiguation heuristic from 48.12% (1,705,559 queries) to 64.19% (2,274,936) in SPEC's gobmk.

II.2.2 Overview

Our new analysis lets us disambiguate the locations v[i] and v[j] in Figure II.2.1. The key to this success is the observation that $i < j$ at every program point where we have access to v. We conclude that $i < j$ by means of a "less-than check". A less-than check is a relationship between two variables that is true whenever we can prove – statically – that one holds a value lesser than the value stored in the other. We know that $i < j$ due to the conditional check at line 7.

A more precise alias analysis brings many advantages to compilers. One of such benefits is optimizations (the extra precision gives compilers information to carry out more extensive transformations in programs) and verification. In the example of Figure II.2.1, if we can prove that v[i] and v[j] do not reference overlapping memory locations, we may assert that the array partition is well performed.

II.2.3 Pre-Analysis

II.2.3.1 The core language

We use a core language to formalize the developments that we present in this chapter. Figure II.2.2 shows the syntax of this language. Our core language contains only those instructions that are essential to describe our static analysis. The reader can augment it with other assembly instructions, to make it as expressive as any industrial-strength program representation. As a testimony to this fact, the implementation that we describe in Section II.2.5 comprises the entire LLVM intermediate representation (Section I.2.3.4). Figure II.2.2 describes programs in Static Single Assignment form [Cyt+91]; therefore, it contains ϕ-functions. Additionally, it contains

arithmetic instructions and conditional branches. These two kinds of instructions feed our static analysis with new information.

Integer constants	::=	$\{c_1, c_2, \ldots\}$
Variables	::=	$\{x_1, x_2, \ldots\}$
Program (P)	::=	$\{\ell_1 : I_1; \ldots, \ell_n : I_n;\}$
Instructions (I)	::=	
– Addition	|	$x_0 = x_1 + x_2$
– ϕ-function	|	$x_0 = \phi(x_1 : \ell_1, \ldots, x_n : \ell_n)$
– Comparison	|	$(x_1 < x_2)$? goto ℓ_t : goto ℓ_f

Figure II.2.2 – The syntax of our language. Variables have a scalar type, e.g., either integer or pointer.

Example 10 *Figure II.2.3 describes a program in our core language. This is an artificial example, whose semantics is trivial. Figure II.2.3 illustrates a few key properties of the strict SSA representation: (i) the definition point of a variable dominates all its uses, and (ii) if two variables interfere, one of them is alive at the definition point of the other. Such properties will be useful in Section II.2.4.3.*

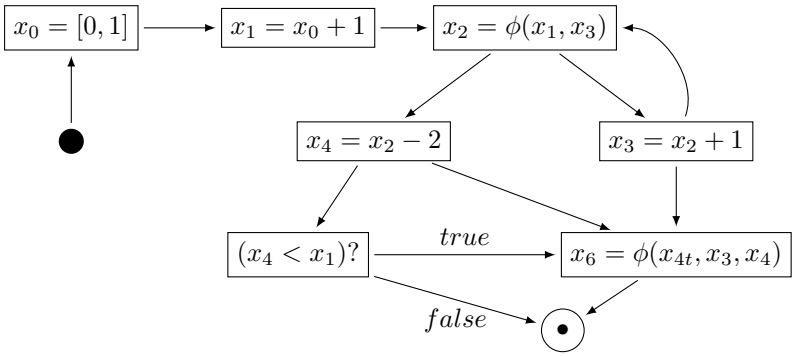


Figure II.2.3 – Program written in our core language.

II.2.3.2 Program representation

For efficiency reasons, we want to implement a *sparse dataflow analysis*. A dataflow analysis is said to be sparse if it runs on a program representation that ensures the *Static Single Information* (SSI) Property [Tav+14] defined in Section I.1.2.

Following Tavares *et al.* [Tav+14], to ensure the SSI property, we split the live range of every variable x at each program point where new information about x can appear. The live range of a variable v is the collection of program points where v is alive. To split the live range of x at a point ℓ, we create a copy $x' = x$ at ℓ, and rename every use of x at every program point dominated by ℓ. We shall write ℓ dom ℓ' to indicate that ℓ dominates ℓ', meaning that any path from the beginning of the control flow graph to ℓ' must cross ℓ. There are three situations that create new less-than information about a variable x:

1. x is defined. For instance, if $x = x' + 1$, then we know that $x' < x$;

2. x is used in a subtraction, e.g,. $x' = x + n, n < 0$. In this case, we know that $x' < x$;
3. x is used in a conditional, e.g., $x < x'$. In this case, we know that $x < x'$ at the true branch, and $x' \leq x$ at the false branch.

The Support of Range Analysis on Integer Intervals. The SSA representation ensures that a new name is created at each program point where a variable is defined. To meet the SSI requirement, we split live ranges at subtractions and after conditionals. Going back to Figure II.2.2, we see that our core language contains only syntax for arithmetic additions. However, we can use range analysis to know that one, or the two, terms of an addition are negative. Recall that the range analysis (Section II.1.3.3) is a static dataflow analysis that associates each variable x to an interval $\mathtt{R}(x) = [l, u], \{l, u\} \subset (\mathbb{N} \cup \{-\infty, +\infty\}), l \leq u$. In our experiments, we have used the implementation of Rodrigues *et al.* [RCP13]. Given $x_1 = x_2 + x_3$, where $R(x_2) = [l_2, u_2]$ and $R(x_3) = [l_3, u_3]$, we have a subtraction if $u_3 < 0$ or $u_2 < 0$. If both variables have positive ranges, then we have an addition. Otherwise, we have an *unknown* instruction, which shall not generate constraints.

Our live range splitting strategy leads to the creation of a different program representation. Figure II.2.4 shows the instructions that constitute the new language. Figure II.2.5 shows the two syntactic transformations that convert a program written in the syntax of Figure II.2.2 into a program written in the syntax of Figure II.2.4. We let $x_0 = x_1 + n \parallel \langle x_2 = x_1 \rangle$ denote a composition of two statements, $x_0 = x_1 - n$ and $x_2 = x_1$. The second instruction splits the live range of x_1. Both statements happen in parallel. Thus, $x_0 = x_1 - n \parallel \langle x_2 = x_1 \rangle$ does not represent an actual assembly instruction; it is only used for notational convenience. Similarly, when transforming conditional tests, we let $\langle x_{1t} = x_1, x_{2t} = x_2 \rangle$ denote two copies that happen in parallel: $x_{1t} = x_1$, and $x_{2t} = x_2$. Whenever there is no risk of ambiguity, we write simply $\langle x_{1t}, x_{2t} \rangle$, as in Figure II.2.4. Parallel copies and ϕ-functions are removed before code generation, after the analyses that require them have already run. This step is typically called *SSA-Elimination phase* (Section I.1.2).

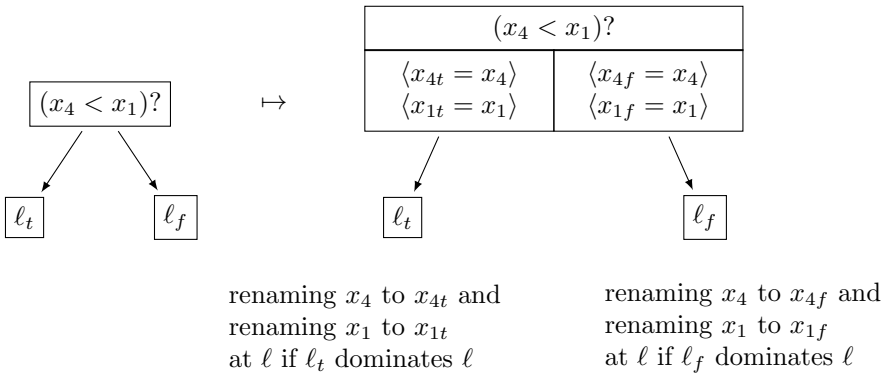
Instructions (I)	::=	
– Addition	|	$x_0 = x_1 + x_2$
– Subtraction	|	$x_0 = x_1 - n \parallel \langle x_2 = x_1 \rangle$
– ϕ-function	|	$x_0 = \phi(x_1 : \ell_1, \ldots, x_n : \ell_n)$
– Comparison	|	$(x_1 < x_2)? \begin{cases} \ell_t : \langle x_{1t}, x_{2t} \rangle \\ \ell_f : \langle x_{1f}, x_{2f} \rangle \end{cases}$

Figure II.2.4 – The syntax of our intermediate language.

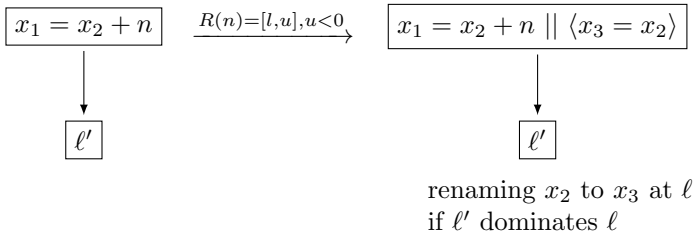
Example 11 *Figure II.2.6 shows the result of applying the rules seen in Figure II.2.5 onto the program in Figure II.2.3.*

II.2.4 The Less Than Check

This section introduces a dataflow analysis whose goal is to construct a "less-than" set for each variable x (pointer or numeric, as we will discuss in Section II.2.4.4). We denote such an object by $\mathtt{LT}(x)$. As we prove in Section II.2.4.3, the important invariant that this static analysis



(a)



(b)

Figure II.2.5 – Transformation rules used to convert the syntax in Figure II.2.2 into the syntax in Figure II.2.4.

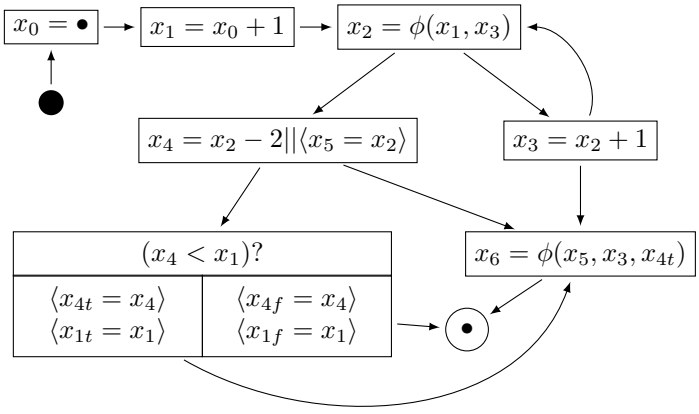


Figure II.2.6 – Figure II.2.3 transformed by the rules in Figure II.2.5.

guarantees is that if $x' \in \mathsf{LT}(x)$, then $x' < x$ at every program point where both variables are alive. Our ultimate goal is to use this invariant to disambiguate pointers, as we explain in Section II.2.4.4.

II.2.4.1 Constraint generation

Once we have a suitable program representation, we use the rules in Figure II.2.7 to generate constraints. These constraints determine the less-than set of variables. Constraint generation is $O(|\mathcal{V}|)$, where $\mathcal{V}$ is the set of variables in the target program. We have four kinds of constraints:

init: Set the less-than set of a variable to empty, e.g., $\mathsf{LT}(x) = \emptyset$.

union: Set the less-than set of a variable to be the union of another less-than set and a single element, e.g., $\mathsf{LT}(x_3) = \{x_1\} \cup \mathsf{LT}(x_2)$.

inter: Set the less-than set of a variable to be the intersection of multiple less-than sets, e.g., $\mathsf{LT}(x) = \mathsf{LT}(x_1) \cap \mathsf{LT}(x_2)$.

copy: Sets the less-than set of a variable to be the less-than set of another variable, e.g., $\mathsf{LT}(x) = \mathsf{LT}(x')$.

$$x = \bullet \rightsquigarrow_1 \mathsf{LT}(x) = \emptyset$$

$$x_1 = x_2 + n \rightsquigarrow_2 \mathsf{LT}(x_1) = \{x_2\} \cup \mathsf{LT}(x_2)$$

$$x_1 = x_2 - n \parallel \langle x_3 = x_2 \rangle \rightsquigarrow_3 \begin{cases} \mathsf{LT}(x_3) = \{x_1\} \cup \mathsf{LT}(x_2) \\ \mathsf{LT}(x_1) = \emptyset \end{cases}$$

$$x = \phi(x_1, \ldots, x_n) \rightsquigarrow_4 \mathsf{LT}(x) = \mathsf{LT}(x_1) \cap \ldots \cap \mathsf{LT}(x_n)$$

$$(x_1 < x_2)? \begin{cases} \ell_t : \langle x_{1t}, x_{2t} \rangle \\ \ell_f : \langle x_{1f}, x_{2f} \rangle \end{cases} \rightsquigarrow_5 \begin{cases} \mathsf{LT}(x_{2t}) = \{x_{1t}\} \cup \mathsf{LT}(x_1) \\ \mathsf{LT}(x_{1f}) = \mathsf{LT}(x_2) \\ \mathsf{LT}(x_{1t}) = \mathsf{LT}(x_1) \\ \mathsf{LT}(x_{2f}) = \mathsf{LT}(x_2) \end{cases}$$

Figure II.2.7 – Constraint generation rules. Numbers labeling each rule are used in the proofs of Section II.2.4.3. n is a variable such that $R(n) = [l, u], l > 0$.

Example 12 *The rules in Figure II.2.7 produce the following constraints for the program in Figure II.2.6:*

- $LT(x_0) = \emptyset$,
- $LT(x_1) = \{x_0\} \cup LT(x_0)$,
- $LT(x_2) = LT(x_1) \cap LT(x_3)$,
- $LT(x_3) = \{x_2\} \cup LT(x_2)$,
- $LT(x_4) = \emptyset$,
- $LT(x_5) = \{x_4\} \cup LT(x_3)$,
- $LT(x_{1t}) = \{x_{4t}\} \cap LT(x_{4t})$,
- $LT(x_{1f}) = LT(x_1)$,
- $LT(x_{4f}) = LT(x_1)$,
- $LT(x_{4t}) = LT(x_4)$,
- $LT(x_6) = LT(x_3) \cap LT(x_{4f}) \cap LT(x_5)$.

II.2.4.2 Constraint solving

Constraints are solved via a worklist algorithm. We initialize $\mathsf{LT}(x)$ to $\mathcal{V}$, for every variable x. During the resolution process, elements are removed from each LT, until a fixpoint is achieved. Theorem 2, in Section II.2.4.3, guarantees that this process terminates. Constraint solving is equivalent to finding transitive closures; thus, it is $O(|\mathcal{V}|^3)$. In practice, we have observed an $O(|\mathcal{V}|)$ behavior, as we show in Section II.2.5.

Example 13 *To solve the constraints in Example 12, we initialize every LT set to $\{x_0, x_1, x_2, x_3, x_4, x_6, x_{1f}, x_{1t}, x_{4f}, x_{4t}\}$, i.e., the set of program variables. Chaotic iterations [NNH05, p-176] on those constraints achieves the following fixpoint: $LT(x_0) = LT(x_4) = LT(x_{4t}) = \emptyset$; $LT(x_1) = LT(x_2) = LT(x_{4f}) = LT(x_{1f}) = LT(x_6) = \{x_0\}$; $LT(x_3) = \{x_0, x_2\}$; $LT(x_5) = \{x_0, x_4\}$; and $LT(x_{1t}) = \{x_{4t}\}$.*

II.2.4.3 Properties

There are a number of properties that we can prove about our dataflow analysis. In this section we focus on two core properties: *termination* and *adequacy*. Termination ensures that the constraint solving approach of Section II.2.4.2 always reaches a fixpoint. Adequacy ensures that our analysis conforms to the semantics of programs. To show termination, we start by proving Lemma 1.

Lemma 1 (Decreasing) *If constraint resolution starts with $LT(x) = \mathcal{V}$ for every x, then $LT(x)$ is monotonically decreasing or stationary.*

> **Proof:** The proof follows from a case analysis on each constraint produced in Figure II.2.7, plus induction on the number of elements in LT. Figure II.2.7 reveals that we have only three kinds of constraints:
>
> - $\mathsf{LT}(x) = \emptyset$: in this case, $\mathsf{LT}(x)$ is stationary;
> - $\mathsf{LT}(x) = \{x'\} \cup \mathsf{LT}(x'')$: by induction, $\mathsf{LT}(x'')$ is decreasing or stationary, and $\mathsf{LT}(x)$ always contains $\{x'\}$;
> - $\mathsf{LT}(x) = \mathsf{LT}(x_1) \cap \ldots \cap \mathsf{LT}(x_n)$: we apply induction on each $\mathsf{LT}(x_i), 1 \leq i \leq n$. □

To prove Theorem 2, which states termination, we need to recall $\mathcal{P}^{\mathcal{V}}$, the semi-lattice that underlines our less-than analysis. $\mathcal{P}^{\mathcal{V}} = \{\mathcal{V}, \cap, \bot = \emptyset, \top = \mathcal{V}, \subseteq\}$ is the lattice formed by the partially ordered set of program variables. Ordering is given by subset inclusion $\subseteq$. The meet operator (greatest lower bound) is set intersection $\cap$. The lowest element in this lattice is the empty set, and the highest is $\mathcal{V}$.

Theorem 2 (Termination) *The constraint resolution process terminates.*

> **Proof:** The proof of this theorem is the conjunction of two facts: (i) Constraint sets are monotonically decreasing; and (ii) they range on a finite lattice. Fact (i) follows from Lemma 1. Fact (ii) follows from the definition of $\mathcal{P}^{\mathcal{V}}$. □

We followed the framework of Tavares *et al.* [Tav+14] to build our intermediate program representation. Thus, we get correctness for free, as we are splitting live ranges at every program point where new information can appear. Lemma 2 formalizes this notion.

Lemma 2 (Sparcity) *$LT(x)$ is invariant along the live range of x.*

> **Proof:** The proof follows from a case analysis on the constraint generation rules in Figure II.2.7. By matching constraints with the syntax that produce them, we find that the abstract state of a variable can only change at its definition point. This property is ensured by the live range splitting strategy that produces the program representation that we use. □

We now show the adequacy of our analysis. In a nutshell, we want to show that if our constraint system determines that a variable x belongs into the less-than set of another variable x', then we know that $x < x'$. This is a static notion: we consider the values of x and x' that exist at the same moment during the execution of a program. Theorem 3 formalizes this observation.

The theorem is proved by induction on the syntax of the transformed language (Figure II.2.4). The operational semantics of this language is a minor variation of the SSA semantics [FP11; Pop06] extended with parallel copies [DF16] and tests. This semantics defines a small-step transition rule $\rightarrow$, which receives an instruction ι, plus a store environment Σ (a map from variables to integer values), and produces a new environment Σ' (a modified version of the store).

Analogously, the constraints of Figure II.2.7 show how instructions change the abstract state LT. We write $\iota \vdash \mathsf{LT} \rhd \mathsf{LT}'$ to denote an abstract transition. We let $\models$ model the following relation: if $x' \in \mathsf{LT}(x)$, then $\Sigma(x') < \Sigma(x)$. If this relation is true for every element in the domain of Σ, then we write $\mathsf{LT} \models \Sigma$.

Theorem 3 (Adequacy) *If $LT_1 \models \Sigma_1 \wedge \iota \vdash \Sigma_1 \rightarrow \Sigma_2 \wedge \iota \vdash LT_1 \rhd LT_2$, then $LT_2 \models \Sigma_2$.*

Proof: The proof follows by a case analysis on the five instructions in Figure II.2.7. In the sequel, we show two cases: the second one (similar to the third) and the fourth one. The first and fifth cases are straightforward.
First case: ι is $x_1 = x_2 + n$. Notice that $n > 0$ because otherwise our range analysis would have led us to transform that instruction into a subtraction, or would have produced no constraint at all. Henceforth, we shall write "$f \setminus a \mapsto b$" to denote function update, i.e.: "λx.if $x = a$ then b else $f(x)$". We have that $x_1 = x_2 + n \vdash \Sigma_1 \rightarrow (\Sigma_1 \setminus x_1 \mapsto \Sigma_1(x_2) + n)$, and $x_1 = x_2 + n \vdash \mathsf{LT}_1 \rhd (\mathsf{LT}_1 \setminus x_1 \mapsto \mathsf{LT}_1(x_2) \cup \{x_2\})$. We look into possible variables $x \in \mathsf{LT}_1 \setminus x_1 \mapsto \mathsf{LT}_1(x_2) \cup \{x_2\}$:

- $x = x_2$: the theorem is true because $\Sigma_1(x_2) + n > \Sigma_1(x_2)$;
- $x \in \mathsf{LT}(x_2)$: From the hypothesis $\mathsf{LT}_1 \models \Sigma_1$, we have that $x < x_2$. We know that $x_2 < x_1$; thus, by transitivity, $x < x_1$.

Second case: ι is a ϕ-function. Then, $x = \phi(x_1, \ldots, x_n) \vdash \Sigma_1 \rightarrow \Sigma_1 \setminus x \mapsto \Sigma_1(x_i)$, for some $i, 1 \leq i \leq n$, depending on the program dynamic control flow. From Figure II.2.7, we have that $\mathsf{LT}_2 = \mathsf{LT}_1 \setminus x \mapsto \mathsf{LT}_1(x_1) \cap \ldots \cap \mathsf{LT}_1(x_n)$. Thus, any $x' \in \mathsf{LT}_2(x)$ is such that $x' \in LT_1(x_i)$. By the hypothesis, $x' < x_i$ for any x_i. By the semantics of ϕ-functions, $x = x_i$; hence, $x' < x$. □

Corollary 1 (Invariance) *Let x_i and x_j be two variables simultaneously alive. If $x_i \in LT(x_j)$, then $x_i < x_j$.*

Proof: In an SSA-form program, if two variables interfere, then one is alive at the definition point of the other [Bud+02; HGG06; Zha+13b]. From Lemma 2, we know that $\mathsf{LT}(x)$ is constant along the live range of x. Theorem 3 gives us that this property holds at the definition point of the variables. □

II.2.4.4 Pointer disambiguation

Pointers, in low-level languages, are used in conjunction with integer offsets to refer to specific memory locations. The combination of a base pointer plus an offset produces what we call a *derived pointer*. The less-than check that we have discussed in this chapter lets us compare pointers directly, if they are bound to a less-than relation, or indirectly, if they are derived from a common base. This observation lets us state the disambiguation criteria below:

Definition 10 (Pointer Disambiguation Criteria) *Let p, p_1 and p_2 be variables of pointer type, and x_1 and x_2 be variables of arithmetic type. We consider two disambiguation criteria:*

1. *Memory locations p_1 and p_2 will not alias if $p_1 \in LT(p_2)$ or $p_2 \in LT(p_1)$.*
2. *Memory locations $p_1 = p + x_1$ and $p_2 = p + x_2$ will not alias if $x_1 \in LT(x_2)$ or $x_2 \in LT(x_1)$.*

The C standard refers to arithmetic types and pointer types collectively as scalar types [ISO11, §6.2.5.21]. Notice that the less-than analysis that we have discussed so far works seamlessly for scalars; thus, it also builds relations between pointers. For instance, the common idiom "`for(int*pi = p;pi < pe;pi++);`" gives us that $\texttt{pi} < \texttt{pe}$ inside the loop. This fact justifies Definition 10.1. Along similar lines, if $\texttt{p}_1 = \texttt{p} + \texttt{x}_1$, we have that $\texttt{p} \in \mathsf{LT}(\texttt{p}_1)$; thus, Definition 10.2 lets us disambiguate a base pointer from its non-null offsets, e.g., $\texttt{p} \neq \texttt{p} + \texttt{n}$, if we know that $\texttt{n} \neq 0$.

Example 14 *Going back to the example of Figure II.2.1, we conclude using the criterion of Definition 10.1 that memory locations* `v[i]` *and* `v[j]` *do not alias in the false branch ($\texttt{i}_\texttt{F} \in LT(\texttt{j}_\texttt{F})$ in Example 13).*

Definition 10 provides one, among several criteria, that can be used to disambiguate pointers. For instance, the C standard says that pointers of different types cannot alias. Aliasing is also impossible in well-defined programs between references derived from non-aliased base pointers. Additionally, derived pointers whose offsets have non-overlapping ranges cannot alias, as discussed in previous work [BR04; RR05] and in Chapter II.1. Thus, our analysis says nothing about `p1` and `p2` in scenarios as simple as: `p1 = malloc(); p2 = malloc()`, or `p1 = p + 1; p2 = p + 2`. Our algorithm of Chapter II.1 is already able to disambiguate `p1` and `p2` in both cases. Our pointer disambiguation criterion does not compete against these other approaches. Rather, as we further explain in Section II.2.6, it complements them.

II.2.5 Evaluation and Experiments

To demonstrate that a "less-than" check can be effective and useful to disambiguate pointers, we have implemented an inter-procedural, context-insensitive version of the analysis described in this chapter in LLVM version 3.7. In this section we shall answer the three research questions that we have seen in Section I.1.3.3.1 to evaluate the precision, the scalability and the applicability of our approach:

Precision: how effective are strict inequalities to disambiguate pairs of pointers?

Scalability: can the analysis described scale up to handle very large programs?

Applicability: can our pointer disambiguation method increase the effectiveness of existing program analyses?

In the rest of his section we provide answers to these questions. Our discussion starts with precision.

II.2.5.1 Precision

The precision of an alias analysis method is usually measured as the capacity of the said method to indicate that two given pairs of pointers do not alias each other. To measure the precision of our method, we compare it against the techniques already in place in the LLVM compiler. Our metric is the percentage of improvement our algorithm adds on top of LLVM's basic disambiguation technique, the `basic-aa` algorithm. Henceforth, we shall refer to it as **BA**. This analysis uses several heuristics to disambiguate pointers, relying mostly on the fact that pointers derived from different allocation sites cannot alias in well-formed programs.

In addition to the basic algorithm, LLVM contains three other alias analyses, whose results we shall not use, for they have been able to resolve a very low number of queries in our experiments. These algorithms are `tb-aa`, which reports as non-aliases pointers of different types, `globalsmodref-aa`, which reports as non-aliases global variables whose addresses have never been taken, and `scev-aa`, which uses a form of value sets to disambiguate pointers [BR04]. We chose to omit `scev-aa` because we have not been able to get results for all our benchmarks using the implementation available in LLVM 3.7. We have evaluated it before in Chapter II.1, and, on average, it solves less than 5% of all the queries.

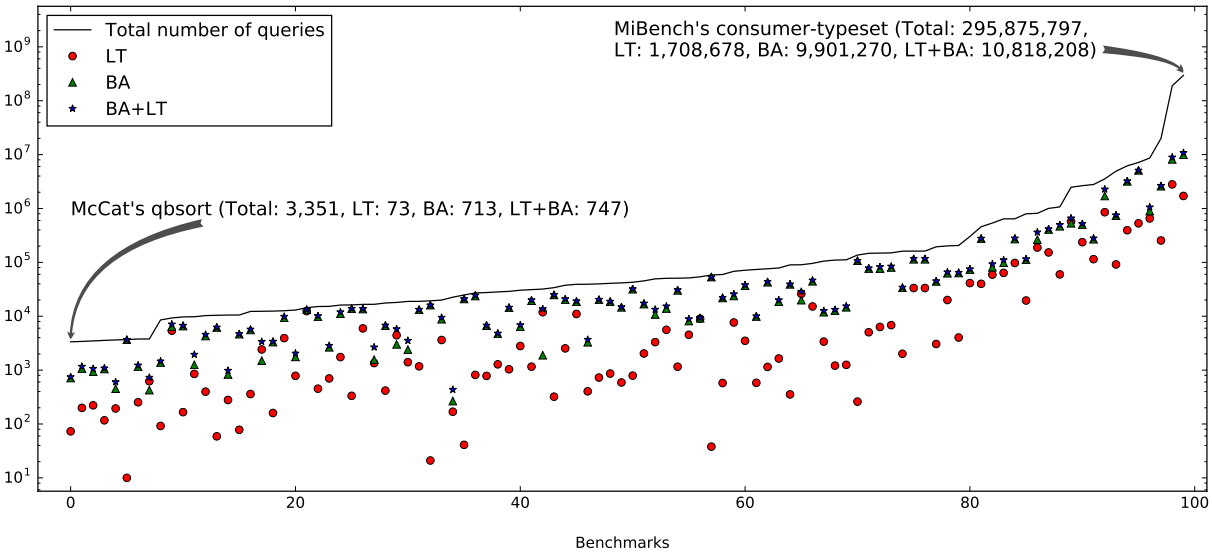


Figure II.2.8 – Effectiveness of our alias analysis (**LT**), when compared to LLVM's basic alias analysis on the 100 largest benchmarks in the LLVM test suite. Each tick in the X-axis represents one benchmark. The Y-axis represents total number of queries (one query per pair of pointers), and number of queries in which each algorithm got a "no-alias" response.

Figure II.2.8 shows the results of the three alias analyses when applied on the 100 largest benchmarks in the LLVM test suite. We have removed the benchmark TSVC from this lot, because its 36 programs were giving us the same numbers. This fact occurs because they use a common code base. Our method rarely disambiguates more pairs of pointers than **BA**. Such result is expected: most of the queries consist of pairs of pointers derived from different memory allocation sites, which **BA** disambiguates, and we do not analyze. The ISO C Standard prohibits comparisons between two references to separately allocated objects [ISO11, §6.5.8p5], even though they are used in practice [Mem+16, p.4].

Nevertheless, Figure II.2.8 still lets us draw encouraging conclusions. There exist many queries that we can solve, but **BA** cannot. For the entire LLVM test suite, our analysis that we shall refer to as **LT**, increases the precision of **BA** by 9.49% (56,192,064 vs. 59,184,181 no-alias

Benchmark	# Queries	BA	LT	BA + LT
lbm	31,944	5.90%	10.15%	**15.74%**
mcf	49,133	15.28%	8.95%	16.52%
astar	95,098	45.54%	16.05%	47.66%
libq	146,301	51.64%	3.45%	52.67%
sjeng	428,082	70.64%	2.03%	71.64%
milc	808,471	31.05%	23.90%	**43.88%**
soplex	1,787,190	21.43%	12.48%	23.53%
bzip2	2,472,234	21.48%	23.09%	**26.70%**
hmmer	2,574,217	8.79%	4.48%	**9.38%**
gobmk	3,492,577	48.49%	22.91%	**63.33%**
namd	3,685,838	22.59%	0.93%	22.76%
omnetpp	12,943,554	18.71%	0.46%	18.81%
h264ref	20,068,605	12.86%	1.29%	13.16%
perl	23,849,576	9.92%	3.87%	10.19%
dealII	37,779,455	75.05%	20.21%	75.46%
gcc	186,008,992	4.26%	1.47%	4.65%

Figure II.2.9 – Comparison between three alias analyses on SPEC CINT 2006. "# Queries" is the total number of queries performed when testing a given benchmark. Percentages show the ratio of queries that yield "no-alias", given a certain alias analysis. The higher the percentage, the more precise the pointer disambiguation method is. We have highlighted the cases in which our less-than check has increased by 10% or higher the precision of LLVM's basic alias analysis.

responses). Yet, in programs that make heavy use of pointer arithmetics, our results are even more impressive. For instance, in SPEC's lbm we disambiguate 11,881 pairs of pointers, whereas **BA** provides precise answers for only 1,888. And, even in situations where **LT** falls behind **BA**, the former can increase the precision of the latter non-trivially. As an example, in SPEC's `gobmk`, **LT** returns 852,368 "no-alias" answers and **BA** 1,705,559. Yet, these sets are mostly disjoint: the combination of both analyses solves 2,274,936 queries: an increase of 34% over **BA**. Figure II.2.9 summarizes these results for SPEC CINT 2006.

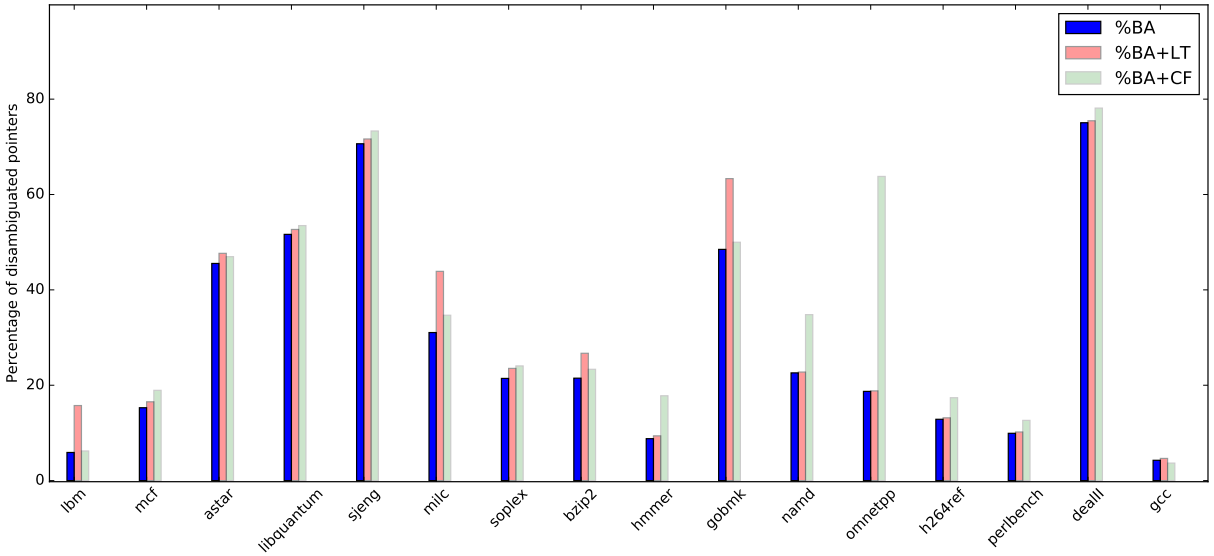


Figure II.2.10 – How two different alias analysis (**LT** and **CF**) increase the capacity of LLVM's basic alias analysis (**BA**) to disambiguate pointers. The Y-axis shows the percentage of no-alias responses. The higher the bar, the better.

How do we compare against Andersen's analysis? Andersen's [And94] inclusion-based alias analysis is the quintessential pointer disambiguation technique. At the time of this writing, the most up-to-date version of LLVM did not contain an implementation of this technique. However, there exists one algorithm available in LLVM 4.0, which is still experimental. We shall call it **CF**, because it uses context free languages (CFL) to model the inclusion-based resolution of constraints, as proposed by Zheng and Rugina [ZR08], and by Zhang *et al.* [Zha+13a]. Jia Chen [Che16] provides a detailed description of the LLVM implementation[1].

Figure II.2.10 compares our analysis and Andersen's. Our numbers have been obtained in LLVM 3.7, whereas **CF**'s has been produced via LLVM 4.0. We emphasize that both versions of this compiler produce exactly the same number of alias queries, and, more importantly, **BA** outputs exactly the same answers in both cases. This experiment reveals that there is no clear winner in this alias analysis context. **BA+LT** is more than 20% more precise than **BA+CF** in three benchmarks: `lbm`, `milc` and `gobmk`. **BA+CF**, in turn, is three times more precise in `omnetpp`. The main conclusions that we draw from this comparison are the following: (i) these analyses are complementary; and (ii) mainstream compilers still miss opportunities to disambiguate alias queries.

II.2.5.2 Scalability

We claim that the less-than analysis that we introduce in this chapter presents – in practice – linear complexity on the size of the target program. Size is measured as the number of instructions present in the intermediate representation of the said program. In this section we provide evidence that corroborates this claim. Figure II.2.11 relates the number of constraints that we produce for a program, using the rules in Figure II.2.7, with the number of instructions in that program. The strong linear relation between these two quantities is visually apparent in Figure II.2.12. And, going beyond visual clues, the coefficient of determination (R^2) between constraints and instructions is 0.992. The closer to 1.0 is R^2, the stronger the evidence of a linear behavior.

As Figure II.2.12 shows, the number of constraints that we produce is linearly proportional to the number of instructions that these constraints represent. But, what about the time to solve such constraints – is it also linear on the number of instructions? To solve constraints, we compute the transitive closure of the "less-than" relation between program variables. We use a cubic algorithm to build the transitive closure [Nuu95]. When fed with our benchmarks, this algorithm is likely to show linear behavior: the coefficient of determination between the number of constraints for all our benchmarks, and the runtime of our analysis is 0.988. This linearity surfaces in practice because most of the constraints enter the worklist at most twice. For instance, SPEC CPU 2006, plus the 308 programs that are part of the LLVM test suite, give us 8,392,822 constraints to solve. For this lot, we pop the worklist 17,800,102 times: a ratio that indicates that each constraint is visited 2.12 times on average until a fixed point is achieved.

We emphasize that our implementation still bears the status of a research prototype. Its runtime is far from being competitive, because, currently, it relies heavily on C++ standard data-structures, instead of using data-types more customized to do static analyses. We use `std::set` to represent LT sets, `std::map` to bind LT sets to variables, and `std::vector` to implement the worklist. Therefore, our implementation still has much room of improvement in terms of runtime. For instance, we took two hours and twelve minutes to solve all the less-than relations between all the scalar variables found in the 16 programs of SPEC CPU that the LLVM C frontend compiles on an 2.4GHz Intel Core i7. We have already observed that most of the LT sets

[1] Available at https://github.com/grievejia/andersen

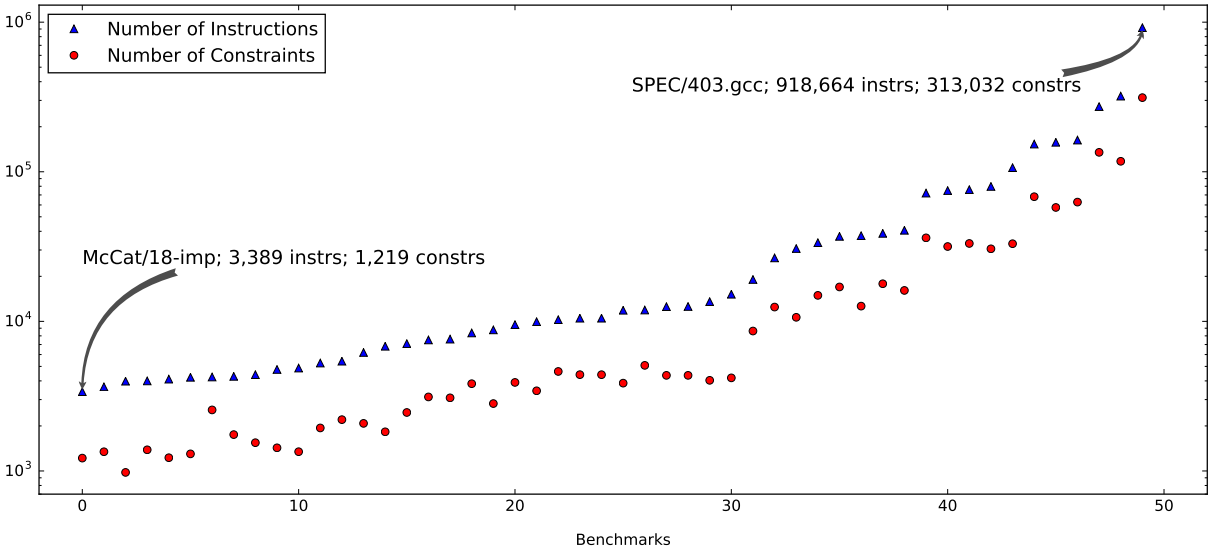


Figure II.2.11 – Comparison between the number of instructions and the number of constraints that we produce (using rules in Figure II.2.7) per benchmark. X-axis represents benchmarks, sorted by number of instructions. The coefficient of determination (R^2) between these two metrics is 0.992, indicating a strong linear correlation.

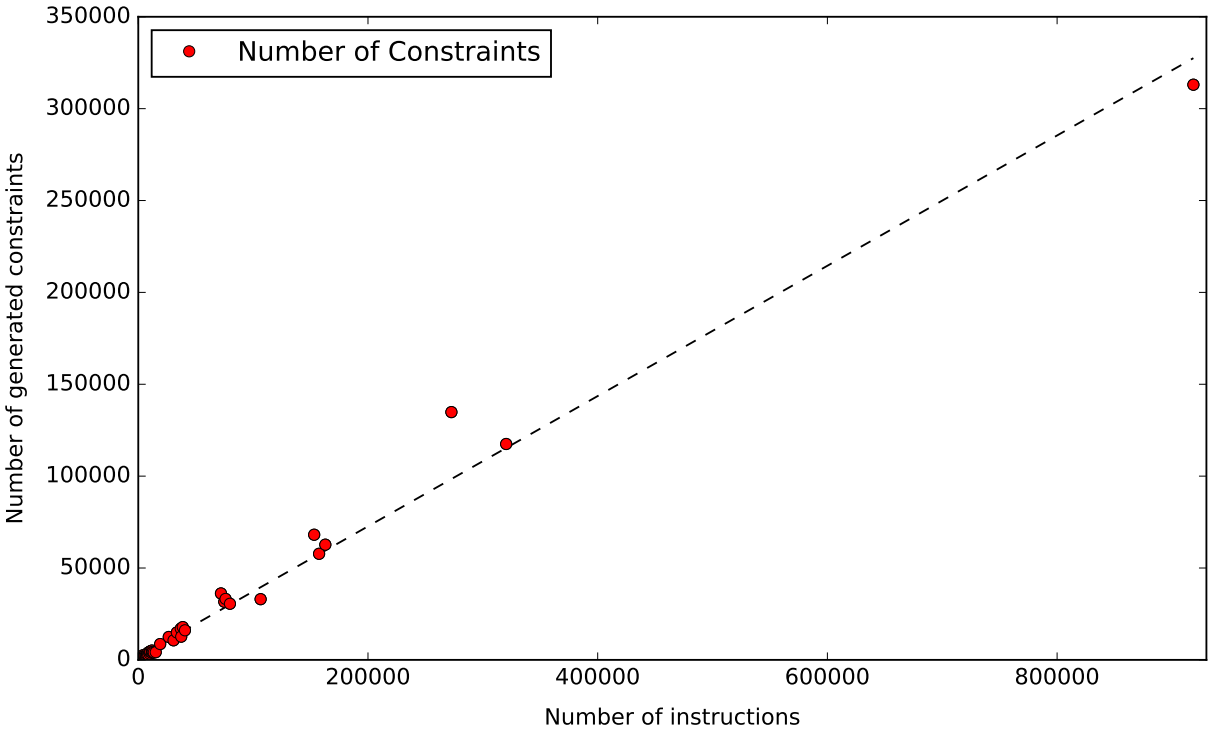


Figure II.2.12 – Linear correlation between the number of instructions and the number of constraints for the programs seen in Figure II.2.11.

end up empty, and that the vast majority of them, over 95%, contain only two or less elements. We intend for use such observations to improve the runtime of our analysis as future work.

II.2.5.3 Applicability

One way to measure the applicability of an alias analysis is to probe how it improves the quality of some compiler optimization, or the precision of other static analyses. In this work, we have opted to follow the second road, and show how our new alias analysis improves the construction of the Program Dependence Graph (PDG), a classic data structure introduced by Ferrante *et al.* [FOW87]. We use the implementation of PDGs available in the FlowTracker system [RQA16], which has a distribution for LLVM 3.7[2]. The PDG is a graph whose vertices represent program variables and memory locations, and the edges represent dependences between these entities. An instruction such as `a[i] = b` creates a data dependence edge from `b` to the memory node `a[i]`. The more memory nodes the PDG contains, the more precise it is, because if two locations alias, they fall into the same node. In the absence of any alias information, the PDG contains at most one memory node; perfect alias information yields one memory node for each independent location in the program.

It is not straightforward to compare LLVM's basic alias analysis against our less-than-based analysis, because the former is intra-procedural, whereas the latter is inter-procedural. Therefore, **BA** ends up creating at least one memory node per function that contains a load or store operation present in the target program. **LT**, on the contrary, joins nodes that exist in the scope of different functions if it cannot prove that they do not overlap. In order to circumvent this shortcoming, we decided to use Csmith [Yan+11][3]. Csmith produces random C programs that conform to the C99 standard, using an assortment of techniques, with the goal to find bugs in compilers. Csmith has one important advantage to us: we can tune it to produce programs with a single function, in addition to the ever present `main` routine. By varying the seed of its random number generator, we obtain programs of various sizes, and by varying the maximum nesting depth of pointers, we obtain a rich diversity of dependence graphs. Figure II.2.13 shows the results that we got in this experiment.

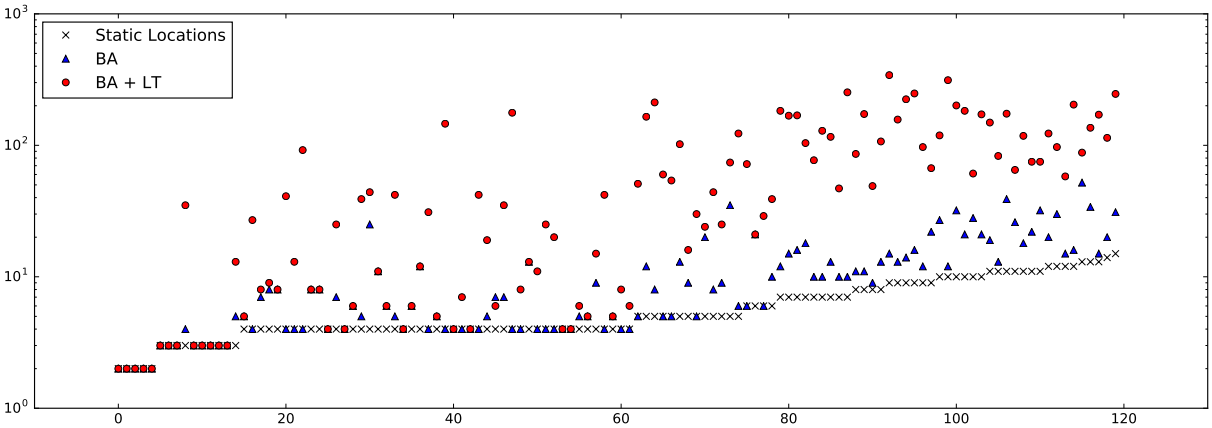


Figure II.2.13 – Precision of dependence graph. The X-axis shows benchmarks, sorted by number of static memory references in the source code. Y-axis shows number of memory nodes in the Program Dependence Graph. The more memory nodes the PDG contains, the more precise it is.

Our alias analysis improves substantially the precision of LLVM's **BA**. We have produced 120 random programs, whose size vary from 50 to 4,030 lines. In total, the 120 PDGs produced with **BA** contain 1,299 memory nodes. When combined, **BA** and **LT** yield 8,114 nodes, an increase of 6.23x. We are much more precise than **BA** because the programs that Csmith produces do not

[2] Available at http://cuda.dcc.ufmg.br/flowtracker/
[3] Available at https://embed.cs.utah.edu/csmith/

read input values: they use constants instead. Because almost every memory indexing expression is formed by constants known at compilation time, **LT** can distinguish most of them. Although artificial, this experiment reveals a striking inability of LLVM current alias analyses to deal with pointer arithmetics. None of the other alias analyses available in LLVM 3.7 are able to increase the precision of **BA** – not even marginally. Although we have not used **CF** in this experiment – it is not available for LLVM 3.7 – we speculate, from reading its source code, that it will not be able to change this scenario.

Notice that our results do not depend on the nesting depth of pointers. Our 120 benchmarks contain 6 categories of programs, which we produced by varying the nesting depth of pointers from 2 to 7 levels. Thus, we had 20 programs in each category. A pointer to `int` of nesting depth 3, for instance, is declared as `int***`. All these programs, regardless of their category, present an average of six static memory allocation sites. On average, **BA** produces PDGs with 11 memory nodes, independent on the bucket, and **BA**+**LT** produce PDGs with 68. The greater the number of static memory allocation sites, the better the results for both **BA** and **LT**. The largest PDG observed with **BA** only has 52 memory nodes (and, for the same program, it has 88 nodes if we augment **BA** with **LT**). The largest graph produced by the combination of **BA** and **LT** has 342 nodes (and only 15 nodes if we use **BA** without **LT** for the same program).

II.2.6 Discussion

In this section we discuss the relational analysis techniques that have been proposed in the literature and are related to the one we propose in this chapter. These analyses, as we saw in Chapter I.3, either do not deal with pointer disambiguation but can be accommodated for alias analysis purposes or already deal with pointer analysis but do not fit with *static* analyses inside compilers.

Before discussing the underlying techniques, we recall that semi-relational analyses associate single program variables with sets of other variables while fully-relational ones associate tuples of variables with abstract information.

Semi-Relational Alias Analyses The work that most closely resembles ours is Bodik *et al.*'s ABCD (short for Array Bounds Checks on Demand) algorithm [BGS00] (Section I.3.2). Similarities stem from the fact that Bodik *et al.* also build a new program representation to achieve a sparse less-than analysis. However, there are five key differences between that approach and ours. The first difference is a matter of presentation: Bodik *et al.* provide a geometric interpretation to the problem of building less-than relations, whereas we adopt an algebraic formalization. Bodik *et al.* keep track of such relations via a data-structure called the *inequality graph*. This graph is implicit in our approach: it appears if we create a vertex v_i to represent each program variable x_i, and add a weighted edge from v_1 to v_2 if, and only if, $v_1 \in LT(v_2)$. The weight of an edge is the difference $v_2 - v_1$, whenever known statically. The other four differences are more fundamental.

Bodik *et al.* use a different algorithm to prove that a variable is less than another. In the absence of cycles in the inequality graph, their approach works like ours: a positive path between v_i to v_j indicates that $x_i < x_j$. This path is implicit in the transitive closure that we produce after solving constraints. However, they use an extra step to handle cycles, which, in our opinion, makes their algorithm difficult to reason about. Upon finding a cycle in the inequality graph, Bodik *et al.* try to mark this cycle as *increasing* or *decreasing*. Cycles always exist due to ϕ-functions. Decreasing cycles cause ϕ-functions to be abstractly evaluated with the *minimum* operator applied on the weights of incoming edges; increasing cycles invoke *maximum* instead. Third, Bodik *et al.* do

not use range analysis. This is understandable, because ABCD has been designed for just-in-time compilers, where runtime is an issue. Nevertheless, this limitation prevents ABCD from handling instructions such as $\mathtt{x_1} = \mathtt{x_2} + \mathtt{x_3}$ if neither $\mathtt{x_2}$ nor $\mathtt{x_3}$ are constants. Fourth, Bodik *et al.*'s program representation does not split the live range of $\mathtt{x_2}$ at an instruction such as $\mathtt{x_1} = \mathtt{x_2} - \mathtt{x_3}, \mathtt{x_3} > \mathtt{0}$. This implementation detail lets us know that $\mathtt{x_2} > \mathtt{x_1}$. Finally, we chose to compute a transitive closure of less-than relations, whereas ABCD works on demand. This point is a technicality. In our experiments, we had to deal with millions of queries. If we tried to answer them on demand, like ABCD does, then said experiments would take too long. We build the transitive closure to answer queries in $O(1)$.

Logozzo and Fähndrich have proposed the Pentagon Lattice to eliminate array bound checks in type safe languages such as C#. This algebraic object is the combination of the lattice of integer intervals and the less-than lattice. Pentagons, like the ABCD algorithm, could be used to disambiguate pointers like we do. Nevertheless, there are differences between our algorithm and Logozzo's. First, the original work on Pentagons describes a dense analysis, whereas we use a different program representation to achieve sparsity. Contrary to ABCD, the Pentagon analysis infers that $\mathtt{x_2} > \mathtt{x_1}$ given $\mathtt{x_1} = \mathtt{x_2} - \mathtt{x_3}, \mathtt{x_3} > \mathtt{0}$ like we do, albeit on a dense fashion. Second, Logozzo and Fähndrich build less-than and range relations together, whereas our analysis first builds range information, then uses it to compute less-than relations. We have not found thus far examples in which one approach yields better results than the other; however, we believe that, from an engineering point of view, decoupling both analyses leads to simpler implementations.

Fully-Relational Alias Analyses As we saw in Section I.3.2, Fully-relational analyses associate tuples of variables with abstract information. Polly's dependence analysis cannot analyze `v[i]` and `v[j]` in Figure II.2.1. These analyses are very powerful; however, they face scalability problems when dealing with large programs. Whereas a semi-relational sparse analysis generates $O(|\mathcal{V}|)$ constraints, $|\mathcal{V}|$ being the number of program variables, a relational one might produce $O(|\mathcal{V}|^k)$, k being the number of variables used in the relations.

II.2.7 Conclusion

Pentagons is an abstract domain invented by Logozzo and Fähndrich to validate array accesses in low-level programming languages. This algebraic structure provides a cheap "less-than check", which builds a partial order between the integer variables used in a program. In this chapter, we saw how we have used the ideas available in Pentagons to design and implement a novel alias analysis. This new algorithm lets us disambiguate pointers with offsets, so common in C-style pointer arithmetic, in a precise and efficient way.

In the next chapter, we combine the two alias analyses we presented in Chapter II.1 and Chapter II.2 with a third new analysis. This time, we shall be interested in non-related pointers to achieve further better precision.

Chapter II.3

Combining Range and Inequality Information for Pointer Disambiguation

Contents

In Chapter II.1 and Chapter II.2 we have presented two novel static alias analysis techniques that we have implemented on top of the LLVM compiler. In this chapter, we join these two approaches that compilers use to analyze programs. To that end, we shall propose adaptations of both methods, implement them differently, to finally introduce an algorithm to combine them.

II.3.1 Context and Motivation

The key insight of the alias analysis we describe in this chapter is therefore not new but we do not know of another work that joins both these research directions into a single path. We shall discuss the key differences between the two techniques we extend and the work we present in this chapter in Section II.3.7.

We call the alias analysis approach we describe "staged", because it works in successive stages. It consists of the following five parts, which we describe in the rest of this chapter:

1. Find ranges for integer variables in the program. We explain this step in Section II.3.2.
2. Group pointers related to the same memory location. We describe this step in Section II.3.3
3. Collect constraints by traversing the program control flow graph. Section II.3.4.1 provides more details about this stage.
4. Solve the constraints produced in phase 3. Section II.3.4.2 explains this phase of our approach.
5. Answer pointer disambiguation queries. This is described in Section II.3.5.

Our analysis lets us perform different pointer disambiguation checks. As we explain in Section II.3.5, we use three different checks to verify if two pointers might alias. All these checks emerge naturally from the data-structures that we build in the effort to find less-than relations between variables. As we will show in Section II.3.7, the less-than check we present in this chapter is more precise than our previous one. To motivate the need for this algorithm, we consider the program in Figure II.3.1. The figure displays the C implementation of the insertion sort routine that makes heavy use of pointers. We know that memory positions `v[i]` and `v[j]` can never alias within the same loop iteration because `i` < `j`. This is related to the way that `j` is initialized, within the `for` statement at line 4.

```
1  void ins_sort(int* v, int N) {
2    int i, j;
3    for (i = 0; i < N - 1; i++) {
4      for (j = i + 1; j < N; j++) {
5        if (v[i] > v[j]) {
6          int tmp = v[i];
7          v[i] = v[j];
8          v[j] = tmp;
9        }
10     }
11   }
12 }
```

Figure II.3.1 – Insertion sort algorithm.

```
1  void ins_sort_opt(int* v, int N) {
2    int i, j,tmp_i,tmp_j,tmp;
3    for (i = 0; i < N - 1; i++) {
4      tmp_i=v[i];
5      for (j = i + 1; j < N; j++) {
6        tmp_j=v[j];
7        if (tmp_i > tmp_j) {
8          tmp = tmp_i;
9          tmp_i = tmp_j;
10         v[j] = tmp;
11       }
12     }
13     v[i]=tmp_i;
14   }
15 }
```

Figure II.3.2 – Implementation of insertion sort, seen in Figure II.3.1, after scalar replacement [Sur+14].

A more precise alias analysis allows compilers to carry out more extensive transformations in

programs and therefore to optimize them. Figure II.3.2 illustrates the benefit of using a precise alias information. The figure shows the result of applying Surendran's [Sur+14] inter-iteration scalar replacement on the insertion sort algorithm seen in Figure II.3.1. Scalar replacement is a compiler optimization that consists in moving memory locations to registers as much as possible. This optimization tends to speed programs up because it removes memory accesses from their source code. In this example, if we can prove that `v[i]` and `v[j]` do not reference overlapping memory locations, we can move these locations to temporary variables. For instance, we have loaded location `v[j]` into `tmp_j` at line 6. We update the value of `v[i]` at line 13.

The rest of this chapter describes an implementation of our staged analysis. Our implementation involves several steps. Throughout the chapter, we use four examples to illustrate each of these steps, when applied onto the program in Figure II.3.1. The first step consists in converting the program into a suitable intermediate representation. From this representation we extract a data-structure called a *Pointer Dependence Digraph*, as Example 16 shows. The main purpose of this graph is to let us apply alias queries on program pointers. The process of collecting constraints that describe less-than relations in the program is the subject of Example 17. A solution to this constraint system gives us, for each variable v, a set of other variables that are known to be less than v. Example 22 shows the sets that we obtain for Figure II.3.1 routine. Finally, we have different ways to perform queries on these "less-than" sets. Example 24 (page 102) clarifies this use of our analysis.

II.3.2 Program Representation and Range pre-Analysis

We solve our combined analysis through abstract interpretation. We shall then work at a low-level representation of programs and abandon the high-level C notation, in favor of the assembly-like language of Figure II.1.6 that we recall in Figure II.3.3 (with a focus on the branch instruction).

$Prog$	$::=$	I^*	; Program
I	$::=$		; Instruction
	$\mid$	$p_0 = \text{malloc}(i_0)$	; Memory allocation
	$\mid$	$v_0 = v_1 + i_0$	; Addition
	$\mid$	$v = *p$	; Load
	$\mid$	$*p = v$	; Store
	$\mid$	$v_0 = \phi(v_1 : \ell_1, v_2 : \ell_2)$	; Phi-function (*Phi*) (See [Cyt+89])
	$\mid$	$(v_0 \mathrel{\mathscr{R}} v_1)\ ?\ S^*B\ ;\ S^*B$	; Branch ($\mathscr{R} \in \{<, \leq, =, \neq, >, \geq\}$)
B	$::=$	$\ell : Phi^* I^*$	; Basic Block
S	$::=$	$v_0 = \sigma(v_1)$	; Sigma-function (See [Sin06])

Figure II.3.3 – The syntax of our language of pointers. Whenever a variable must be explicitly a pointer, we name it p. Variables that need to be integers are named i. Variables that can be either an integer or a pointer are named v. We use the Kleene Star (*) to indicate zero or more repetitions of a non-terminal.

Example 15 *Figure II.3.4 shows the* Control Flow Graph *(cfg) of the program seen in Figure II.3.1, in e-SSA form. The σ-functions rename every variable used in a comparison. Renaming lets us, for instance, infer that $x_{iT} > x_{jT}$, because: (i) these variables are copies of x_i and x_j; and (ii) they only exist in the true side of the test $x_i > x_j$.*

A core component of our pointer disambiguation method is a *range analysis*, which can be

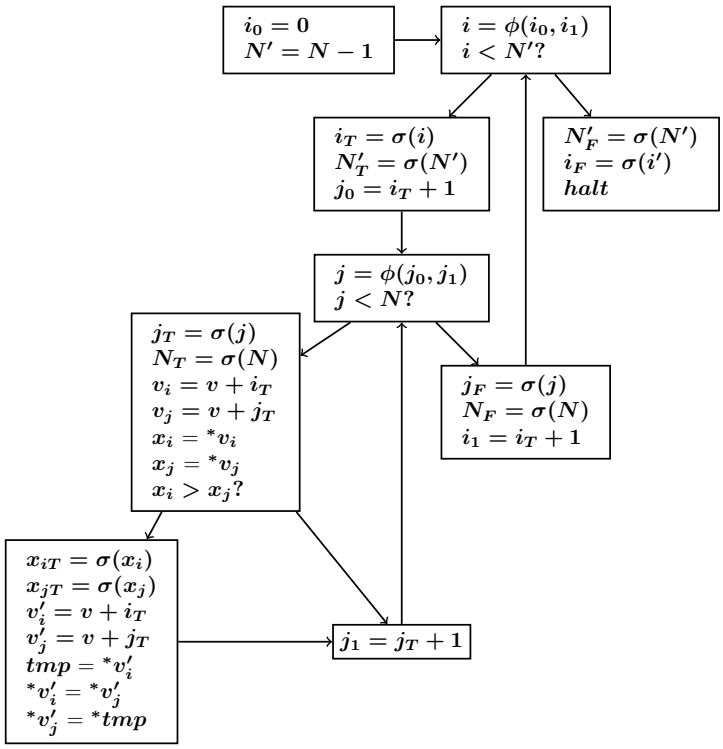


Figure II.3.4 – Control Flow Graph of program in Figure II.3.1. $i_T = \sigma(i)$ denotes the "true" version of i after the test $i < N$. $i = \phi(i_0, i_1)$ denotes the value of i inside the first loop, its value is i_0 if the flow comes from the first block, or i_1 if the flow comes from the returning edge $i_1 = i_T + 1$.

symbolic or numeric. In this context, a symbol is any name in the program syntax that cannot be built as an expression of other names.

In this work, we have adopted a non-relational numeric range analysis [RCP13] on the classic integer interval [CC77]. In the sequel, we let $\mathrm{R}(\mathtt{v}) = [l, u]$ be the range of variable `v` computed by any range analysis.

We would like to emphasize that like in Chapter II.1 and Chapter II.2 the range analysis that we shall use here to obtain intervals for integer variables is *immaterial* for the correctness of our work. The only difference they make is in terms of precision and scalability. The more precise the range analysis used, the more precise the pointer analysis produced. However, precision has a cost in time.

II.3.3 Grouping Pointers in Pointer Digraphs

Recalling the pointer disambiguation criteria of Definition 10, we know that we can disambiguate two pointers, $\mathtt{p}_1$ and $\mathtt{p}_2$, whenever we can prove that $\mathtt{p}_1 < \mathtt{p}_2$. This relation is only meaningful for pointers that are offsets from the same *base pointer*[1]. A base pointer is a reference to the zeroth address of a memory block. As an example, a statement like "`u = malloc(4)`" will create a base pointer referenced by `u`. Formal arguments of functions, such as `v` in Figure II.3.1, are also base-pointers.

[1]The ISO C Standard forbids relational comparisons between pointers, even with the same type, to different allocated objects [ISO11, Sec-6.5.8 p5].

In this section, we build a data-structure that we call *Pointer Dependence Digraph* (PDD) and shall use it to disambiguate non related pointers. The less-than check may therefore be run, with respect to the C standard, on the rest of pairs of pointers. The PDD is a directed acyclic graph (DAG) with origin at one or more base-pointers. All other nodes in this data-structure that are not base-pointers are variables that can be defined by an offset o from one of the base pointers. We let o be a symbol, such as a constant or the name of a variable, as defined in Section II.3.2. For instance, the relation $p = p_0 + o$ is represented, in the PDD, by an edge from the node p to the node p_0 labeled by $\omega_{p,p_0} = o$. We denote this edge by $p \rightarrow_o p_0$.

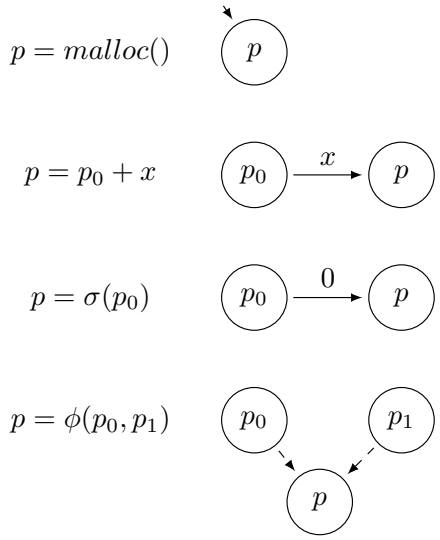


Figure II.3.5 – Rules to generate the pointer dependence digraph.

When analyzing a program, we build as many PDDs as the number of pointer definition sites in the program. Notice that this number is a *static* concept. A memory allocation site within a loop still gives us only one allocation site, even if the loop iterates many times. Such a data-structure is constructed according to the rules in Figure II.3.5, during a traversal of the program control flow graph. Notice that we have a special treatment for ϕ-functions. A ϕ-function is a special instruction used in the SSA format to join the live ranges of variables that represent the same name in the original program before the SSA transformation (details about the SSA form were presented in Section I.1.2). We mark nodes created by ϕ-functions as dashed edges in the PDD. In this way, we ensure that a PDD has no cycles, because in an SSA-form program, the only way a variable can update itself is through a ϕ-function.

Example 16 *The rules seen in Figure II.3.5, once applied onto the control flow graph given in Figure II.3.4, give us the PDD shown in Figure II.3.6.*

II.3.4 A Constraint-Based Analysis

We solve the less-than analysis via a constraint system. This constraint system is formed by five different kinds of constraints, which involve two variables, and a relational operator. Valid relational operators are $\mathscr{R} \in \{=, <, \leq, \geq, >\}$, each one giving birth to one of the five types of constraints. If *Prog* is a program formed according to the syntax seen in Figure II.3.3, and $\{v_1, v_2\} \in Prog$ are two variables, then we let $v_1 \mathscr{R} v_2$ be a constraint. We use the same names to denote program variables and constraint variables. Program variables are elements that exist

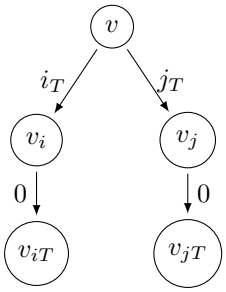


Figure II.3.6 – Pointer dependence graph of program in Figure II.3.1.

in the syntax of *Prog*; constraint variables are symbols, which, as we shall see in Section II.3.4.2, are mapped to "less-than" information. Because it is always possible to know which entity we refer to, given the context of the discussion, we shall not use any notation to distinguish them.

To explain the semantics of constraints, we describe the semantics of programs. A program *Prog* is formed by instructions, which operate on states $\sigma : Var \mapsto Int$. We say that a variable v is *defined* at a state σ, if $\sigma(v) = n$; otherwise, the variable is undefined at that state, a fact that we denote by $\sigma(v) = \perp$. Constraints determine relations between defined variables, given a state σ. Definition 11 expresses this notion.

Definition 11 (Intuitive Semantics) *We say that a constraint $v_1 \mathscr{R} v_2$ satisfies a state σ if $\sigma(v_1) \mathscr{R} \sigma(v_2)$ whenever v_1 and v_2 are defined for σ.*

II.3.4.1 Collecting constraints

During this phase, we collect a set $\mathscr{C}$ of constraints according to the rules in Figure II.3.7. Recall that R(v) denotes the numeric range of variable v given by a pre-analysis. We let $R(\mathtt{v})_\downarrow$ and $R(\mathtt{v})_\uparrow$ be the lower and upper bounds of interval R(v), respectively. All the constraints that we produce in this stage follow the template $p_1 \mathscr{R} p_2$, where $\mathscr{R} \in \{<, \leq, =\}$. The rules in Figure II.3.7 are syntax-directed. For instance, an assignment $\mathtt{p} = \mathtt{q} + \mathtt{v}$ lets us derive the fact $\mathtt{p} > \mathtt{q}$, if the range analysis of Section II.3.2 is capable of proving that v is always strictly greater than zero. The "=" equality models set inclusion, and is asymmetric as we shall explain in details in Section II.3.4.2.

Constraints are simple, and the reader will understand their meaning from their syntax; however, a few remarks are in order. Firstly, the difference between *add* and *gep*[2] is the fact that the latter represents an addition on a pointer p plus an offset v_1, whereas the former represents general integer arithmetics. We distinguish both because the C standard forbids any arithmetic operation on pointers other than adding or subtracting an integer to it. Therefore, to avoid comparing a pointer to an integer variable, the constraints that we produce for arithmetic operations involving pointers are different than those we produce for similar instructions that involve only integers, as Figure II.3.7 shows.

A second aspect of our constraint system, which is worth mentioning, is the fact that we do not try to track less-than information throughout pointers. Hence, as Figure II.3.7 shows, we do not generate constraints for loads and stores. In other words, we do not build less-than relations for data stored in memory. Consequently, our current implementation of the less-than lattice misses opportunities to disambiguate *second-order* pointers, i.e., pointers to pointers. This omission is not a limitation of the theory that we present in this chapter. Rather, it is an implementation

[2]The name *gep* is a short form for *get element pointer*, the expression used to define a new pointer address in the LLVM compiler.

$$
\begin{array}{rcl}
\textit{Initialization} & \Rightarrow & \mathscr{C} = \emptyset \\
\\
\textit{gep} : q_1 = p + v_1 & \Rightarrow & \begin{cases} \text{if } R(v_1)_{\downarrow} > 0 \ \mathscr{C} \cup = \{p < q_1\} \\ \text{if } R(v_1)_{\downarrow} \geqslant 0 \ \mathscr{C} \cup = \{p \leqslant q_1\} \\ \text{if } R(v_1)_{\uparrow} < 0 \ \mathscr{C} \cup = \{q_1 < p\} \\ \text{if } R(v_1)_{\uparrow} \leqslant 0 \ \mathscr{C} \cup = \{q_1 \leqslant p\} \end{cases} \\
\\
\begin{array}{r} \textit{add} : v = v_1 + v_2 \\ \text{similar for } v \text{ and } v_2 \\ \text{with } v = v_2 + v_1 \end{array} & \Rightarrow & \begin{cases} \text{if } R(v_2)_{\downarrow} > 0 \ \mathscr{C} \cup = \{v_1 < v\} \\ \text{if } R(v_2)_{\downarrow} \geqslant 0 \ \mathscr{C} \cup = \{v_1 \leqslant v\} \\ \text{if } R(v_2)_{\uparrow} < 0 \ \mathscr{C} \cup = \{v < v_1\} \\ \text{if } R(v_2)_{\uparrow} \leqslant 0 \ \mathscr{C} \cup = \{v \leqslant v_1\} \end{cases} \\
\\
\textit{sub} : v = v_1 - v_2 & \Rightarrow & \begin{cases} \text{if } R(v_2)_{\downarrow} > 0 \ \mathscr{C} \cup = \{v < v_1\} \\ \text{if } R(v_2)_{\downarrow} \geqslant 0 \ \mathscr{C} \cup = \{v \leqslant v_1\} \\ \text{if } R(v_2)_{\uparrow} < 0 \ \mathscr{C} \cup = \{v_1 < v\} \\ \text{if } R(v_2)_{\uparrow} \leqslant 0 \ \mathscr{C} \cup = \{v_1 \leqslant v\} \end{cases} \\
\\
\textit{icmp} : p^1 \ \mathscr{R} \ p^2 & \Rightarrow & \begin{cases} \text{if } \mathscr{R} = \text{“}<\text{”}, \text{ then } \mathscr{C} \cup = \{p_1^{\mathrm{T}} < p_2^{\mathrm{T}}\} \cup \{p_2^{\mathrm{F}} \leqslant p_1^{\mathrm{F}}\} \\ \text{if } \mathscr{R} = \text{“}\leqslant\text{”}, \text{ then } \mathscr{C} \cup = \{p_1^{\mathrm{T}} \leqslant p_2^{\mathrm{T}}\} \cup \{p_2^{\mathrm{F}} < p_1^{\mathrm{F}}\} \\ \mathscr{C} \cup = \{p_1^{\mathrm{T}} = p^1\} \cup \{p_1^{\mathrm{F}} = p^1\} \cup \{p_2^{\mathrm{T}} = p^2\} \cup \{p_2^{\mathrm{F}} = p^2\} \end{cases} \\
\\
\textit{union} : v = \phi(v_i) & \Rightarrow & \mathscr{C} \cup = \{v = \phi(v_i)\} \\
\\
\begin{array}{r} v = c - v_1, c \in \mathbb{N} \\ q = *p \\ *q = p \end{array} & \Rightarrow & \textit{nothing}
\end{array}
$$

Figure II.3.7 – Constraints produced for different statements in our language. The notation $\mathscr{C} \cup = S$ is a shorthand for $\mathscr{C} = C \cup S$. v_1 and v_2 are constants or scalar variables.

decision, which we took to simplify the design of our algorithm. Handling pointers to pointers is possible in different ways. For instance, we can use some pre-analysis, à la Andersen [And94] or à la Steensgaard [Ste96], to group memory references into single locations. After this bootstrapping phase, we can treat these memory locations as variables, and extract constraints for them using the rules in Figure II.3.7.

The construction of constraints for tests is more involved. The e-SSA form, which we have discussed in Section I.1.2.1, provides us explicit new versions of variables, e.g.: $\mathtt{p_T}, \mathtt{q_T}, \mathtt{p_F}, \mathtt{q_F}$, after a test such as $\mathtt{p} < \mathtt{q}$. We use T as a subscript for the new variable name created at the *true* branch; F has similar use for the *false* branch. Thus, the conditional test $\mathtt{p} < \mathtt{q}$ for instance gives us two constraints: $\mathtt{p_T} < \mathtt{q_T}$ and $\mathtt{p_F} \geq \mathtt{q_F}$.

Example 17 *The rules in Figure II.3.7, when applied onto the cfg seen in Figure II.3.4, give us that* $\mathscr{C} = \{N' < N,\ i = \phi(i_0, i_1),\ i_T < N'_T,\ N'_F \leq i_F,\ i_T < j_0,\ j = \phi(j_0, j_1),\ i_T = i,\ j_T = j,\ j_T < N_T,\ N_F \leq j_F,\ v \leq v_i,\ v < v_j,\ v \leq v'_i,\ v < v'_j,\ j_T < j_1,\ i_T < i_1,\ x_{jT} < x_{iT},\ x_{iF} \leq x_{jF}\}$.

The Relation between PDDs and Constraints. The purpose of the PDDs of Section II.3.3 is to avoid comparing pointers that are not related by C-style arithmetics. They do not bear influence on the production of constraints; rather, they are used only in the queries that we shall describe in Section II.3.4.2. This fact means that the grouping of pointers in digraphs is not an essential part of the idea of using strict inequalities to disambiguate pointers. However, PDDs

are important from an operational standpoint: they provide a way of separating pointers that are not related by arithmetic operations; hence, are incomparable. Thus, the phases described in Section II.3.3 and in this section are independent, and can be performed in any order.

II.3.4.2 Solving constraints

The objective of this phase is to obtain Pentagons-like abstract values for each variable or pointer of the target program. Henceforth, we shall call the set of program variables $\mathcal{V}$. The product of solving constraints is a relation LT. Given $\mathtt{v} \in \mathrm{V}$, $\mathsf{LT}(\mathtt{v})$ will keep track of all the variables that are *strictly* less than $\mathtt{v}$. In addition to LT, we build an auxiliary set GT. $\mathsf{GT}(\mathtt{v})$ keeps track of all the variables that are *strictly* greater than $\mathtt{v}$. Ordering relations are reflexive; hence, if $\mathtt{x} < \mathtt{y}$, then $\mathtt{y} > \mathtt{x}$. This fact means that we can build GT from LT, or vice-versa; hence, we could write our solver with only one of these relations. However, using just one of them would result in an unnecessarily heavy notation. Consequently, throughout the rest of this section we shall assume that the following two equations are always true:

$$\mathsf{GT}(v) = \bigcup_{v_i \ s.t. \ v \in \mathsf{LT}(v_i)} \{v_i\}$$

$$\mathsf{LT}(v) = \bigcup_{v_i \ s.t. \ v \in \mathsf{GT}(v_i)} \{v_i\}$$

Figures II.3.8 and II.3.10 contain the definition of our constraint solver. Constraints are solved via *Chaotic Iterations* [NNH05, p-176]: we compute for each program variable a sequence of abstract values until reaching a fixpoint. The non-reflexive equal is used to model relations that are known to be true before sigma nodes. Relational information that are true about variables before a conditional branch continue to hold also in the "then" and "else" paths that sprout from that branch. Example 18 illustrates this fact:

$$\textit{strict less than} : x < y \quad \Rightarrow \quad \begin{cases} \mathsf{LT}(y) \cup = \mathsf{LT}(x) \cup \{x\} \\ \mathsf{GT}(x) \cup = \mathsf{GT}(y) \cup \{y\} \end{cases}$$

$$\textit{less than} : x \leqslant y \quad \Rightarrow \quad \begin{cases} \mathsf{LT}(y) \cup = \mathsf{LT}(x) \\ \mathsf{GT}(x) \cup = \mathsf{GT}(y) \end{cases}$$

$$\textit{non reflexive eq} : x = y \quad \Rightarrow \quad \begin{cases} \mathsf{LT}(x) \cup = \mathsf{LT}(y) \\ \mathsf{GT}(x) \cup = \mathsf{GT}(y) \end{cases}$$

Figure II.3.8 – Rules to solve constraints.

Example 18 *If the relation* $(\mathtt{x} < \mathtt{y})$ *holds before a conditional node that uses the predicate* $(\mathtt{x} < \mathtt{z}?)$*, then these two relations are also true:* $(\mathtt{x_T} < \mathtt{y}$ *and* $\mathtt{x_F} < \mathtt{y})$*, where* $\mathtt{x_T}$ *is the new name of* $\mathtt{x}$ *in the "then" branch of the conditional, and* $\mathtt{x_F}$ *is the new name of* $\mathtt{x}$ *in the "else" branch.*

Dealing with ϕ-functions. *Join nodes* are program points that denote loops and conditional branches. In SSA-form programs, these nodes are created by ϕ-functions. These instructions

give us special constraints, as Figure II.3.7 shows. A ϕ-function such as $v = \phi(v_1, v_2)$ yields the constraint $\{v = \phi(v_i)\}, 1 \leq i \leq 2$. When used in the head of loops, ϕ-functions may join variables that represent initialization values and loop variant values for a given variable v. To build a less-than relation between these variables, we must find out how the value of v evolves during the execution of the program. It might increase, it might decrease, or it might oscillate. If v only decreases, then it will be less than or equal to the maximum among its initialization values. In the opposite case, in which v only increases, the variable produced by the ϕ-function will be greater than or equal to the minimum among its initialization values.

To infer relations between the variable defined by a ϕ-function, and the variables used as arguments of that ϕ-function, we perform a *growth check*. On the resolution of each join constraint such as $x = \phi(x_1, \ldots, x_n)$, we check if x is present in the LT or GT sets of some x_i which is loop variant. If x is present in the LT of each loop-variant right-hand side operand, then the program contains only execution paths in which x grows. That is to say that x is always receiving a value that is greater than itself. Dually, if x is present in the GT of each loop-variant x_i, then there exist only paths in the program along which x decreases. And, how do you know that a certain x_i is loop-variant? We use dominance information: if the definition of a ϕ-function, e.g., x, dominates the definition point of one of its arguments, e.g., x_i, then we know that this ϕ-function occurs in a loop, and we say that x_i is loop-variant. This observation holds for the so called *natural loops*, which characterizes *reducible control flow graphs*. For the definition of these concepts, we refer the reader to the work of Ferrante *et al* [FOW87]. Example 19 illustrates these observations.

Example 19 *Variable x in Figure II.3.9 (a) increases along the execution of the program. Differently, variable x in Figure II.3.9 (b) oscillates: it might increase or decrease, depending on the path along which the program flows. Both ϕ-functions exist in natural loops.*

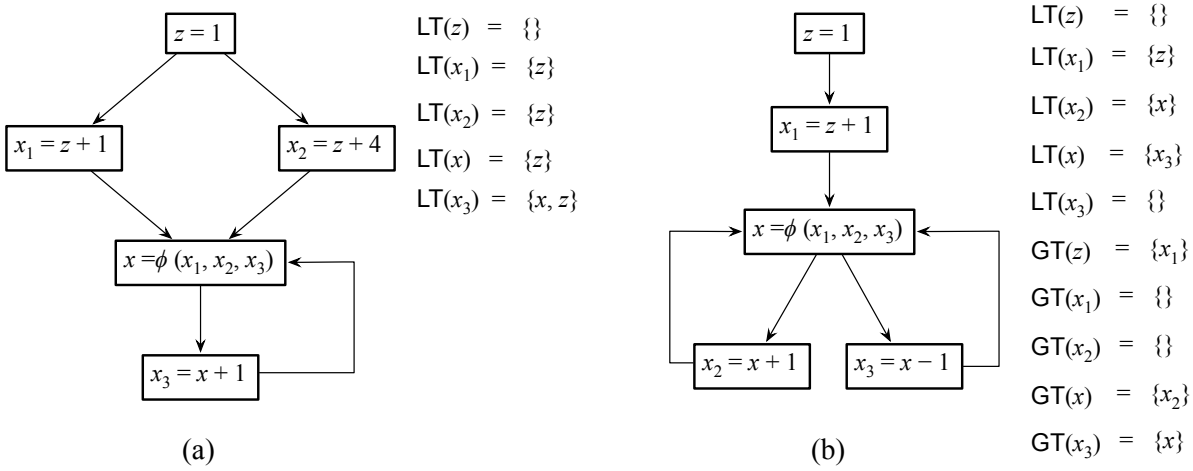


Figure II.3.9 – Less-than and greater-than sets built by our analysis. In II.3.9(a) we can prove that $z \in \mathsf{LT}(x)$ since x grows up only within the loop. In Figure II.3.9(b) x oscillates: no information about x and z is derived.

Figure II.3.10 shows how we handle constraints that exist due to ϕ-functions. We use *conditional constraints* to deal with them. A conditional constraint is formed by a *trigger* T and an *action* a, denoted by the notation $T \Rightarrow a$. The action a is evaluated only if the trigger T is true. The constraint $x = \phi(x_i)$ has three different triggers, and at any moment, at least one of them is true. Triggers check if particular LT and GT sets contain specific variables. Example 20 explains how we compute the less-than set of x.

$$join : x = \phi(x_i) \\ 0 \leqslant i \leqslant n \Rightarrow \begin{cases} i : \forall x_j \in Dom(x),\ x' \in \mathsf{LT}(x_j) \Rightarrow \mathsf{LT}(x) \cup = \bigcap\limits_{x_j \notin Dom(x)} \mathsf{LT}(x_j) \\ ii : \forall x_j \in Dom(x), x' \in \mathsf{GT}(x_j) \Rightarrow \mathsf{GT}(x) \cup = \bigcap\limits_{x_j \notin Dom(x)} \mathsf{GT}(x_j) \\ iii : \text{Otherwise} \Rightarrow \begin{cases} \mathsf{LT}(x) \cup = \bigcap\limits_{i=0}^{n} \mathsf{LT}(x_i) \\ \mathsf{GT}(x) \cup = \bigcap\limits_{i=0}^{n} \mathsf{GT}(x_i) \end{cases} \end{cases}$$

Figure II.3.10 – Solving constraints for ϕ-functions. We let $x' \in \{x, x_T, x_F\}$ and $Dom(x) = \{x_i \text{ s.t. } x \text{ dominates } x_i\}$, $A \equiv \forall x_j \in Dom(x),\ x' \in \mathsf{LT}(x_j)$ and $B \equiv \forall x_j \in Dom(x), x' \in \mathsf{GT}(x_j)$. A indicates that x increases in the loop, and B indicates that x decreases. Variables x_T and x_F are new names of x created by σ-functions after conditional branches.

Example 20 *Variable x, defined in Figure II.3.9 (a) has increasing value, because it belongs into $LT(x_3)$ and $Dom(x) = \{x_3\}$. Therefore, trigger i, in Figure II.3.10 applies, and we know that $LT(x) = LT(x) \cup (LT(x_1) \cap LT(x_2))$. Thus, $LT(x) = LT(x) \cup (\{z\} \cap \{z\}) = \{z\}$.*

Implementation of the Constraint Solver. We solve the constraint set $\mathscr{C}$ via the method of Chaotic Iterations. We repeatedly insert constraints into a worklist W, and solve them in the order they are inserted. The evaluation of a constraint might lead to the insertion of other constraints into W. This insertion is guided by a *Constraint Dependence Graph* (CDG). Each vertex v of the CDG represents a constraint, and we have an edge from v_1 to v_2 if, and only if, the constraint represented by v_2 reads a set produced by the constraint represented by v_1. Upon popping a constraint such as $x < y$, we verify if its resolution changes the current state of LT sets. If it does, then we insert back into the worklist every constraint that depends on y, i.e., that reads $\mathsf{LT}(y)$. This algorithm has asymptotic complexity $\mathcal{O}(n^3)$. However, in practice our implementation runs in time linear on the number of constraints, as we show empirically in Section II.3.6. The process of constraint resolution is guaranteed to terminate, as Theorem 4 proves. Example 21 illustrates this approach.

Example 21 *We let $\mathscr{C} = \{x < y, y < z\}$. The result of solving $\mathscr{C}$ to build LT sets is given in Figure II.3.11. Our worklist is initialized to the constraint set $\mathscr{C}$. Each variable v is bound to an abstract state $\mathsf{LT}(v) = \emptyset$ and $\mathsf{GT}(v) = \emptyset$. Resolution reaches a fixpoint when the worklist is empty. In each iteration, a constraint is popped, and abstract states are updated following the rules in Figures II.3.8 and II.3.10. We also update the worklist according to the constraint dependence graph.*

Theorem 4 (Termination) *Constraint resolution is guaranteed to terminate.*

Proof: The worklist based solver reaches a fixpoint, because of two reasons. First, the updating of abstract states is monotonic. The inspection of Figures II.3.8 and II.3.10 shows that the abstract state of variable v, e.g., $\mathsf{LT}(v)$ and $\mathsf{GT}(v)$, is only updated via union with itself plus extra information, if available. Second, abstract states are represented by points in a lattice of finite height: at most the less-than or greater-than set of a variable will contain all the other variables in the program. □

WL_i	$pop(WL_i)$	$\mathsf{LT}(x)$	$\mathsf{LT}(y)$	$\mathsf{LT}(z)$
$\{x<y, y<z\}$	–	$\emptyset$	$\emptyset$	$\emptyset$
$\{x<y, y<z\}$	$x<y$	$\emptyset$	$\{x\}$	$\emptyset$
$\{y<z, y<z\}$	$y<z$	$\emptyset$	$\{x\}$	$\{y,x\}$
$\{y<z\}$	$y<z$	$\emptyset$	$\{x\}$	$\{y,x\}$
$\{\}$	–	$\emptyset$	$\{x\}$	$\{y,x\}$

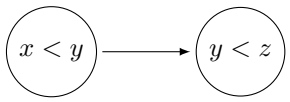


Figure II.3.11 – Resolution of the constraints produced in Example 21. The order in which we solve constraints is dictated by their dependences in the CDG, which we show on the right. Each line depicts one step of the algorithm.

Example 22 *Figure II.3.4 gives us the following constraint system:* $\mathscr{C} = \{N' < N,\ i = \phi(i_0, i_1),\ i_T < N'_T,\ N'_F \leq i_F;\ i_T < j_0,\ j = \phi(j_0, j_1),\ i_T = i,\ j_T = j,\ j_T < N_T,\ N_F \leq j_F,\ v \leq v_i,\ v < v_j,\ v \leq v'_i,\ v < v'_j,\ j_T < j_1,\ i_T < i_1,\ x_{jT} < x_{iT},\ x_{iF} \leq x_{jF}\}$. *The solution that we find for the* LT *sets is the following:* $\mathsf{LT}(i_0) = \mathsf{LT}(N') = \mathsf{LT}(j_F) = \mathsf{LT}(i_F) = \mathsf{LT}(N'_F) = \mathsf{LT}(x_{jT}) = \mathsf{LT}(x_{iF}) = \mathsf{LT}(x_{jF}) = \mathsf{LT}(i) = \emptyset$, $\mathsf{LT}(i_1) = \{i_T\}$, $\mathsf{LT}(N) = \{N'\}$, $\mathsf{LT}(j_0) = \{i_T, i_0\}$, $\mathsf{LT}(j_1) = \{j_T, j_0, i_T\}$, $\mathsf{LT}(i_T) = \{i_0\}$, $\mathsf{LT}(j) = \{i_T\}$, $\mathsf{LT}(j_T) = \{j_0, i_T\}$, $\mathsf{LT}(N'_T) = \{i_T, i_0\}$, $\mathsf{LT}(x_{iT}) = \{x_{jT}\}$, $\mathsf{LT}(v_j) = \{v\}$, $\mathsf{LT}(v'_j) = \{v\}$.

The Semantics of Constraints. The *concrete state* of a program variable $\mathtt{v}$ in the e-SSA form is the set of values that $\mathtt{v}$ can receive throughout the execution of a program. The *abstract state* of a variable $\mathtt{v}$ is $\mathsf{LT}(\mathtt{v})$. If $\mathtt{y} \in \mathsf{LT}(\mathtt{v})$, then we know that $\mathtt{y}$ is strictly less than $\mathtt{v}$ at every program point where $\mathtt{y}$ and $\mathtt{v}$ are simultaneously alive. In other words, it is still possible that $\mathtt{y} \in \mathsf{LT}(\mathtt{v})$, but $\mathtt{y} \not< \mathtt{v}$ if we look into the concrete values of these variables at different moments during the execution of the same program. Theorem 5 states the core property that our constraint system delivers.

Valid Solutions. We say that LT is a valid solution for a constraint system $\mathscr{C}$, which models a program *Prog* if it meets the requirements in Definition 12. Program states, i.e., σ, are defined in Section II.3.4.

Definition 12 (Valid Solution) *We say that* $\mathsf{LT} \models \sigma$ *if, for any* v_1 *and* v_2*, well-defined at* σ*, we have that:* $\mathsf{LT} \models \sigma \wedge v_1 \in \mathsf{LT}(v_2) \Rightarrow \sigma(v_1) < \sigma(v_2)$.

Theorem 5 states that the solution that we find for a constraint system is valid. The proof of the theorem relies on the semantics of instructions. Each instruction I transforms a state σ into a new state σ'. We shall not provide transition rules for these instructions, because they have an obvious semantics, which has been described elsewhere [Naz+14, Fig.3].

Theorem 5 (Correctness of the Strict Less-Than Relations) *Our algorithm produces valid solutions to the constraint system.*

Proof: The proof of Theorem 5 consists in a case analysis on the different constraints that can create the relation $v_1 \in \mathsf{LT}(v_2)$ in Figure II.3.8 and Figure II.3.10. We go over constraints build in these figures to prove that if $v_1 \in \mathsf{LT}(v_2)$ then $v_1 < v_2$. The proof goes by induction on the size of LT; if LT is empty, the property is trivially verified.

- The constraint *Strict less than* $x < v_2$ updates the strict less than set of v_2. $\mathsf{LT}(v_2) = \mathsf{LT}(v_2) \cup \mathsf{LT}(x) \cup \{x\}$. For the sake of clarity, we let $\mathsf{LT}'(v_2)$ be the strict less than set of v_2 before the update introduced by the constraint $x < v_2$. Henceforth, $v_1 \in \mathsf{LT}(v_2)$

if $v_1 \in \mathsf{LT}'(v_2)$ or $v_1 \in \mathsf{LT}(x)$, or $v_1 = x$. If $v_1 = x$, then we are done, due to the constraint $x < v_2$. Otherwise, if $v_1 \in \mathsf{LT}(x)$, by induction, $v_1 < x$, and by transitivity we get $v_1 < v_2$. In case $v_1 \in \mathsf{LT}'(v_2)$, by induction on $\mathsf{LT}'(v_2), v_1 < v_2$. In fact, since the strict less than set of v_2 is only growing up if updated, then if a property holds for its members once, it holds even after updates. In other words, $\mathsf{LT}'(v_2)$ may be an update of $\mathsf{LT}''(v_2)$ in which we have inserted v_1 after resolving a constraint. In this case, $\mathsf{LT}''(v_2) \subseteq \mathsf{LT}'(v_2) \subseteq \mathsf{LT}(v_2)$ with $v_1 \in \mathsf{LT}''(v_2)$. In the remaining of the proof, we shall be interested only on elements updating $\mathsf{LT}(v_2)$.

- The constraint *less than* straightforward based on the *Strict less than* constraint proof.
- The constraint *non reflexive equal* $v_2 = x$, updates $\mathsf{LT}(v_2)$ with $\mathsf{LT}(x)$: $\mathsf{LT}(v_2) = \mathsf{LT}(v_2) \cup \mathsf{LT}(x)$. We focus on $v_1 \in \mathsf{LT}(x)$. By induction, we get $v_1 < x$. From Figure II.3.7 we know that the constraint comes from a test. Thus, without loss of generality, we can assume that this constraint is $x_T = x$ (x_T is v_2), coming from a test $x < y$. The condition $v_1 < x$ is true before the test. Because the e-SSA conversion does not change the semantics of the target program, the condition is still true after renaming the operands of the test, thus $v_1 < x_T$ and therefore $v_1 < v_2$.
- The constraint $x = \phi(x_i)$. We show the proof for the first case of Figure II.3.10, the second one is similar and the third one is straightforward. In this case we have $\forall x_j \in Dom(x),\ x' \in \mathsf{LT}(x_j)$ implying $\mathsf{LT}(x) \cup = \bigcap_{x_j \notin Dom(x)} \mathsf{LT}(x_j)$ where $x' \in \{x, x_T, x_F\}$. In the general case we would have: $\mathsf{LT}(x) \cup = \bigcap_{i=0}^{n} \mathsf{LT}(x_i)$ that we can write

$$\mathsf{LT}(x) \cup = \left(\bigcap_{x_j \notin Dom(x)} \mathsf{LT}(x_j)\right) \bigcup \left(\bigcap_{x_j \in Dom(x)} \mathsf{LT}(x_j)\right)$$

since x may have any of x_i values. We want to prove that if $y \in \bigcap_{x_j \notin Dom(x)} \mathsf{LT}(x_j)$, then $y < x$ under condition A. We let $x_{j_0} \in Dom(x)$ such that $x' \in \mathsf{LT}(x_{j_0})$. This means that x_{j_0} is defined as follows: $x_{j_0} = x' + a; a > 0$. Hence, there exists a redefinition x'_i of x among ϕ operands such that $\langle x = x'_i \rangle$ and $x_{j_0} = x'_i + a$. Since $a > 0$ we obtain $x'_i < x_{j_0}$ with two possible situations for x'_i:
  - $x'_i \in Dom(x)$, then $x' \in \mathsf{LT}(x'_i)$ which gives again $\exists x''_i$ such that $x''_i < x'_i < x_{j_0}$.
  - $x'_i \notin Dom(x)$, then $y \in \mathsf{LT}(x'_i)$. As there cannot be any infinite descending sequence of ϕ redefinitions, all instances of variable x are greater than x'_i. By induction on LT, $y \in \mathsf{LT}(x'_i)$ gives $y < x_i$ and therefore $y < x$.

II.3.5 Answering Alias Queries

The final product of the techniques discussed in the previous section is a table that associates each variable v with a set $\mathsf{LT}(v)$ of other variables that are less than v. This table gives us the means to prove that some pairs of pointers cannot dereference overlapping memory regions. We use three different tests to show that pointers cannot alias each other. The process of applying such tests on pairs of pointers shall be, henceforth, called a *query*. Figure II.3.12 illustrates the relationship between the three tests that we use to answer queries. If we cannot conclude that two pointers always refer to distinct regions, then we say – conservatively – that they *may alias*. Currently, we do not use our infra-structure to prove that two pointers *must alias* each other.

The three pointer disambiguation checks that Figure II.3.12 outlines are complementary and definitive. By complementary, we mean that none of them subsumes the others. By definitive, we mean that they all are conservatively safe: if any of these checks reports that pointers p_1 and p_2 do not alias, then this information, regardless of the results produced by the other checks, is enough to ensure that p_1 and p_2 refer to different locations. In the rest of this section, we provide further details about the three pointer disambiguation tests that we use. Briefly, we summarize them as follows, assuming that we want to disambiguate pointers p_1 and p_2:

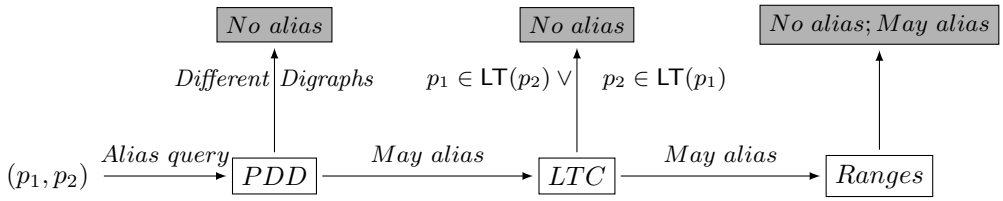


Figure II.3.12 – Steps used in the resolution of pointer disambiguation queries.

PDD: If p_1 and p_2 belong to different pointer dependence digraphs, then they are said to be unrelated. PDDs are described in Section II.3.3.

Less-Than: We consider two disambiguation criteria:

1. If $p_1 \in \mathsf{LT}(p_2)$, or vice-versa, then these pointers cannot point to overlapping memory regions.
2. Memory locations $p_1 = p + x_1$ and $p_2 = p + x_2$ will not alias if $x_1 \in \mathsf{LT}(x_2)$ or $x_2 \in \mathsf{LT}(x_1)$.

Ranges: If we can reconstruct the ranges covered by p_1 and p_2, and these ranges do not overlap, then p_1 and p_2 cannot alias each other.

II.3.5.1 The digraph test

We have seen, in Section II.3.3, how to group pointers that are related by C-style pointer arithmetics into digraphs. Pointers that belong to the same PDD can dereference overlapping memory regions. However, pointers that are in different digraphs cannot alias, because they reference different memory allocation blocks. Memory blocks are different if they have been allocated at different program sites. Example 23 illustrates this observation.

Example 23 *The program in Figure II.3.13 contains two different memory allocation sites. The first is due to the argument* argv*. The second is due to the* malloc *operation that initializes pointer* s1*. These two different locations will give origin to two different pointer dependence digraphs, as we show in the figure.*

```
1  int main(int argc, char** argv) {
2    int n = strlen(argv[1]);
3    char* s1 = (char*) malloc(n);
4    char* s2 = argv[1];
5    char* s3 = s1;
6    char* s4 = s2 + n;
7    while (s2 < s4) {
8      *s3 = *s2;
9      s2++;
10     s3++;
11   }
12 }
```

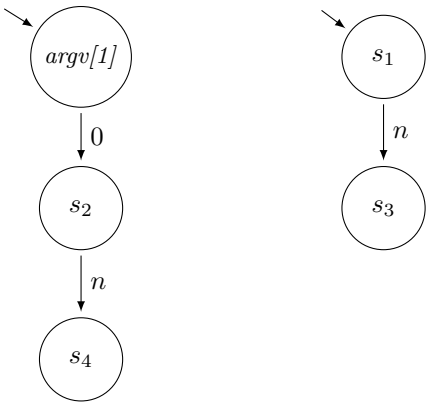


Figure II.3.13 – Program that contains two different memory locations, which will give origin to two unconnected pointer dependence digraphs. For the sake of brevity, the program is kept in high level language.

As we have mentioned in Section II.3.4.1, we do not track abstract information that flows through memory locations. In other words, we do not generate constraints for information that comes out of load and store instructions. Similarly, our current implementation of the Digraph Test also does not consider pointers to pointers when building PDDs. Thus, our PDDs contain vertices that represent only top-level variables, i.e., variables that our baseline compiler, LLVM, represents in Static Single Assignment form. The consequence of this implementation decision is that currently we do not disambiguate pointers `p1` and `p2` in the following code snippet: `{p1 = v + 4; *x = v; p2 = *x + 8;}`, because we forgot the fact that `v` and `*x` represent the same memory location.

II.3.5.2 The less-than test

The less-than test relies on the relations constructed in Section II.3.4.2 to disambiguate pointers. This test is only applied onto pointers that belong to the same pointer dependence digraph. From Theorem 5, we know that if $p_1 \in \mathsf{LT}(p_2)$, then $p_1 < p_2$ whenever these two variables exist in the program. Along similar lines, we have $p_1 < p_2$ if $x < y$ with $p_1 = p + x$ and $p_2 = p + y$. Therefore, they cannot dereference aliasing locations. Example 24 illustrates the application of this test.

Example 24 *We want to show that locations* `v[i]` *and* `v[j]` *in Figure II.3.1 cannot alias each other. The cfg of function* `ins_sort` *appears in Figure II.3.4. In the cfg's low-level representation,* `v[i]` *corresponds to* $\mathtt{v_i}$*, and* `v[j]` *corresponds to* $\mathtt{v_j}$*. We know that* $\mathsf{LT}(\mathtt{j_T}) = \{\mathtt{j_0}, \mathtt{i_T}\}$ *(as seen in Example 22) and* $\mathtt{v_i} = \mathtt{v} + \mathtt{i_T}$ *and* $\mathtt{v_j} = \mathtt{v} + \mathtt{j_T}$*. Therefore, we can conclude that these two pointers* $\mathtt{v_i}$ *and* $\mathtt{v_j}$ *do not alias.*

II.3.5.3 The ranges test

The so called *Ranges test* is a byproduct of the key components of the previous PDD test. This test consists in determining an expression of the range of intervals covered by two pointers, p_1 and p_2, that share the same pointer digraph. If these two ranges do not intersect, then we can conclude that p_1 and p_2 do not alias. Algorithmically, this test proceeds as follows:

1. Find the *closest common ancestor* p_a of p_1 and p_2. We say that p_a is the *closest common ancestor* of these pointers if, and only if, (i) it is an ancestor, i.e., dominates both p_1 and p_2; and (ii) for any other pointer $p' \neq p_a$ that dominates p_1 and p_2, we have that p' dominates p_a.
2. Rewrite p_1 and p_2 as functions of p_a. To this end, we repeat the following re-writing rules:
   (a) If $p_x \rightarrow_e p_i, i \in \{1, 2\}$ in the pointer dependence digraph, then we replace p_i by $p_x + e$.
   (b) If $p_x \neq p_a$, then repeat step 2a.
3. Let $p_a + e_1$ and $p_a + e_2$ be the final expressions that we obtain for pointers p_1 and p_2. If $R(e_1) \cap R(e_2) = \emptyset$, then we report that p_1 and p_2 do not alias. Otherwise, we report that these pointers may alias.

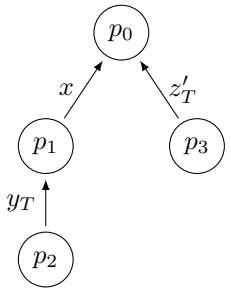
Step 2 above is collapsing a path $\mathcal{P}(p_i, p_a) = (p_i, \ldots, p_a)$ in the pointer digraph into a single edge $p_i \rightarrow p_a$. This technique relies on the same ideas than the ones we introduced in Chapter II.2 to disambiguate pointers: if two pointers cover non-overlapping ranges, then they cannot alias. However, in terms of implementation, we use range analysis on the integer interval lattice in this work. In Chapter II.2 we used a symbolic range analysis. There is no theoretical limitation that prevents us from using a symbolic lattice to reuse our previous algorithm test. We have opted to use the simpler interval lattice because it is already available in LLVM, the compiler that we have used to implement our alias analysis. This also allows us to distinguish between

the addition and subtraction instruction in the constraint generation process of Figure II.3.7. Example 25 shows how the range test works concretely.

Example 25 *Consider the program in Figure II.3.14a. Our goal in this example is to disambiguate pointers* p2 *and* p3*. We have that* $LT(\texttt{p0}) = \emptyset$, $LT(\texttt{p1}) = \{\texttt{p0}\}$, $LT(\texttt{p2}) = \{\texttt{p0}, \texttt{p1}\}$ *and* $LT(\texttt{p3}) = \{\texttt{p0}\}$. $\texttt{p}_2 \notin LT(\texttt{p3})$ *and* $\texttt{p}_3 \notin LT(\texttt{p2})$*. The simple less than check is not able to disambiguate these pointers; hence, we resort to the range test. The dependence graph of Figure II.3.14a reveals that the lowest common ancestor of* p3 *and* p2 *is* p0*.* $\mathcal{P}(p2, p0) = (p2, p1, p0)$ *and* $\mathcal{P}(p3, p0) = (p3, p0)$*. Our re-writing algorithm gives us that* $\texttt{p2} = \texttt{p0} + \texttt{y} + \texttt{x}$*, and* $\texttt{p3} = \texttt{p0} + \texttt{z}$*. Using range information, we get that:* $\texttt{R}(\texttt{y} + \texttt{x}) = \texttt{R}(\texttt{y}) + \texttt{R}(\texttt{x}) = [3, 4]$ *and* $\texttt{R}(\texttt{z}) = [5, 7]$*. These ranges do not intersect; therefore,* p2 *and* p3 *do not alias.*

```
1  void pdd_ex(int N){
2    if (N>0){
3      int *p0=malloc(N);
4      int x = 1;
5      int y = (rand()%2) + 2;
6      int z= (rand()%3) + 5;
7      int* p1=p0+x;
8      int* p2=p1+y;
9      int* p3=p0+z;
10   }
11 }
```

(a) A program.



(b) Its Pointer Dependence Digraph.

Figure II.3.14 – Disambiguating pointers with PDD and Ranges.

The impact of range analysis on the precision of our method. In Section II.3.2 we emphasized that the more precise the range analysis for integer variables that we use, the more precise the pointer analysis that we produce. Let us now detail how the difference in precision affects our analysis of pointers. In Figure II.3.15, we give a simple program where our goal is to disambiguate memory locations $p[i]$ and $p[i+1]$. Suppose that we have renamed variable i to i_0 at line ℓ_3 and to i_1 at ℓ_4 and memory locations $p[i]$ and $p[i+1]$ to respectively p_0 and p_1. We have $\mathcal{P}(p_0, p) = (p_0, p)$ and $\mathcal{P}(p_1, p) = (p_1, p)$. The re-writing algorithm gives us that $p_0 = p + i_0$ and $p_1 = p + i_1$.

The range analysis we use gives: $\texttt{R}(i_0) = [0, N-1]$ and $\texttt{R}(i_1) = [1, N]$. These ranges are over-approximated and make i_0 and i_1 possibly have the same value. Therefore, pointers p_0 and p_1 may dereference overlapping memory regions which is clearly not the case. However, if we adopt a more precise abstract interpretation to generate the possible values of i_0 and i_1 $\mathcal{A}(i_0) = [0, N-1] \cap 2x; x \in \mathbb{N}$ and $\mathcal{A}(i_1) = [1, N] \cap (2x+1); x \in \mathbb{N}$, then we can disambiguate p_0 and p_1 since $\mathcal{A}(i_0) \cap \mathcal{A}(i_1) = \emptyset$.

We close this section with an example in which our three tests fail. In this case, we say that pointers *may alias*. In Example 26, below, we fail to disambiguate pointers, but they alias indeed. We might also fail to disambiguate pointers that do not alias. This type of omission is called a *false positive*.

Example 26 *The program in Figure II.3.16 is the same as the program in Figure II.3.14a, except that the ranges of variables* x*,* y *and* z *have been modified. Due to this modification, none*

```
1 void range(int N, int* p){
2   for(int i = 0; i < N ; i += 2){
3         p[i] = i;             // i0 = i; p0 = p + i0
4         p[i + 1] = i + 1; // i1 = i; p1 = p + i1
5   }
6 }
```

Figure II.3.15 – Program to show the effect of precision of integer analysis on pointer disambiguation.

of our three previous tests can prove that p_2 *and* p_3 *do not alias. The tests fail for the following reasons:*

- p_2 *and* p_3 *are derived from the same base pointer* p_0*; hence, they may alias due to the test of Section II.3.5.1.*
- *Neither* $p_3 \in LT(p_2)$ *nor* $p_2 \in LT(p_3)$. y *and* z*, the offsets of* p_2 *and* p_3 *are not compared since the pointers are not directly related to the same base pointer; hence,* p_2 *and* p_3 *may alias according to the less-than check of Section II.3.5.2.*
- *The range test of Section II.3.5.3 does not fare better. We have that* $p_2 = p_0 + x + y$ *and* $p_3 = p_0 + z$. $R(x + y) = [3, 6]$ *and* $R(z) = [3, 7]$. *These two intervals intersect; hence, the range test reports that* p_2 *and* p_3 *may alias.*

```
1 int may_alias(int N, int C) {
2   int* p0 = malloc(N);
3   int x = 1;
4   int y = C ? 2 : 5;
5   int z = C ? 3 : 7;
6   int* p1 = p0 + x;
7   int* p2 = p1 + y;
8   int* p3 = p0 + z;
9 }
```

Figure II.3.16 – Our analysis reports that pointers p2 and p3 may alias.

II.3.6 Evaluation

In this section, we discuss the empirical evaluation of our analysis and its implementation in the LLVM compiler, version 3.7.

All our experiments have been performed on an Intel i7-5500U, with 16GB of memory, running Ubuntu 15.10. The goal of these experiments is to show that: (i) our alias analysis increases the capacity of LLVM to disambiguate pointers; (ii) the asymptotic complexity of our algorithm is linear in practice, and (iii) the three pointer disambiguation tests are useful.

II.3.6.1 On the Complexity of our Analysis

Our analysis can be divided into preprocessing steps and the actual alias tests. The first step in the preprocessing phase is the collection of constraints. This collection runs in linear time on the

number of program instructions ($O(i)$), because it consists in going through the code, verifying if each instruction defines constraints. The second step of the preprocessing phase consists in building a pointer dependence graph. At this stage we go through the program instructions searching for pointers. Additionally, pointer attributes are propagated along the graph. This step complexity is $O(i+p+e)$, with i being the number of program instructions, p the number of pointers and e the number of edges in the dependence graph. The final preprocessing step consists in running the worklist algorithm. This part of our implementation is equivalent to the problem of building transitive closures of graphs. The worst case of transitive closure is cubic and since every variable could be related to all others, the worst case complexity is $O(c^3 * v)$, c being the number of constraints and v the number of variables. Nevertheless, we have not observed this complexity empirically. The experiments that we perform in this section seem to indicate that our algorithm runs in $O(c)$ in general. This lower complexity is justified by a simple observation: a program variable tend to interact with only a handful of other variables. Thus, the number of possible dependences between variables, in practice, is limited by their scope in the source code of programs that we analyze.

After all the preprocessing, each of our three alias tests have constant complexity. In other words, the relevant computations have been already performed on the preprocessing steps. Thus, each test just checks pointer attributes in a table. Keeping this table requires $O(v^2)$ space, as the number of possible relations between variables is quadratic in the worst case. However, usually this table shall demand linear space.

II.3.6.2 On the Precision of our Analysis

In this section, we compare our analysis against a pointer analysis that is available in LLVM 3.7: the so called *basic alias analysis*, or basicaa for short. This algorithm is currently the most effective alias analysis in LLVM, and is the default choice at the O3 optimization level. It relies on a number of heuristics to disambiguate pointers that we have presented in Section I.2.4.2.1

Program	#Queries	%basicaa	%sraa	%(sraa + basicaa)	%CF
473.astar	90686	40.68	52.15	62.14	X
445.gobmk	3556322	45.16	12.01	48.81	49.91
458.sjeng	528928	67.51	49.65	73.29	73.82
456.hmmer	2894984	8.55	19.60	22.86	53
464.h264ref	19809278	12.37	6.96	14.59	18.64
471.omnetpp	13987157	18.72	1.49	19.67	X
429.mcf	66687	10.70	31.56	34.37	14.05
462.libquantum	120479	39.41	30.27	53.04	42.16
401.bzip2	2485116	20.84	50.67	51.88	22.75
403.gcc	198350037	3.97	13.09	15.51	12.3
483.xalancbmk	11674531	15.35	27.40	32.41	X
400.perlbench	31129433	7.37	12.98	16.28	21.11
470.lbm	31193	3.45	40.17	41.70	3.81

Figure II.3.17 – Comparison between three different alias analyses. We let sraa + basicaa be the combination of our technique and the *basic* alias analysis of LLVM. Numbers in basicaa, sraa, sraa + basicaa, and CF show percentage of queries that answer "no-alias". The X sign indicates that the exhaustive evaluator could not run with Andersen's analysis and aborted.

Figure II.3.17 shows how LLVM basic alias analysis, Andersen's inclusion-based analysis [And94], and our approach fare when applied on the integer programs in SPEC CPU2006 [Hen06]. Henceforth, we shall call our analysis sraa, to distinguish it from LLVM basicaa and Andersen's analysis called CF (because it uses context free languages (CFL) to model the inclusion-based resolution of constraints). Our algorithm sraa is implemented in LLVM 3.7.1. However, CF is distributed in

LLVM versions greater than 3.9.0. In Figure II.3.17, CF numbers have been produced via LLVM 3.9.0. We emphasize that both versions of this compiler produce exactly the same number of alias queries. We notice that, even though the basic alias analysis disambiguates more pointers in several programs, our approach surpasses it in some benchmarks. It does better than basic on `lbm` and `bzip2`. Moreover, we improve basicaa results considerably when both analyses run together on `gobmk`. Visual inspection of `lbm`'s code reveals a large number of hardcoded constants. These constants, which are used to index memory, give our analysis the ability to go beyond what basicaa can do. This fact shows that LLVM still lacks the capacity to benefit from the presence of constants in the target code to disambiguate pointers. In LLVM 3.9.0, we noticed that the basic alias analysis is run as the default analysis instead of no − aa in LLVM 3.7.1 which always returns *may alias*. Therefore, to fairly compare against CF analysis, we consider the numbers we get when combined with basicaa. For programs evaluated by both, this experiment reveals that there is no clear winner between our analysis and CF. In terms of total number of disambiguated pointers, our analysis is slightly more precise than Andersen's one. We did not combine the two analyses, because they do not run in the same version of LLVM. However, we expect our analysis to increase the precision of Andersen's, as the latter does not deal with pointers with offsets.

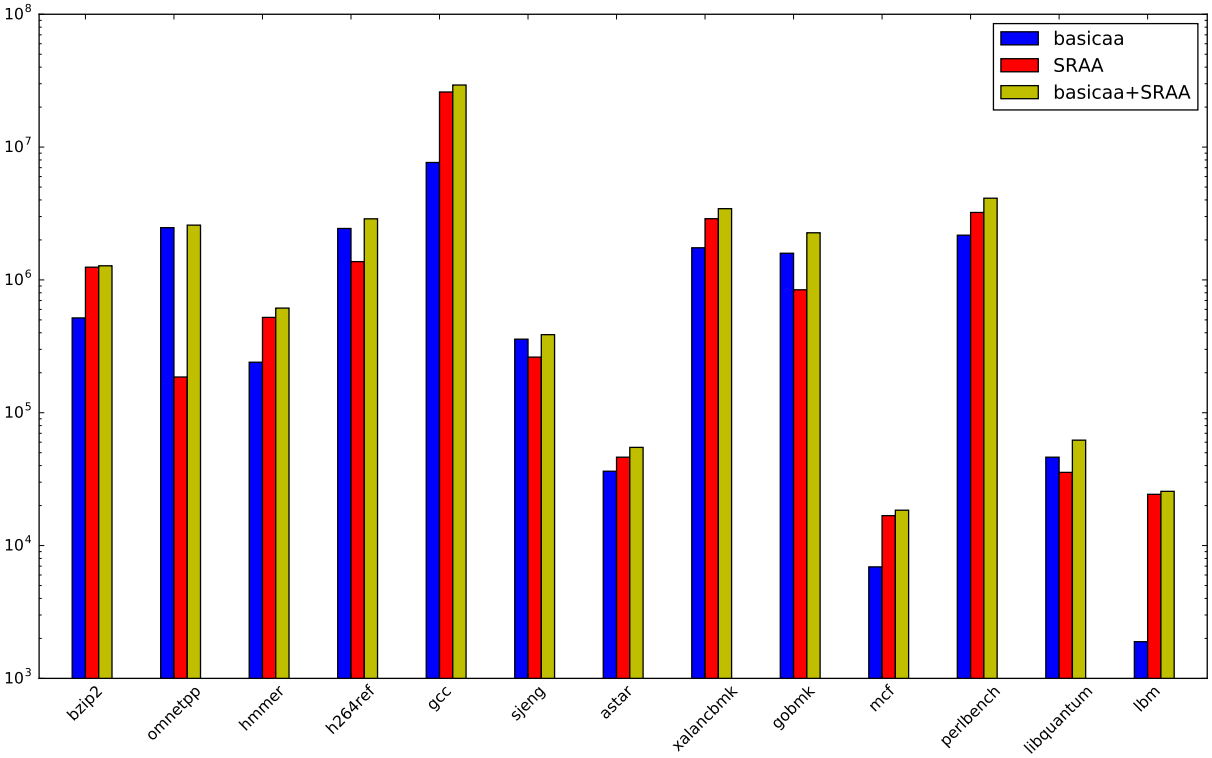


Figure II.3.18 – Comparison between sraa and LLVM basic alias analysis, showing how it improves LLVM capacity to disambiguate pointers. The X-axis shows SPEC CPU2006 benchmarks. The Y-axis shows the number of queries answering no alias. The higher the bar, the better.

Figure II.3.18 compares our analysis against basicaa in terms of the absolute number of queries that they can resolve. A *query* consists of a pair of pointers. We say that an analysis *solves* a query if it is able to answer *no alias* for the pair of pointers that the query represents. When applied onto the programs available in SPEC CPU2006, both analyses, basicaa and sraa, are able to solve several queries. There is no clear winner in this competition, because each analysis outperforms the other in some benchmarks. However, in absolute terms, `sraa` is able to solve about *twice* more queries ($3.6\ 10^7$ vs. $1.9\ 10^7$) than basicaa. Because neither analysis is a superset of the other, when combined they deliver even more precision. The obvious conclusion of this

experiment is that sraa adds a non-trivial amount of precision on top of basicaa. A measure of this precision is the number of queries missed by the latter analysis, and solved by the former.

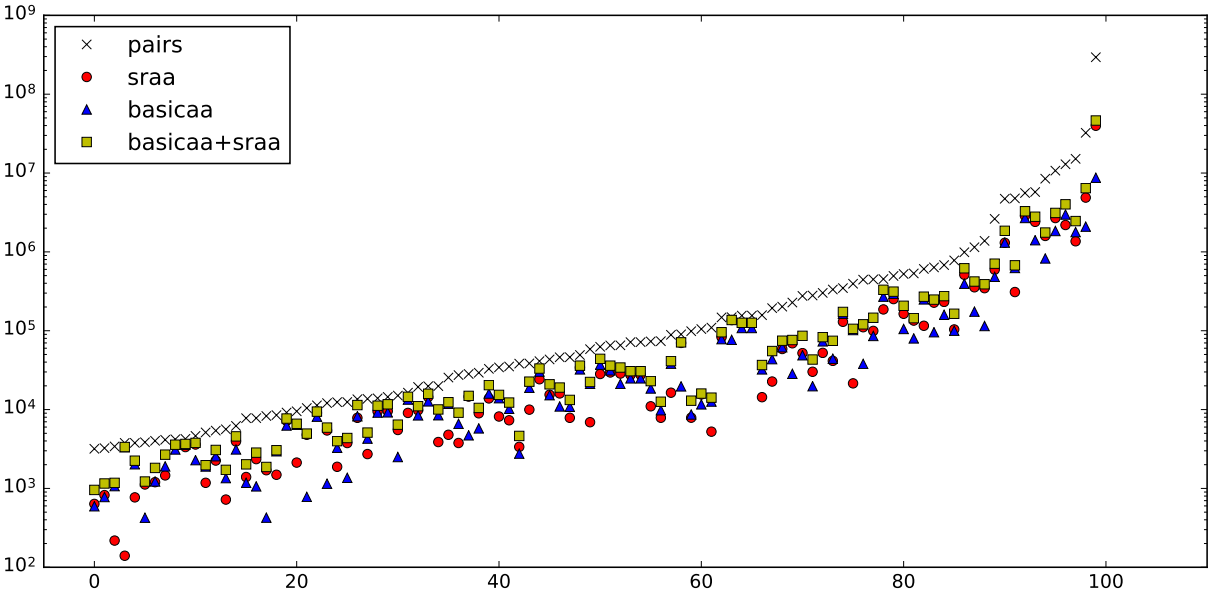


Figure II.3.19 – Effectiveness of our alias analysis (sraa), when compared to LLVM basic alias analysis on the 100 largest benchmarks in the LLVM test suite (TSCV removed). Each tick in the X-axis represents one benchmark. The Y-axis represents total number of queries (one query per pair of pointers), and number of queries in which each algorithm got a "no-alias" response.

Similar numbers are produced by benchmarks other than SPEC CPU2006. For instance, Figure II.3.19 shows the same comparison between basicaa and sraa, this time on the 100 largest benchmarks available in the LLVM's test suite. Nevertheless, the results found in Figure II.3.19 are similar to those found in Figure II.3.18. Our sraa outperforms basicaa in the majority of the tests, and their combination outperforms each of these analyses separately in every one of the 100 samples. We have included the total number of queries in Figure II.3.19, to show that, in general, the number of queries that we solve is proportional to the total number of queries found in each benchmark.

The Role of the Three Disambiguation Tests. Figure II.3.20 shows how effective is each one of the three disambiguation tests that we have discussed in Section II.3.5. In our staged approach, these tests work in succession: given a query q, first we use the PDD test of Section II.3.5.1 to solve it. If this test fails, then we use the less-than test of Section II.3.5.2 to solve q. If this second method still fail to disambiguate that pair of pointers, then we invoke the third test, based on range analysis and discussed in Section II.3.5.3. As Figure II.3.20 shows, most of the queries can be answered by simply checking that different pointers belong to different pointer dependence digraphs (PDDs). Yet, the algebraic rules present in the other two tests are essential in some benchmarks. In particular, the less-than check improves the PDD test by more than 30% on average, and in some cases, such as SPEC's `lbm`, it more than doubles its precision. In some cases, e.g., `astar`, `perlbench` and `lbm`, it is thanks to this test that we outperform basicaa.

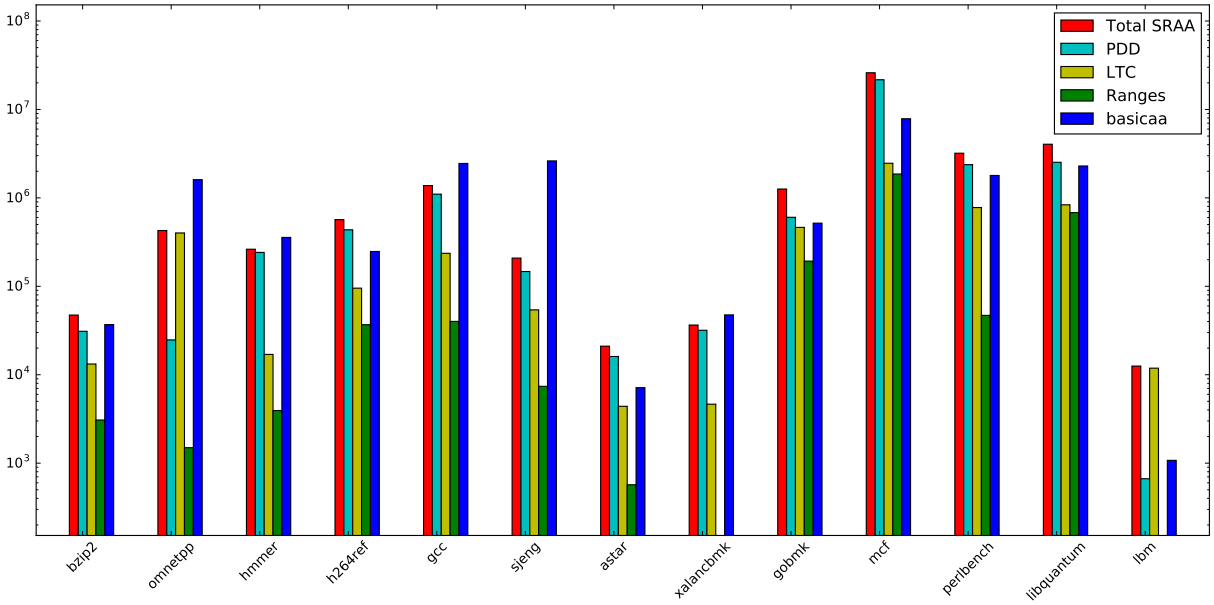


Figure II.3.20 – How the three tests in sraa compete to disambiguate pairs of pointers. The Y-axis denotes the number of disambiguated pairs of pointers.

Figure II.3.20 shows that the range analysis test is not very effective when compared to the other two tests. This low effectiveness is due to two reasons. First, the range test is based solely on a numeric range analysis. Even though a numeric range analysis is enough to distinguish p_1 and p_2 defined as $p_1 = p + 1$ and $p_2 = p + 2$, usually most of the offsets are symbols, not constants. The ability to use symbols to distinguish pointers is one of the key factors that led us to use a less-than check (LTC) in this work. Second, the range test is the last one to be applied. Therefore, most of the queries have already been solved by the other two approaches by the time the range test is called. Specifically, there is a large overlap between the less-than test and the range test. Notably, we have observed that 11.97% of all the queries solved by the less-than check could also be solved by the range test. To fundament this last point, Figure II.3.21 shows the total number of queries solved by each test. In this experiment, we run each test independently; hence, the success of one resolution strategy does not prevent the others from being applied. Nevertheless, the range test of Section II.3.5.3 is still the least effective of the three approaches that we have discussed in this chapter.

Runtime. Figure II.3.22 shows the runtime of our alias analysis on the 100 largest benchmarks in the LLVM test suite. As the figure shows, we finish all the tests for all the benchmarks, but eleven, in less than one second. Nevertheless, we observe a linear behavior. In Figure II.3.22, the coefficient of determination (R^2) between the number of instructions and the runtime of our analysis is 0.8284. The closer to 1.0 this metric, the more linear the correlation between the two quantities.

Figure II.3.23 discriminates the runtime of each disambiguation test. `403.gcc`, our largest benchmark, took approximately 200 seconds to finish. Such long time happens because, in the process of solving constraints, we build the transitive closure of the less-than relations between variables. The graph that represents this transitive closure might be cubic on the number of variables in the program. The figure also shows the total time taken by our alias analysis, which includes the time to construct the transitive closure of the pointer dependence digraph and collecting constraints. The LTC test accounts for most of the execution time within the pointer disam-

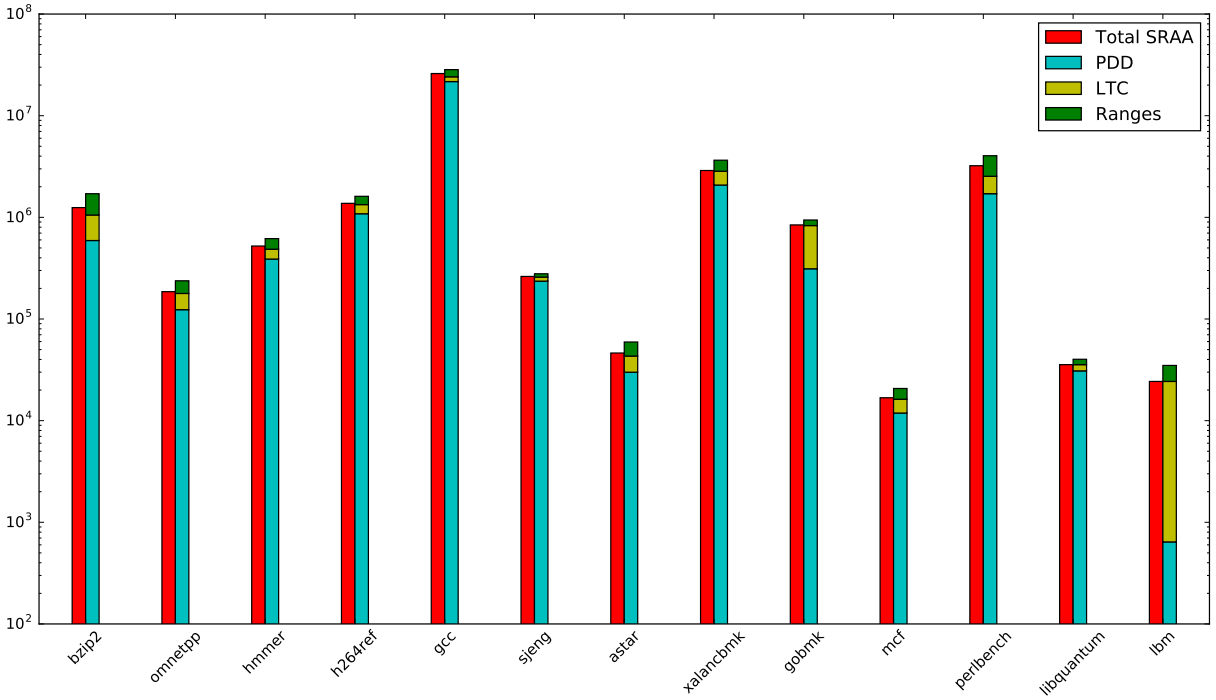


Figure II.3.21 – How the three tests in sraa *separately* compete to disambiguate pairs of pointers.

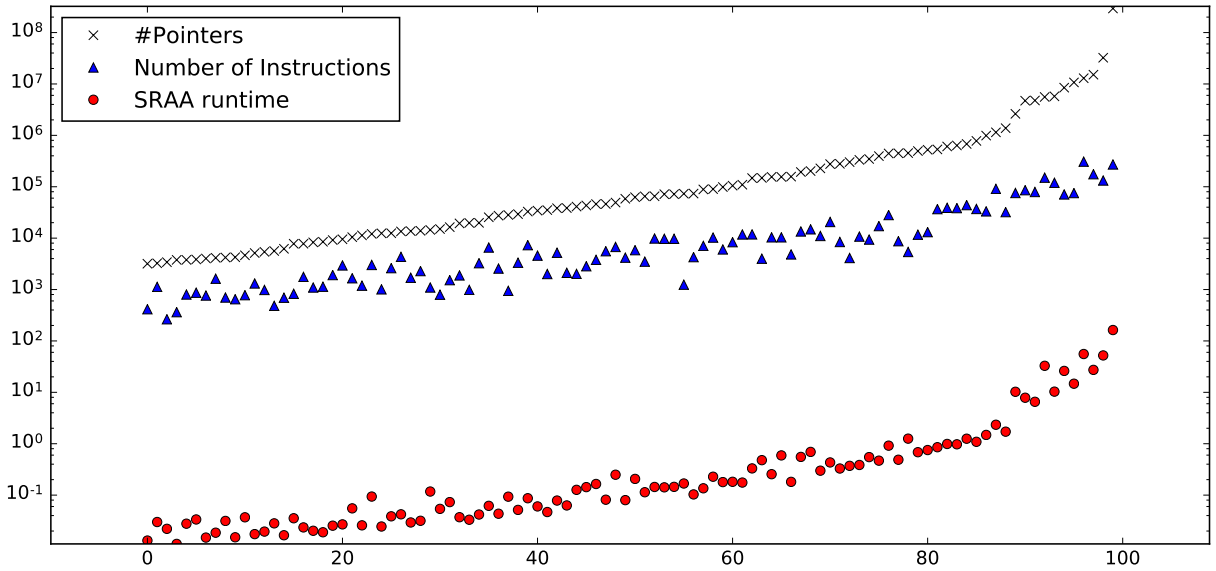


Figure II.3.22 – Comparison between the number of queries and the total runtime (using the 3 tests) per benchmark. X-axis represents LLVM benchmarks, sorted by number of instructions. We measure the SRAA runtime in seconds.

biguation phase of this experiment. This behavior is due to the growth check that we use to solve ϕ-functions. The range and PDD tests take similar runtimes.

In Figure II.3.24, the less-than check runs only if the PDD test fails. Once we have built LT relations, this test amounts to consulting hash-tables. As a final observation, we notice that the range test tends to be the least time-intensive among the three disambiguation strategies that we have. It takes less time because it only runs on pairs of pointers within the same PDD.

Figure II.3.25 compares the time to run the three disambiguation tests in a staged fashion against

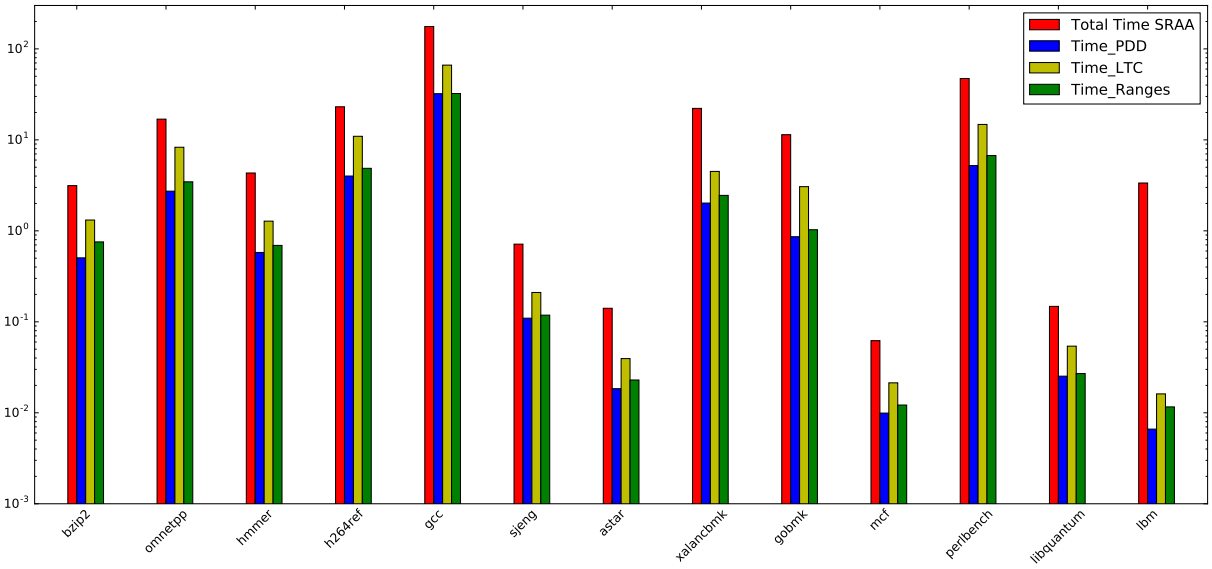


Figure II.3.23 – Time in seconds needed by each test (PDD, LTC, and Ranges) of our analysis sraa to disambiguate SPEC CPU2006 pairs of pointers.

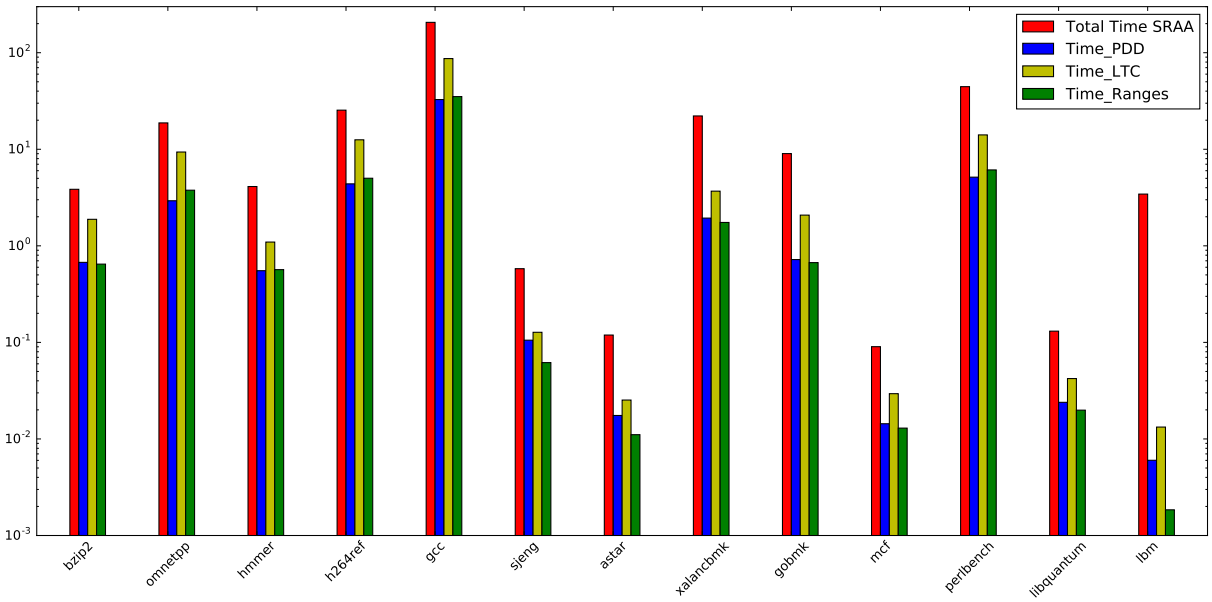


Figure II.3.24 – Time in seconds needed by each test (PDD, LTC, and Ranges) when run selectively to disambiguate SPEC-2006 pairs of pointers.

the time to run these tests independently. The non-staged approach is theoretically slower, because it always runs $3 \times Q$ tests, whereas the staged approach runs $Q + (Q - S_1) + (Q - S_1 - S_2)$, where S_1 are the queries solved by the first test, and S_2 are the queries solved by the second. Nevertheless, Figure II.3.25 does not show a clear winner. The fact that the three tests are relatively fast explains this result.

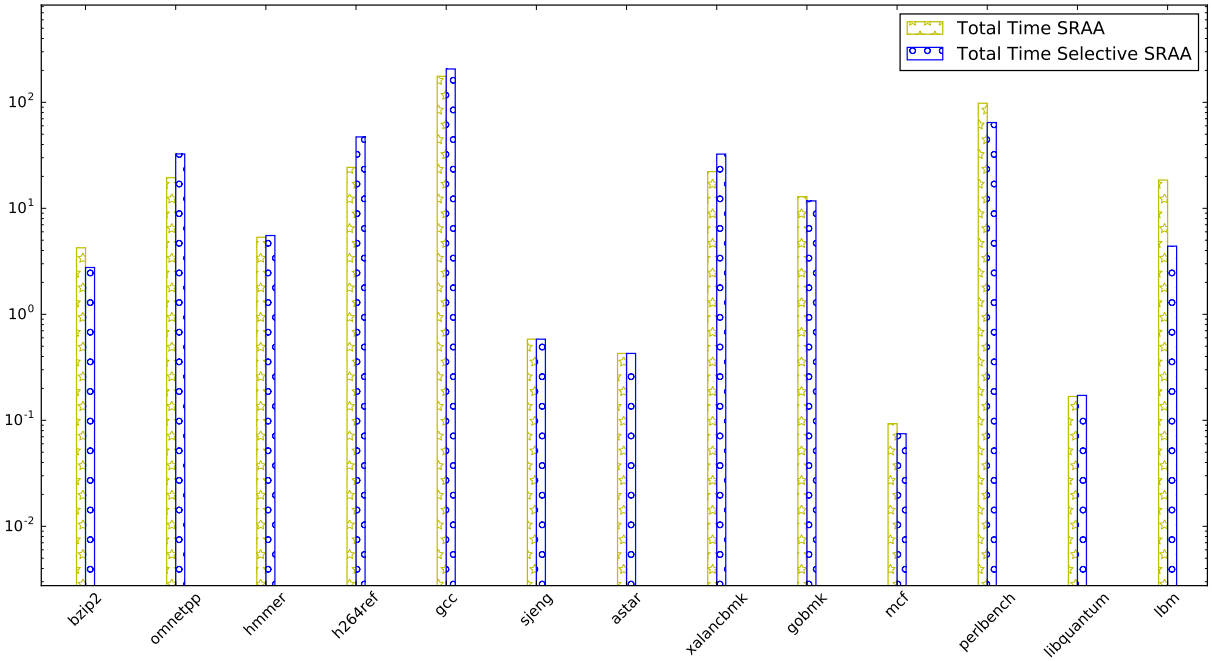


Figure II.3.25 – Comparison between time (in seconds) needed to run sraa *selectively* (non overlapping tests) and *separately* (disjoint tests)

II.3.7 A Comparison Against Alias Analysis Techniques of Chapter II.1 and Chapter II.2

Symbolic Range Analysis of Pointers We compare the range test we develop in this work to the one we presented in Chapter II.1

- In Chapter II.1 we used a symbolic range analysis while in this work we use a numeric one. In our previous work, we found out that 20.47% of the pointers in the three benchmark suites we experimented with have exclusively symbolic ranges. Classic range analysis would not be able to distinguish them. Thus, our range test is less precise in this work than the analysis introduced in Chapter II.1.
- Some of the precision lost by the use of a less accurate range analysis may be recovered by the synthesis of less-than relations between variables. Indeed, in this work, from the numeric range analysis information, we synthesize less-than relations between variables, which could provide the same kind of information a symbolic range analysis would provide, but in a more precise way (a less-than set may contain more than one variable, compared to the unique symbolic upper bound of a given interval).
- In Chapter II.1 we adopt an algebraic formalization to associate pointers with allocation sites. Whereas, we provide a geometric interpretation by building the pointer dependence digraph in the current work. This enables the efficient and precise storage of all the relations between base pointers of all pointers under analysis. In Example 27 we show how this geometric formalization allows us to handle more precisely some pairs of pointers compared to Chapter II.1.

Example 27 *We consider the control flow graph in Figure II.3.26 in which our goal is to disambiguate pointers c_1 and d_1. The allocation site, loc_1, associated to a_1 is unique in this program. It*

is created at $a_1 = malloc()$ *and propagated to other pointers. Using the global analysis*[3] *we introduced in Chapter II.1 we would conclude that* c_1 *and* d_1 *"may alias" since* $GR(c_1) = \mathsf{loc}_1 + [2, 4]$ *and* $GR(d_1) = \mathsf{loc}_1 + [4, 5]$*, and the two ranges have a non-null intersection. On the contrary, the Ranges test we develop in this work is able to disambiguate pointers* c_1 *and* d_1 *as their common immediate ancestor is* a_3*. Rewriting the pointers based on* a_3 *gives:* $c_1 = a_3 + 2$ *or* $c_1 = a_3 + 3$*, and* $d_1 = a_3 + 4$*. Since* $[2, 3] \cap [4, 4] = \emptyset$*, then* c_1 *and* d_1 *certainly do not alias.*

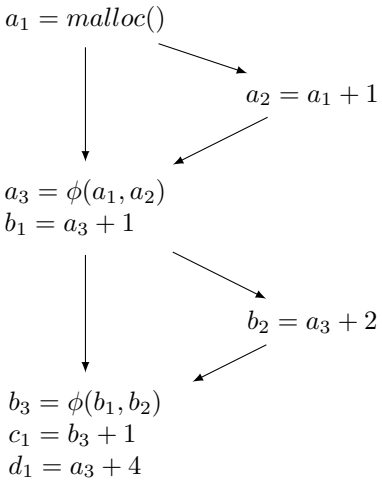


Figure II.3.26 – Illustration of the difference in precision between Chapter II.1 and the Ranges test of this work.

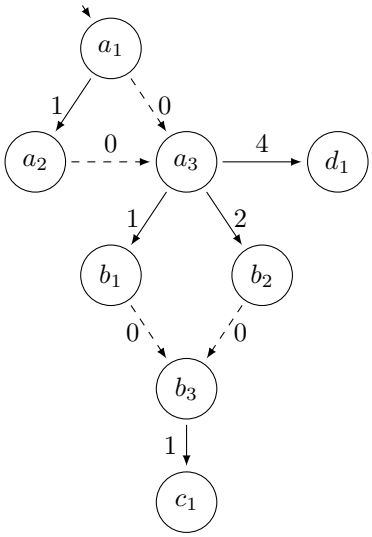


Figure II.3.27 – PDD of program in Figure II.3.26.

Pointer Disambiguation via Strict Inequalities Solving pointer arithmetic based on strict inequalities was also the key idea of our work introduced in Chapter II.2. However, there are some key differences between the two approaches:

- First of all, given the instruction $v = v_1 + v_2$, and the two information (using ranges) $v_1 > 0$ and $v_2 > 0$, the final information we obtain are different: in Chapter II.2, we only generate the constraint $\mathsf{LT}(v) = \{v_1\} \cup \mathsf{LT}(v_1)$. In this work, we also generate the constraint $v_2 < v$ (which is equivalent to $\mathsf{LT}(v) = \{v_2\} \cup \mathsf{LT}(v_2)$). This is linked to the technical reason we develop in the next paragraph.
- Another difference is technical: in Chapter II.2 we consider integers and pointers as variables of scalar type while in the present implementation we make the difference between the two types and handle separately the *add* and *gep* instructions. This is because if v and v_1 are pointers with v_2 integer, *i.e.* $p = p_1 + v_2$, then following the C standard, the comparison between p and v_2 is not allowed, thus we should not store any relationship between p and v_2.
- The third difference between the two works is related to the way we handle sparsity. In Chapter II.2, the new information associated to a variable is created at definition sites and is invariant throughout the analysis. On the contrary, in this work we update the abstract state of a variable whenever a new information is appearing. The main benefit of such update is precision. Indeed, because we compute a transitive closure of all the collected

[3] We have also developed a more fine-grained analysis in Chapter II.1 that we call *local analysis* and run inside SESE blocks with join nodes. Note that this analysis is not helpful in this case because d_1 is not defined based on the join pointer b_3.

relations after their collection, less-than relations are propagated backward and not only forward. Example 28 illustrates the difference.

- A last difference between the work we present in this chapter and the one we detailed in Chapter II.2 is the way we solve ϕ-functions. In Chapter II.2 we do not have the *growth check*. Therefore, we do not build any relation between a ϕ variable and its initialization values. For instance, in the insertion sort algorithm of Figure II.3.1, our previous less that check fails to discover the relation between $\mathtt{i_T}$ and $\mathtt{j_T}$ because we forget that $\mathtt{j}$, which is increasing in the true branch, is greater than or equal $\mathtt{j_0}$ which is strictly greater that $\mathtt{i_T}$.

Clearly, the analysis described here is more expensive than the one of Chapter II.2. However, we experimentally showed that it is still competitive. Moreover, the static single information property [Tav+14] still holds since the abstract state of a variable is unique and invariant at all program points of *its live range*.

Example 28 *In this example we illustrate the difference in precision between the work of this chapter and the one we presented in Chapter II.2. This is related to the way we compute the transitive closure. We consider the program snippet in Figure II.3.28. In Chapter II.2, since* $x2 = x1 - 4$ *is a subtraction, we rename variable* $x1$ *to* $x1'$ *for instance. The final result of the analysis gives:* $LT(x1) = \{x0\}$, $LT(x1') = \{x0, x2\}$, $LT(x2) = \{\}$, $LT(x3) = \{x0, x1', x2\}$. *Using the rules of Figure II.3.7 and Figure II.3.8, we generate and solve constraints which gives:* $LT(x1) = \{x0\}$, $LT(x1) = \{x0, x2\}$, $LT(x2) = \{\}$, $LT(x3) = \{x0, x1, x2\}$. *We want to disambiguate pointers* $p1$ *and* $p3$ *which are both defined from the same base pointer* p. *Following the second criterion to answer alias queries in the Less-Than test (Section II.3.5), in the previous work we answer "may alias", however in the current one we answer "no alias". Notice that this is because we lost the relation between* $x1$ *and* $x3$ *after the renaming in the split point. The pointer* $p1$ *has already been defined at this point using* $x1$ *and not* $x1'$.

```
1  int x0;
2  int* p = malloc((x3+1)*sizeof(int));
3  int x1 = x0 + 2;
4  int* p1 = p + x1;
5  int x2 = x1 − 4;
6  int x3 = x1 + 6;
7  int* p3 = p + x3;
```

Figure II.3.28 – Program to illustrate the differnce in precision between Chapter II.2 and the Less-Than test of this work.

II.3.8 Conclusion

This chapter has described a novel algebraic method to disambiguate pointers. The technique that we have introduced uses a combination of less-than analysis and classical range analysis to show that two pointers cannot dereference the same memory location. We have demonstrated that our technique is effective and useful: its implementation on LLVM lets us increase the ability of this compiler to separate pointers by almost five times in some cases. And, contrary to previous algebraic approaches, our analysis scales up to very large programs, with millions of assembly instructions. We believe that this type of technique opens up opportunities to new program optimizations. The impact of our analyses on such optimizations is a challenge that we try to address in the next chapter.

Chapter II.4

Alias Analyses for LLVM Optimizations

Contents

To evaluate the efficiency of our pointer analyses in terms of optimizing programs, we have evaluated the impact they have on the LLVM optimizations. We studied three examples of passes that optimize programs: *Loop Invariant Code Motion* (LICM), *Dead Store Elimination* (DSE), and *Global Value Numbering* (GVN). The range analysis of pointers uses symbolic ranges in its version introduced in Chapter II.1 while these ranges are numeric in the combined version of the analysis presented in Chapter II.3. Since symbolic ranges are more precise than numeric ones, we chose in this chapter to study the impact of the algorithms of Chapter II.1 (rbaa) and Chapter II.3 (sraa).

For LICM optimizations, we aim at making the pass hoist and sink more loads and stores outside loops based on the extra alias analysis information provided by our analysis passes rbaa and sraa. For GVN, we measure the effect of the analyses in terms of the number of deleted loads and instructions. We shall be interested in the number of deleted store instructions when studying the DSE optimizations. We compare our results to the basic configuration of LLVM.

In this chapter, we report the number of optimizations performed by LICM, GVN, and DSE run among other O3 optimizations, and the number of O3 optimizations only. Some of O3 passes invalidate our analysis rbaa and cause a lot of SageMath crashes. Sage is a powerful software used for computations in advanced mathematics. In our rbaa analysis, it represents the basis of the symbolic range analysis for integer variables that we use as pre-analysis. To avoid some of these crashes, we have removed suspicious passes, and to fairly compare against O3, we also removed these passes from the O3 package. The rbaa analysis was implemented in LLVM 3.5 while sraa was implemented in LLVM 3.7. Each of these versions represents the last LLVM release at the moment of implementation. However, the two versions do not show the same order of passes neither exactly the same implementations. Some passes have been added to LLVM 3.7 list of analyses and optimizations run by O3. This makes the number of handled instructions different and the comparison of the two versions not meaningful. For these reasons we will separately present results for our new analyses. In the sequel, a cross `X` in the reported figures denotes that we were not able to run the program under the specified configuration.

II.4.1 Loop Invariant Code Motion Optimization

The *Loop Invariant Code Motion* transform pass (introduced in Section I.1.4.2) is a *loop function* that aims at hoisting or sinking invariant calls and loads outside the loop. To that end, it ensures that every loop has a preheader inserted by the pre-pass loop-rotate. The loop-rotate is a transform pass that builds the blocks where instructions hoisted or sunk may be safely moved. Note that moving an instruction out of the loop is only possible if it is guaranteed to be executed at least once. Going back to the example we studied in Section I.1.4.2, we recall in Figure II.4.1 the program of Figure I.1.10a and show the control flow graph generated in Figure II.4.2. We want to remove the load instruction at line ℓ_7 out to the preheader where the loop condition ($p_2 > p_1$) is satisfied and the instruction is executed only once. Notably, and as we shall detail in the sequel, an alias analysis is necessary to enable this optimization. Note that none of our alias analysis techniques presented in Part II nor those basically used in the LLVM 3.7 compiler prove that p_2 and p_1 do not alias.

To investigate how this optimization decides to hoist or sink instructions, we have visually inspected the pass code. We found that a load or a store instruction analyzed by LICM is hoisted out of the loop if it satisfies three conditions:

1. It has loop invariant operands: This test ensures that hoisted instructions have only loop-invariant operands so that it would be safe to hoist the instruction out to the preheader.
2. It can be sunk or hoisted: This test returns *true* if the hoister or sinker can handle the

```
1  assert(N>1);
2  int* p = malloc (2*N*sizeof(int));
3  int *p1, *p2, a;
4  *p = 8;
5  a = 10;
6  p1 = p + N;
7  p2 = p + 2 * N - 1;
8  while(p2>p1){
9        a = *p;
10       *p2 = 4;
11       p2 --;
12 }
```

Figure II.4.1 – Example to show the LICM hoisting load optimization.

given instruction. In this very test, LICM uses the alias analysis results if needed. Note that volatile loads are not hoistable while loads from constant memory are always safe to hoist. The alias analysis allows LICM to check the existence of writes to memory in the analyzed loop and whether a load instruction has a may alias store in which case it is not safe to sink or hoist the instruction.

3. It is safe to be executed unconditionally: An instruction is said "safe to execute unconditionally" if either of the following is true:
   - It is safe to speculatively execute: This means that executing the instruction has no side effects besides calculating the result and does not have undefined behavior like dividing by zero or loading from an invalid pointer. This also includes checking for `malloc` and `alloca` execution since they might cause a memory leak. Due to these conditions, this test rarely returns true for load instructions. Many patches have been proposed (and applied to next LLVM releases) to overcome this limitation like running a context-sensitive analysis to check whether it is safe to execute the instruction, taking alignment into account[1], and exploiting the "dereferenceable_or_null attribute"[2] that tells the optimizer that the value loaded is known to be either dereferenceable or null.
   - It is guaranteed to execute: This test is run if the previous one "is Safe To Speculatively Execute" fails. Its goal is to check whether the load will be executed for sure or not. This means that there are no paths out of the loop which do not execute this instruction, i.e., the instruction should dominate all of the loop exit blocks. In fact, if the loop may throw exceptions, the load instruction may not be executed in the loop body so cannot be put in the preheader where it will be *wrongly* executed. If the loop is infinite, the load instruction is not handled since the loop has no exit blocks.

An instruction is sunk if both of the following tests return *true*:

1. It is not used in the loop,
2. It can be sunk or hoisted.

Due to the limitations related to the analysis chaining behavior (Section I.2.4.2.1), to experiment LICM optimizations, we have inserted our rbaa and sraa passes before the LICM pass and after an instance of loop-rotate[3]. We evaluate our analyses for LICM within O3 optimizations and against O3 when analyzing and optimizing Mallocbench, Prolang-C, and Ptrdist benchmarks of the LLVM

[1] Available at http://lists.llvm.org/pipermail/llvm-commits/Week-of-Mon-20150817/294018.html.
[2] Available at http://lists.llvm.org/pipermail/llvm-commits/Week-of-Mon-20150518/276919.html.
[3] Note that we have altered the order in which the pass loop-rotate is executed without breaking its dependencies regarding LICM to avoid a permanent Sage crash when interfering with the loop-rotate.

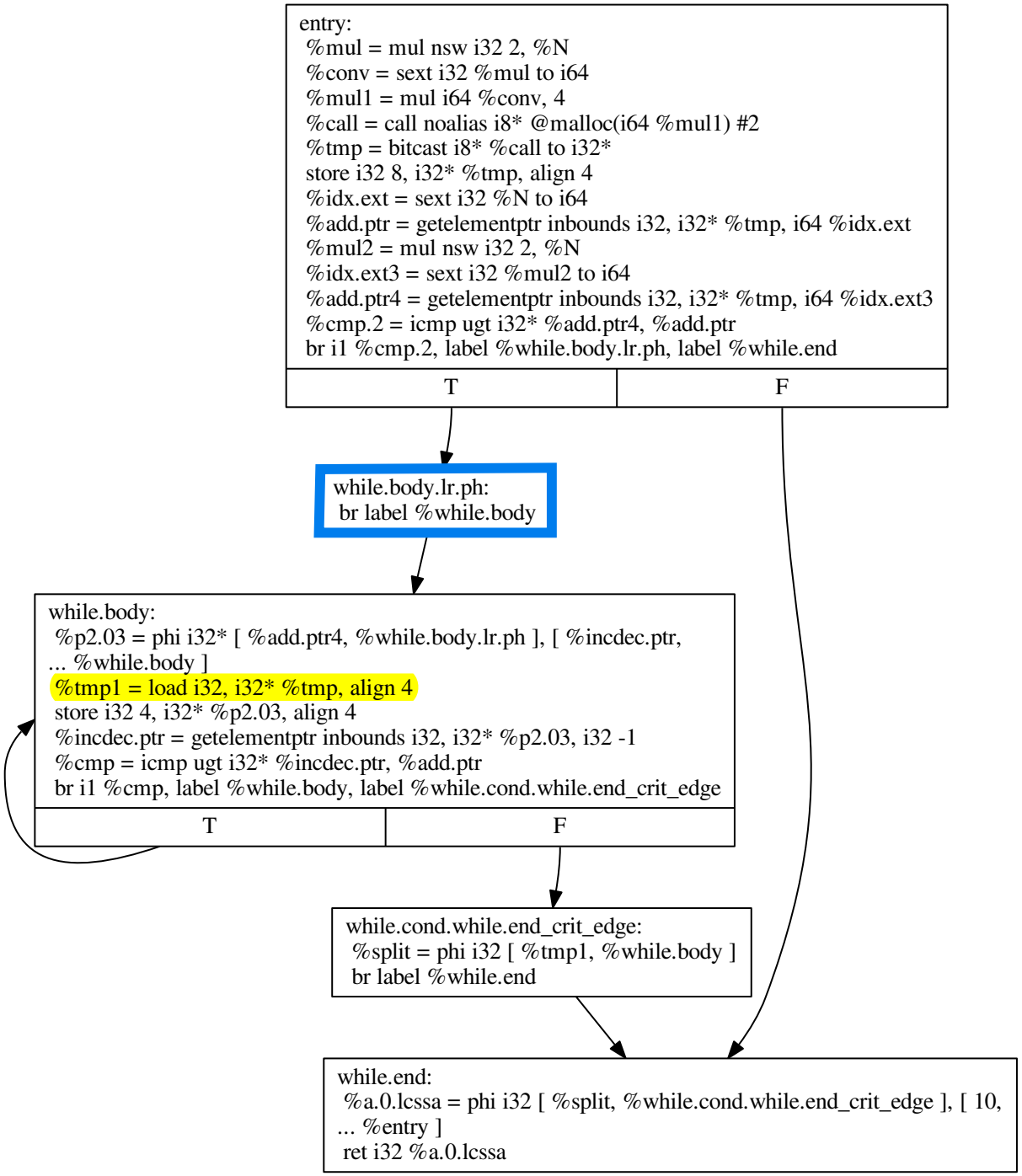


Figure II.4.2 – CFG of program in Figure II.4.1 after applying the loop rotate, the basic alias analysis and the LICM optimization pass. The highlighted block is the preheader where the loop invariant load instruction %`tmp1` = `load i32, i32*%tmp, align 4` should be hoisted.

test-suite. We give the statistics for the symbolic range analysis of pointers (rbaa) (Chapter II.1) in Figure II.4.3 and for the combined analysis (sraa) in Figure II.4.4.

In Figure II.4.3 and Figure II.4.4, the programs under test do not show any removed calls or show few of them compared to the hoisted or sunk loads. We will therefore focus on load optimizations since these benchmarks do not seem to include a potential for call optimizations.

Program	#Inst	#callHoistedSunk		#loadHoistedSunk	
		O3	O3rbaa	O3	O3rbaa
cdecl	3074	0	0	6	6
football	9623	0	0	13	**26**
simulator	7113	0	0	4	4
assembler	4048	0	0	0	0
loader	1763	0	0	0	0
gnugo	3361	0	0	2	**12**
unix-tbl	9584	0	0	54	54
agrep	11888	0	0	9	**71**
fixoutput	387	0	0	0	0
compiler	2866	0	0	0	0
bison	12877	0	0	38	38
archie-client	5011	0	0	0	0
TimberWolfMC	69806	0	X	597	X
allroots	423	0	0	0	0
unix-smail	4046	0	0	1	1
plot2fig	1243	0	0	2	2
bc	13556	0	0	1	1
yacr2	5331	0	0	86	**100**
ks	1304	0	0	7	**11**
anagram	796	0	0	1	1
ft	1602	0	0	3	**4**
cfrac	7766	0	X	4	X
espresso	46954	0	0	103	**243**
gs	40976	0	X	14	X

Figure II.4.3 – Optimizations performed by LICM for O3 and rbaa within O3 (O3rbaa). Benchmarks are run on LLVM 3.5. The X sign denotes that the program has not been analyzed due to a range pre-analysis fail.

Program	#Inst	#callHoistedSunk		#loadHoistedSunk	
		O3	O3sraa	O3	O3sraa
cdecl	4859	0	0	5	**6**
football	14529	0	X	22	X
simulator	7698	0	0	10	**16**
assembler	6661	0	0	0	0
loader	2263	0	0	0	0
gnugo	5449	0	0	10	**12**
unix-tbl	8101	1	X	61	X
agrep	15382	4	X	53	X
fixoutput	369	0	0	1	**5**
compiler	3515	0	0	0	0
bison	15645	1	1	165	**179**
archie-client	5939	0	0	0	0
TimberWolfMC	98792	2	7	1287	**1447**
allroots	574	0	0	0	0
unix-smail	5435	4	4	3	3
plot2fig	3217	1	1	3	3
bc	10632	0	0	18	**19**
yacr2	6583	0	0	144	**190**
ks	1368	0	0	8	**11**
anagram	993	1	1	4	4
ft	1646	0	0	4	4
cfrac	7353	1	1	5	**6**
espresso	50751	1	1	301	**398**
gs	55281	0	X	20	X

Figure II.4.4 – Optimizations performed by LICM for O3 and sraa within O3 (O3sraa). The number of instructions is the one computed before all analysis and transform passes of LLVM 3.7 are run.

LICM uses the basic alias analysis of LLVM to build alias sets. We already showed in Section II.1.6 that our alias analysis rbaa based on the symbolic range analysis of integer variables outperforms both scev and basicaa used to disambiguate pointers. On the optimizations yielded by the two analyses basicaa and rbaa, Figure II.4.3 shows that our analysis improves the performance of LICM optimization by a factor of 1.77. This factor does not include the programs TimberWolfMC, cfrac, and gs for which our analysis combined with the other LLVM passes reports compilation errors. Due to the lack of time and the complexity of the LLVM pass-chaining, we did not investigate to find out whether the analysis abort is due to our sraa pass output or the way that the LICM pass consider it.

In Chapter II.3, we compare our analysis sraa based on the three disambiguation tests to LLVM alias analyses using the SPEC benchmarks in terms of percentage of disambiguated pointers. However, the results we collected in Figure II.4.4 are sufficient to conclude that when it is run on the LLVM test-suite, our analysis outperforms again the basicaa and scev analyses.

These results obtained for sraa and rbaa are promising and prove that the alias analyses inside LLVM could benefit from more precision to enable more optimizations. We also believe that the performance of LICM could be significantly improved whenever the other conditions it checks to hoist a load instruction (for instance) are made more precise. These tests are as important as alias analysis information, but not as popular.

II.4.2 Dead Store Elimination

The *Dead Store Elimination* pass removes a store if it is preceded by another store instruction made dead by the current one. To delete the first store, DSE checks the following conditions:

1. It is removable,
2. It is not a possible self read,
3. It is overwritten.

We shall discuss these conditions later. For now, among these tests, we find the condition "is overwritten", i.e., the second store should totally overwrite the first one. To discover the presence of a potential dead store, DSE performs a top down walk on the basic blocks. Whenever it finds a store, it seeks for its memory dependence: the previous read or write that is related to the current store. Such relation is given by the memdep pass that uses alias analysis. If the preceding memory dependence found is a load, then it is not possible to optimize the store since the same memory region is read. Example 29 illustrates this fact. However, if the memory dependence returned by memdep is a store instruction, then DSE checks the three conditions and if they are satisfied, the dead store is deleted.

Example 29 *Consider the program snippet in Figure II.4.5. In this example, we suppose that* memdep *reports the dependence* `d = *a` *to the instruction* `*b = 6` *analyzed by* DSE. *This means that the instruction* `d = *a` *which is a read, accesses the memory region pointed by pointer* `b`. *Let us call this region* `X`. *The* DSE *pass does not seek for a previous store, e.g.,* `*a = 3`, *because if this store is completely overwritten by* `*b = 6`, *then it modifies the region* `X` *which is read by* `d = *a`. *Thus, the deletion of* `*a = 3` *cannot be performed.*

Suppose now that memdep returns a dependent instruction which is a store. DSE checks for the following conditions to possibly delete that store.

1. The instruction is not a possible self read: DSE does not remove the store instruction if it is a possible *self read*. A self read store is an instruction that does not make its input dead.

```
1 void dse_load(int *a, int *b){
2 *a = 3;
3 int d = *a;
4 *b = 6;
5 }
```

Figure II.4.5 – Program illustrating the memory dependence used by DSE.

Let us consider two memory locations A and B and the following couple of instructions:

$$memcpy\ (A \leftarrow B)$$
$$memcpy\ (A \leftarrow A)$$

The second store to A does not kill the first one since the value stored to A in the first memcpy which is B is assigned to A in the second store. Therefore, removing the first store is unsafe. Note also that we can have the same situation of self read if in the second assignment we store to A the value of C which may alias A.

2. The instruction is removable: This condition is set to *true* if the variable stored to is non *volatile* (the compiler is able to not preserving the data dependencies of involved variables).
3. The instruction is overwritten: If a store instruction satisfies both the first and second condition of DSE store removing, *i.e.*, is removable and is not a possible self read, then the transform pass may remove that store if it is overwritten by the following one. In other words, the second stored value has a larger size than the first one. To distinguish the two situations that DSE considers to handle this condition, we let p_1 and p_2 be two memory locations and consider the following instructions:

```
1 *p1 = a;
2 *p2 = b;
```

The first situation DSE checks for is whether p_1 and p_2 have the same address: $p_1 = p_2$. If true, then it compares the sizes of a and b and returns *true* if $size(a) \leq size(b)$.
The second situation is a bit more complicated. In fact, p_1 and p_2 could not be the same but alias. Therefore, they dereference overlapping memory regions which lets the second store (to the memory slots pointed by p_2) potentially overwrites the first one (the memory slots pointed by p_1). DSE performs the check by trying to rewrite p_1 and p_2 as *constant* offsets from the same base pointer and then compare the size of offsets added to the stored values. For instance, if we have:

```
1 p1 = p + c1;
2 p2 = p + c2;
3 *p1 = a;
4 *p2 = b;
```

then, we compare $size(a) + c1$ to $size(b) + c2$. Note that all these comparisons are done at the byte level.

Figure II.4.6 shows the store optimizations performed by DSE. For programs we were able to analyze and optimize, our analysis rbaa run with O3 optimizations was able to remove 1.18x of dead store instructions more than basicaa does (0.064% for rbaa against 0.054% for basicaa). Regarding pointer disambiguation performance, our analysis rbaa combined with the basic analysis was able to disambiguate 1.31x more pairs of pointers for the same programs (Figure II.1.14).

Program	#Inst	#StoreDeleted	
		O3	O3rbaa
cdecl	3763	0	X
football	8410	0	0
simulator	8909	0	0
assembler	5622	0	X
loader	3066	0	0
gnugo	4140	3	3
unix-tbl	9660	5	X
agrep	11623	0	**2**
fixoutput	462	0	0
compiler	4896	0	0
bison	12868	0	0
archie-client	4954	0	X
TimberWolfMC	70115	26	X
allroots	452	0	0
unix-smail	4026	0	0
plot2fig	1700	0	0
bc	14394	26	**30**
yacr2	5737	0	0
ks	1652	0	X
anagram	825	0	0
ft	1666	0	1
cfrac	8582	16	X
espresso	51133	15	X
gs	46282	42	**48**

Figure II.4.6 – Optimizations performed by DSE for O3 and rbaa within O3 (O3rbaa). The number of instructions is the one computed before DSE is run on LLVM 3.5.

We may think that the proportions of DSE optimizations and rbaa pointer disambiguation results are coherent since the pairs of pointers that an alias analysis disambiguates are not necessarily used in optimizations and specifically in the dead store elimination one. However, tested with micro benchmarks we have written, we found out that some stores were not deleted while they should be. For these programs, our tested analyses rbaa and sraa answers "no alias" for the involved pairs of pointers but DSE keeps the dead stores unchanged. To understand this unexpected behavior, we consider the programs of Figure II.4.7 that we investigate using the sraa alias analysis tests.

In both programs, we want to eliminate the dead store at line ℓ_6. Notice that the unique difference between the programs in Figure II.4.7a and Figure II.4.7b is the offset of pointer `v` at lines ℓ_6 and ℓ_7. In the former, the offset is variable `i` that goes from $[1, \texttt{N}-1]$ and in the latter it is constant and equals 1. In both cases the basic alias analysis answers may alias for the two couples of memory locations $(\texttt{v[i]}, \texttt{v[N]})$ and $(\texttt{v[1]}, \texttt{v[N]})$. Therefore, when using basicaa, `DSE` stops seeking for possible optimizations while analyzing the store instruction at line ℓ_7 (for the reasons detailed in Example 29). When analyzing the two algorithms using our alias analysis sraa, the memory dependence reported by memdep is the store at line ℓ_6 (and not the load at line ℓ_7). This is because our less-than test is able to prove that $\texttt{v}+\texttt{i} < \texttt{v}+\texttt{N}$ and $\texttt{v}+\texttt{1} < \texttt{v}+\texttt{N}$ inside the loop.

Although memdep, based on the alias information provided by sraa, is able to find the suitable dependence to eliminate the store at line ℓ_6, DSE does not delete it in the program of Figure II.4.7a. Looking at the three conditions tested to assert the store deletion, we found out that the test "is overwritten" fails. DSE was able to see the two memory locations `v[i]` at lines ℓ_6 and ℓ_7 as offsets from the same base pointer which is `v`. However, it fails when it comes to compare offsets since they are not constant (offset `i`). To further validate our claim, we attempted to optimize the program in Figure II.4.7b which is similar to the program in Figure II.4.7a but with constant offsets. This time, DSE manages to perform the optimization and deletes the dead store at line ℓ_6. The control flow graph of the resulting analysis and optimization is illustrated

```
1  int variable_offset(int* v, int N){
2    v[N] = 42;
3    int i = 1;
4    if (N > 1)
5          while (i < N){
6            v[i] = 0;
7            v[i] = v[N];
8            i++;
9    }
10 return 0;
11 }
```

(a) Program with variable offsets.

```
1  int constant_offset(int* v, int N){
2    v[N] = 42;
3    int i = 1;
4    if (N > 1)
5          while (i < N){
6            v[1] = 0;
7            v[1] = v[N];
8            i++;
9    }
10 return 0;
11 }
```

(b) Program with constant offsets.

Figure II.4.7 – Programs to evaluate DSE limitations.

in Figure II.4.8. This optimized version of the program in Figure II.4.7b does not show any store to the memory location v[1] of value 0. DSE was able to delete the store as offsets used to access the array v are constant.

Based on the experiments we conducted to evaluate the LLVM DSE pass under different alias analysis algorithms, we conclude that providing this pass with a more precise alias analysis does not guarantee a better performance. Yet, we find the behavior of DSE regarding constant offsets not surprising. Comparing variable offsets is not straightforward since, at the time of this writing, there is no analysis in the distributed versions of LLVM that provides this information. We believe, however, that this imprecision may be partially recovered by adding a range analysis information (Section II.1.3.3). An over-approximation of the possible values an integer variable may take throughout the program execution would allow us to compare these variables.

II.4.3 Global Value Numbering

The *Global Value Numbering* (GVN) sequence of optimizations consists in eliminating common sub-expressions, partially redundant expressions, and hoisting loop-invariant expressions out of loops. Its role is to recognize expressions in the program that compute the same static value. Similarly to the LICM and DSE passes, we experimented the optimizations related to memory accesses performed by GVN within LLVM O3. The GVN pass deletes loads based on the *memory dependence* (memdep) analysis that uses the alias analysis information to catch relations between pointers. GVN looks for redundant loads for instance in a way much similar to how DSE does to discover dead stores. If the analyzed instruction is a load, GVN looks for its dependencies this time locally (inside the same basic block) and non-locally (in basic blocks dominated by the current one).

Before presenting the performance of GVN on the LLVM test-suite, let us better understand what is the global value numbering and how such optimization looks like once applied on a simple program code. To that end, we consider the program in Figure II.4.9.

In this example, our aim is to propagate the loaded value v[b] at line ℓ_{11} to line ℓ_{12} at which v[b] is loaded again. This propagation is not straightforward and needs the disambiguation of v[a] and v[b] because we have a write to v[a] at line ℓ_{11} which may affect v[b]'s value. The second expression that may be propagated is the loaded value v[a] from line ℓ_{11} to line ℓ_{12} to perform the addition instruction. No alias analysis information is needed here since we never write to

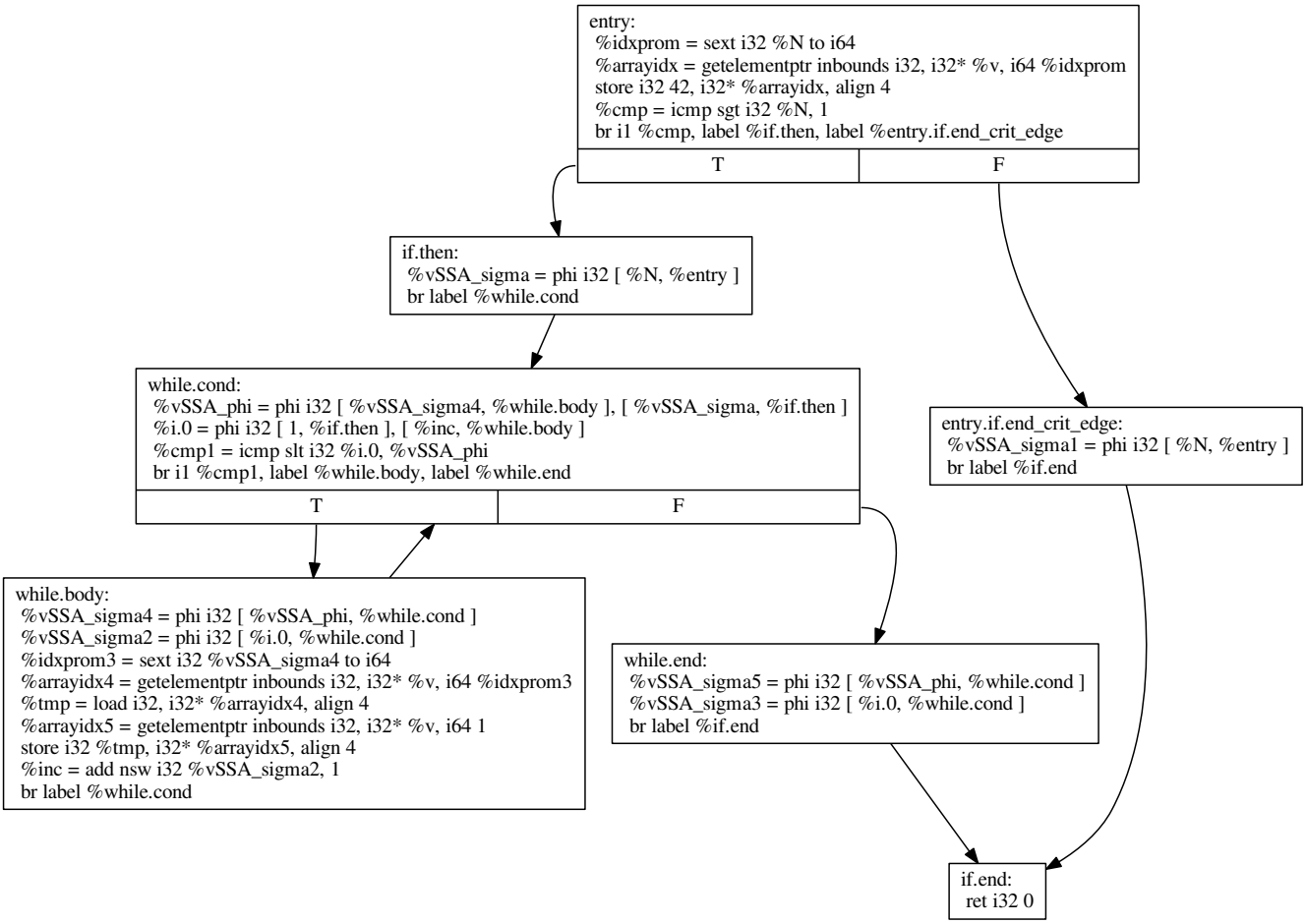


Figure II.4.8 – Control flow graph of the program in Figure II.4.7b. We focus on the while loop body to note the presence of only one store instruction. The store `v[1] = 0` has been removed by DSE since it is killed by the store `v[1] = v[N]` (`store i32 %tmp, i32* %arrayidx5, align4`). The stored value `v[N]` is denoted by `%tmp` and the memory location `v[1]` is called `%arrayidx5`.

```
1  int* switchCase(int N) {
2    int* v = (int*)malloc(6 * sizeof(int));
3    int a, b;
4    v[0] = N;
5    v[1] = N;
6    switch(N % 3) {
7      case 0:  a = 0; b = 3; break;
8      case 1:  a = 1; b = 4; break;
9      default: a = 2; b = 5;
10   }
11   v[a] += v[b];
12   v[a] += v[b];
13 return v;
14 }
```

Figure II.4.9 – Example to study load value propagation performed by the global value numbering.

another pointer meanwhile. The GVN transform pass is responsible for these value propagations. Figure II.4.10 and Figure II.4.11 show the code of the two instructions of lines ℓ_{11} and ℓ_{12} in the LLVM IR. They illustrate respectively the original and the desired optimized versions. Memory locations v[a] and v[b] should be loaded once in the optimized version. These values are propagated from previous instructions.

```
1  %arrayidx = getelementptr inbounds i32, i32* %tmp, i64 %idxprom   ; vb = v + b
2  %tmp3 = load i32, i32* %arrayidx, align 4                         ; tmp3 = load vb
3  %arrayidx2 = getelementptr inbounds i32, i32* %tmp, i64 %idxprom1 ; va = v + a
4  %tmp6 = load i32, i32* %arrayidx2, align 4                        ; tmp6 = load va
5  %add = add nsw i32 %tmp6, %tmp3                                   ; add = tmp3 + tmp6
6  store i32 %add, i32* %arrayidx2, align 4                          ; store add va
7  %tmp9 = load i32, i32* %arrayidx4, align 4                        ; tmp9 = load vb
8  %tmp12 = load i32, i32* %arrayidx6, align 4                       ; tmp12 = load va
9  %add7 = add nsw i32 %tmp12, %tmp9                                 ; add7 = tmp12 + tmp9
10 store i32 %add7, i32* %arrayidx6, align 4                         ; store add7 va
```

Figure II.4.10 – Instructions of ℓ_{11} and ℓ_{12} in Figure II.4.9 in LLVM IR.

```
1  %arrayidx = getelementptr inbounds i32, i32* %tmp, i64 %idxprom   ; vb = v + b
2  %tmp3 = load i32, i32* %arrayidx, align 4                         ; tmp3 = load vb
3  %arrayidx2 = getelementptr inbounds i32, i32* %tmp, i64 %idxprom1 ; va = v + a
4  %tmp6 = load i32, i32* %arrayidx2, align 4                        ; tmp6 = load va
5  %add = add nsw i32 %tmp6, %tmp3                                   ; add = tmp3 + tmp6
6  store i32 %add, i32* %arrayidx2, align 4                          ; store add va
7  %add7 = add nsw i32 %tmp12, %tmp9                                 ; add7 = add + tmp3
8  store i32 %add7, i32* %arrayidx6, align 4                         ; store add7 va
```

Figure II.4.11 – Instructions of ℓ_{11} and ℓ_{12} in Figure II.4.9 optimized.

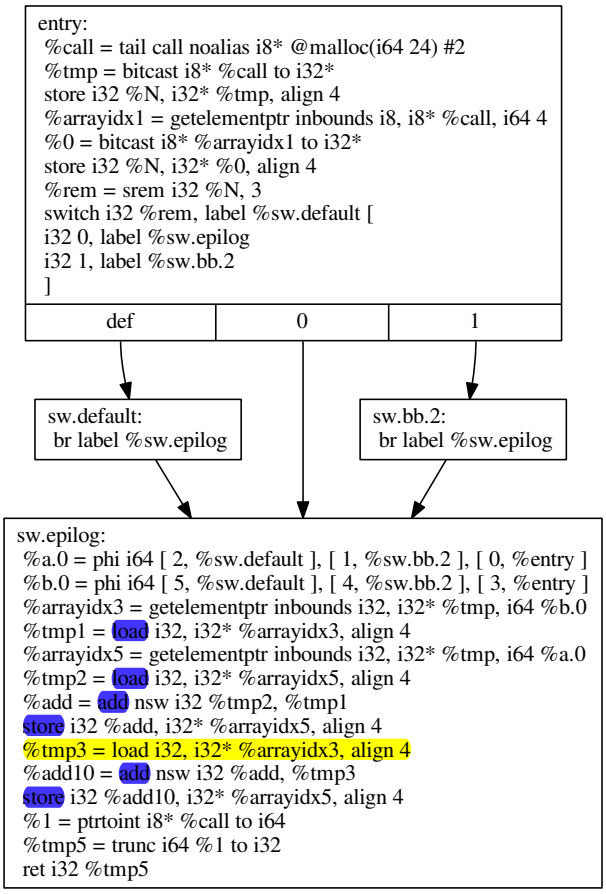
Let us now feed this example to LLVM to analyze and possibly optimize it. We shall study the output of both O3 optimizations and our analysis sraa with the GVN pass (we could also use rbaa instead). Note that the relation ($\mathtt{a} < \mathtt{b}$) always holds at memory accesses v[a] and v[b]. Our analysis sraa is able to disambiguate these pointers via the LTC. Figure II.4.12a shows the CFG of the O3 optimized version in the LLVM intermediate representation. Figure II.4.12b shows the optimized version we get using our alias analysis. In both versions the second load of v[a] is removed. However, only using our sraa analysis allows GVN to remove the second load of v[b] (%`tmp3` = `load i32`, `i32`* %`arrayidx3`, `align 4`) which is still present after *all* O3 optimizations were performed but using the *basic* analysis.

In the sequel, we consider bigger programs and present how GVN performs on the LLVM test-suite when combined with our symbolic range analysis of pointers, rbaa. In Figure II.4.13 we compare the number of merged blocks, deleted instructions, simplified instructions, and loads deleted by GVN among O3 passes with and without our analysis rbaa. Enhanced with the sraa analysis, GVN is able to remove more dead instructions for all the benchmarks we were able to run except for ks and anagram. This is also the case for load instructions (not counted among deleted instructions) where the improvement rate is almost 1.5.

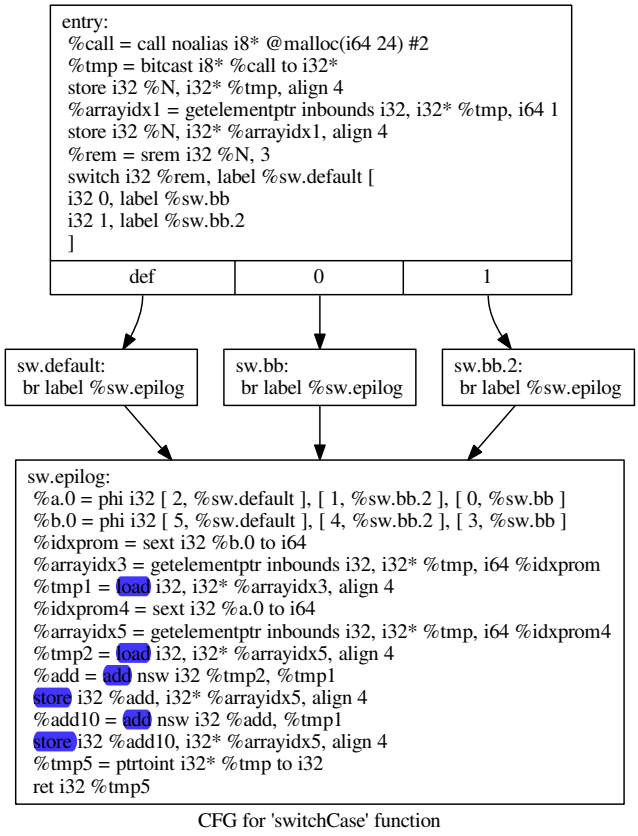
To conclude, our analysis sraa is able to disambiguate more pairs of pointers than the basic alias analysis (basicaa) of LLVM as we show in Section II.1.6. This extra precision, put into work for the GVN optimizations, proves that the compiler still needs more precise alias analyses to achieve better performance.

II.4.4 Conclusion

In this chapter, we adopted another applicability metric to evaluate the alias analysis techniques we introduced in Part II. We explored how these methods improve the quality of some



(a) Program in Figure II.4.9 after O3 optimizations.



(b) Program in Figure II.4.9 after GVN optimization and sraa analysis. Note that the instruction `%tmp3 = load i32, i32* %arrayidx3, align 4` is removed.

Figure II.4.12 – Comparing GVN optimizations on the program of Figure II.4.9 with O3 and O3sraa.

LLVM memory optimizations. Our study reveals some improvements mostly related to pointer arithmetic solving, but also witnesses that the lacks of LLVM optimizations does not come only from the lack of analysis precision. To be performed, optimizations should satisfy a lot of conditions related to memory and execution safety. These conditions are as important as dependency information but not as popular.

Program	#Inst	blockMerged		instDeleted		instSimplified		loadsDeleted	
		O3	O3rbba	O3	O3rbba	O3	O3rbba	O3	O3rbba
cdecl	3101	2	2	117	**120**	93	96	0	0
football	9703	3	3	1014	**1051**	11	11	2	2
simulator	9681	1	1	44	**509**	8	8	15	**18**
assembler	4259	3	3	141	**146**	1	1	6	6
loader	2267	1	1	127	**130**	2	2	1	1
gnugo	3483	4	4	204	**246**	1	1	13	13
unix-tbl	9428	4	X	605	X	3	3	X	X
agrep	11946	4	X	748	X	5	5	21	X
fixoutput	389	0	0	2	**3**	2	2	0	1
compiler	2870	0	0	73	**87**	3	3	1	**7**
bison	12946	4	4	792	**808**	1	1	1	**3**
archie-client	5025	0	0	273	**279**	1	1	2	**6**
TimberWolfMC	69496	3	X	5504	X	15	15	31	X
allroots	422	0	0	33	**37**	17	**21**	1	1
unix-smail	4129	1	X	180	X	133	X	0	X
plot2fig	1240	0	0	79	**83**	2	2	0	**2**
bc	13620	3	3	769	**797**	3	3	25	**31**
yacr2	5447	3	3	305	**318**	6	6	1	**7**
ks	1418	2	2	93	93	64	64	0	0
anagram	796	0	0	29	29	16	16	0	0
ft	1733	1	1	163	**172**	2	2	1	**3**

Figure II.4.13 – Optimizations performed by GVN for O3 and rbaa within O3 (O3rbaa).

Conclusion and Future Work

Summary

In this thesis, we have been interested in analyzing relations between program variables to efficiently handle memory accesses that are performed by pointers in C-like languages. Our main goal was to design as precise static analyses as possible to help compilers optimize programs in terms of locality and runtime. Therefore, all our contributions were developed with scalability in mind. We have achieved this goal thanks to two fundamentals. The first one is the abstract interpretation framework and its maturity that have allowed us to design correct-by-design analyses based on over-approximations. The second fundamental is sparsity: a way to build scalable analyses without sacrificing precision and correctness.

In Part I of this manuscript, we have started by introducing static and pointer analysis notions and presented the program representation we use. The lack of precision we noticed while experimenting with some of the C compilers with different architectures has further motivated this thesis. The state-of-the-art in the pointer analysis field is abundant. In this part, we have also studied the various techniques proposed in the literature and exposed their limitations. This study has revealed a need for more precise and especially scalable analyses to be embedded in the compiler construction.

To recover part of the missing precision and lack of efficiency, we have developed in Part II three alias analysis techniques to solve pointer arithmetic. Our first contribution is an algorithm that disambiguates pointers based on an off-the-shelf symbolic range analysis for integer variables. This algorithm was implemented on top on the LLVM compiler. We were actually able to disambiguate 1.35x more queries than the alias analyses currently available in LLVM. On the complexity of this analysis, we have shown and proved its linear behavior with regard to the number of instructions for programs we ran. Our second contribution is a semi-relational analysis. We have provided an algorithm to disambiguate pointers based on less-than relations we collect and propagate from the analyzed programs. We have shown that generated constraints are linear on the number of instructions which makes our analysis run in reasonable time. Finally, we have extended these two techniques into a third one. The algorithm we have introduced combines the range and less-than tests and deals also with non-related pointers. Results we have presented reveal a better precision for the combined and adapted method. Although this technique is more expensive than the two others, we have shown that experimentally it is still competitive. The analyses we have presented in this part were implemented on top of the LLVM compiler as dynamic passes. To further evaluate our analyses and the LLVM program transformations, we have studied their impact on compiler optimizations. We have measured the number of optimizations performed by some of the transform passes when one of our alias analyses is run. These passes are alias analysis clients. Through these experiments, we have shown that our techniques do not only outperform LLVM analyses in terms of number of disambiguated pointers, but also enable more optimizations thanks to the extra precision they provide. We have also studied these optimizations and found out that, along with providing them with more precise alias information,

LLVM optimization algorithms may need some enhancement related to the way they measure transformation side effects and how they perform checks for optimizing opportunities.

Toward Precise Alias Analyses in Production Compilers

As a future short term work, we intend to upstream the alias analysis techniques we developed in this thesis to the LLVM compiler distribution. To reach this point, we should make sure that the pre-analyses and transformations we borrow from previous work to obtain the suitable representation are working efficiently. We mean by efficiency the way the algorithms are implemented including the data structures used, and also the need to prove the correctness including corner case testing. For instance, this concerns LLVM passes we use to transform programs to e-SSA form and the pre-analyses we run to build the symbolic or numeric range analyses for program integer variables. Furthermore, we believe that our less-than check could be implemented differently to achieve a better runtime. Since the vast of abstract states ended up empty or with a few elements, we may find a way to track only interesting relations that would enable us disambiguate related pointers.

Besides program optimization and data locality, pointer analyses are useful for program verification like eliminating array out of bound checks or checking the safety of pointer dereference. Another promising application for pointer analysis in the scope of program verification is related to *Satisfiability Modulo Theories* (SMT) solver applications. SMT solvers, used in testing and constructing programs [NL14], are clever tools to prove program properties but face scalability issues. Given a large program and a complex formula to prove, they often fail to report an answer within a reasonable amount of time. One challenge to make such tool scaling is simplifying formulas arising from program analyses. Since pointer analyses are the key for designing memory models [WBW17], it would be interesting to develop this line of research and find out whether the analyses we designed in this thesis are as beneficial for verification as they are for optimization.

Finally, in the long term, an interesting future for alias analysis evaluation would be to set up a tool to measure the optimal non aliasing rates. Such a tool should, most probably, rely on expensive exhaustive techniques to answer the question: "How many pairs of pointers do not really alias in the analyzed program?". Until now, evaluating an analysis under the metric of number of disambiguated pairs of pointer requires a comparison with the existent implemented techniques. Assuming the analyses are correct, we just claim that the higher the number of "no alias" answers, the better. Yet, we ignore how far these analyses are from the real static no alias rates and whether more precision is worth investigation on not.

Pointer Analyses for C-based Parallel Languages

Parallel programming languages like openMP, openCL, or CUDA are based on the C language. They support threading and explicitly handle memory accesses. A major challenge when programming *Graphics Processing Unit* (GPU) architectures consists in balancing parallelism with locality to achieve the proper trade-off for performance. Proper data placement, whether manual (scratchpad[4]) or automatic (cache), is in fact, tightly linked with thread scheduling. In the sequel, we give some premises of using our alias analyses for *General-purpose computing on graphics processing Unit* (GPGPU). Our ultimate goal would be to infer bounds the set of mem-

[4] Also called "shared memory". This memory offers the highest speed access for threads within a block to access a common data.

ory locations accessed by a thread block. This analysis will be a driver for further optimizations like code transformation to use software-managed caches, using either per-thread data copies or bulk memory copy commands.

For GPUs, we want to transform a given *kernel* code (code run on the GPU) that directly accesses to global memory into a code that first copies (once, in parallel, row by row) from global memory to a scratchpad, then accesses it many times.

The desired analysis should provide the developer with memory access bounds for each `block` and for each 2D array in order to: (i) compare the number of thread accesses to the number of global memory slots accessed, (ii) measure the shared memory allocation required to create the buffer and whether it is suitable for the scratchpad.

In the sequel, we give some premises of using the alias analyses we presented in this thesis to safely evaluate data-set volumes of GPU applications.

Memory optimization for GPUs. To motivate the need for a new memory analysis for GPU kernels, we consider the program in Figure II.4.14. The figure shows a non optimized variant of `jacobi` algorithm run on the device (GPU) side. Each thread reads all its neighbors from the `src` array and stores the sum in the `dst` array.

The idea of using alias analysis comes from the need to bound offset ranges of memory locations accessed by these threads. In this program, this is performed through `src_0`, `src_1`, and `src_2` defined from the base pointer `src`. Using a range-based alias analysis, we may know that all threads of a given block access the sub-matrix of two dimensions from $blockIdx.y \times 16 - 1$ to $16(blockIdx.y + 1)$ and from $blockIdx.x \times 16 - 1$ to $16(blockIdx.x + 1)$. This would allow us to measure the data volume accessed based on `src` which is 16×16. Therefore, we could build the program in Figure II.4.15. In this optimized version, we use a shared *buffer*, a copy of data read a couple of times from the global memory by different threads. Instead of reading data from the global memory, now each thread reads the local *buffer* to perform its computations. This data reuse should make the program run faster thanks to the fewer global memory accesses performed.

```
__global__ void stencil_kernel(float * src, float * dst, int N)
{
    int tidx = blockIdx.x * 16 + threadIdx.x;
    int tidy = blockIdx.y * 16 + threadIdx.y;

    float acc = 0.f;

    float* src_0 = src + (tidy-1) * N;
    float* src_1 = src + tidy * N;
    float* src_2 = src + (tidy+1) * N;

    // Unrolled loop
    acc += src_0[tidx-1]; //1: src_01 = src_0 + tidx - 1
    acc += src_0[tidx  ]; //2: src_02 = src_0 + tidx
    acc += src_0[tidx+1]; //3: src_03 = src_0 + tidx + 1
    acc += src_1[tidx-1]; //4: src_14 = src_1 + tidx - 1
    acc += src_1[tidx  ]; //5: src_15 = src_1 + tidx
    acc += src_1[tidx+1]; //6: src_16 = src_1 + tidx + 1
    acc += src_2[tidx-1]; //7: src_27 = src_2 + tidx - 1
    acc += src_2[tidx  ]; //8: src_28 = src_2 + tidx
    acc += src_2[tidx+1]; //9: src_29 = src_2 + tidx + 1
    dst[tidy * N + tidx] = acc
}
```

Figure II.4.14 – Non-optimized `jacobi` algorithm written in CUDA.

```
__global__ void stencil_kernel(float * src, float * dst)
{
    int tidx = blockIdx.x * 14 + threadIdx.x - 1;
    int tidy = blockIdx.y * 14 + threadIdx.y - 1;

    __shared__ float buffer[16][16];

    if(in_range(tidx) && in_range(tidy)) {
        buffer[threadIdx.y][threadIdx.x] = src[tidy  ][tidx  ];
    }
    else {
        // handle boundary conditions
    }

    __syncthreads();

    if(threadIdx.x >= 1 && threadIdx.x < 15 && threadIdx.y >= 1 && threadIdx.
        y < 15)
    {
        int tidx_local = threadIdx.x;
        int tidy_local = threadIdx.y;
        float acc = 0.f;
        // Unrolled loop
        acc += buffer[tidy_local-1][tidx_local-1];    // 1
        acc += ...
        acc += buffer[tidy_local+1][tidx_local+1];    // 9
        dst[tidy * N + tidx] = acc;
    }
}
```

Figure II.4.15 – Program in Figure II.4.14 with shared memory accesses. The "`__shared__`" keyword makes the declared variable resident in shared memory.

Problem statement. We intend to develop an analysis based on our pointer analysis algorithms to bound the number of memory accesses to the global memory that are performed in the kernel. The output of such an analysis would be the number of memory slots accessed by each base pointer in the kernel programs regarding the two dimensions. We give below the steps that should be performed:

1. Grouping pointers into digraphs,
2. Collecting symbolic ranges for integer variables,
3. Collecting symbolic modular ranges for integer variables,
4. Computing memory bounds for array accesses.

The first and second step were detailed in Section II.3.3 and Section II.1.3.3, and the fourth step consists in computing the final bounds based on the symbolic and symbolic modular ranges associated to base pointers determined in step 1 by pointer dependence digraph. However, our analysis still lack precision to determine symbolic ranges with "holes" that correspond to the analyzed array dimension. We want to find out an overaproximation of the accessed slots modulo the array size parameter. In the literature, this abstract domain is called the *wrapping-intervals* [Neu93]. The wrapping intervals have been essentially used to analyze integer variables for verification purposes such as discovering array and arithmetic overflows [Min12; Mas93]. However, they were designed for arithmetic numeric operations which makes them unadapted for symbols.

Thesis Context and Collaborations

In this thesis, I had the chance to collaborate with researchers from different universities and research laboratories. Since the beginning of my PhD, I have been working with my PhD advisor **Laure Gonnord**. Together, we wrote the research report [MG15] in 2015. I have also been

a member of the project *PROSPIEL* (Profiling and specialization for locality) with **Sylvain Collange** and **Fernando Magno Quintão Pereira** as the *principal investigators*. Within this project, I had the opportunity to have many collaborations with Laure Gonnord, Fernando Pereira, and his graduated and undergraduate students at the *Compilers Lab* of the *Federal University of Minas Gerais*, Brazil. Indeed, together, we published the contributions of Chapter II.1, Chapter II.2, and Chapter II.3. Below, I give an estimation to each of my collaboration rates:

- Symbolic Range Analysis of Pointers
  1. Implementation + Validation: 0%
  2. Formalisation + Development: 60%
- Pointer Disambiguation via Strict Inequalities
  1. Implementation + Validation: 40%
  2. Formalisation + Development: 25%
- Combining Range and Inequality Information for Pointer Disambiguation
  1. Implementation + Validation: 50%
  2. Formalisation + Development: 60%

With Sylvain Collange, I learned more about GPU architectures and started the memory analysis for GPGPU introduced in the future work.

References

[Aho+06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2006.

[Alv+15] Péricles Alves, Fabian Gruber, Johannes Doerfert, Alexandros Lamprineas, Tobias Grosser, Fabrice Rastello, and Fernando Magno Quintão Pereira. "Runtime Pointer Disambiguation". In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*. ACM, 2015, pp. 589–606.

[Ana97] C. Scott Ananian. "The Static Single Information Form". MA thesis. Princeton University, 1997.

[And94] Lars Ole Andersen. "Program Analysis and Specialization for the C Programming Language". PhD thesis. DIKU, University of Copenhagen, 1994.

[AWZ88] B. Alpern, M. N. Wegman, and F. K. Zadeck. "Detecting equality of variables in programs". In: *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988*. ACM, 1988, pp. 1–11.

[Bay72] Rudolf Bayer. "Symmetric Binary B-Trees: Data Structure and Maintenance Algorithms". In: *Acta Inf.* 1 (1972), pp. 290–306.

[BE94] William Blume and Rudolf Eigenmann. "Symbolic Range Propagation". In: *Proceedings of IPPS '95, The 9th International Parallel Processing Symposium, April 25-28, 1995, Santa Barbara, California, USA*. 1994, pp. 357–363.

[BGS00] Rastislav Bodík, Rajiv Gupta, and Vivek Sarkar. "ABCD: eliminating array bounds checks on demand". In: *Proceedings of Conference on Programming Language Design and Implementation PLDI, Vancouver, Britith Columbia, Canada*. ACM, 2000, pp. 321–333.

[Bla+03] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. "A static analyzer for large safety-critical software". In: *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation PLDI, San Diego, California, USA, June 9-11, 2003*. 2003, pp. 196–207.

[Bou93] François Bourdoncle. "Efficient chaotic iteration strategies with widenings". In: *Formal Methods in Programming and Their Applications*. Ed. by Dines Bjørner, Manfred Broy, and Igor V. Pottosin. Vol. 795. lncs. springer, 1993, pp. 128–141.

[BR04] Gogul Balakrishnan and Thomas Reps. "Analyzing memory accesses in x86 executables". In: *Compiler Construction, 13th International Conference CC, Held as Part of the Joint European Conferences on Theory and Practice of Software ETAPS, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*. Springer, 2004, pp. 5–23.

[BS16] George Balatsouras and Yannis Smaragdakis. "Structure-Sensitive Points-To Analysis for C and C++". In: *Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings*. 2016, pp. 84–104.

[Bud+02] Zoran Budimlic, Keith D. Cooper, Timothy J. Harvey, Ken Kennedy, Timothy S. Oberg, and Steven W. Reeves. "Fast copy coalescing and live-range identification". In: *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation PLDI, Berlin, Germany, June 17-19, 2002*. ACM, 2002, pp. 25–32.

[Cal+06] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. "Beyond Reachability: Shape Abstraction in the Presence of Pointer Arithmetic". In: *Static Analysis, 13th International Symposium SAS, Seoul, Korea, August 29-31, 2006, Proceedings*. Springer, 2006, pp. 182–203.

[CC77] Patrick Cousot and Radhia Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". In: *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages POPL, Los Angeles, California, USA, January 1977*. ACM, 1977, pp. 238–252.

[CC92] Patrick Cousot and Radhia Cousot. "Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation". In: *Programming Language Implementation and Logic Programming, 4th International Symposium, PLILP'92, Leuven, Belgium, August 26-28, 1992, Proceedings*. 1992, pp. 269–295.

[CCF91] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. "Automatic Construction of Sparse Data Flow Evaluation Graphs". In: *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages POPL, Orlando, Florida, USA, January 21-23, 1991*. ACM, 1991, pp. 55–66.

[CE81] Edmund M. Clarke and E. Allen Emerson. "Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic". In: *Logics of Programs, Workshop, Yorktown Heights, New York, May 1981*. 1981, pp. 52–71.

[CH78] Patrick Cousot and Nicolas Halbwachs. "Automatic Discovery of Linear Restraints Among Variables of a Program". In: *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages POPL, Tucson, Arizona, USA, January 1978*. ACM, 1978, pp. 84–96.

[Che16] Jia Chen. *CFL Alias Analysis*. Google's Summer of Code Report. 2016.

[CIA12] Bill Campbell, Swami Iyer, and Bahar Akbal-Delibas. *Introduction to Compiler Construction in a Java World*. Chapman and Hall/CRC, 2012.

[Cyt+89] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. "An Efficient Method of Computing Static Single Assignment Form". In: *POPL*. 1989, pp. 25–35.

[Cyt+91] Ron Cytron, Jeanne Ferrante, Barry Rosen, Mark Wegman, and Kenneth Zadeck. "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph". In: *ACM Transactions On Programming Languages and Systems* 13.4 (1991), pp. 451–490.

[DF16] Delphine Demange and Yon Fernandez De Retana. "Mechanizing conventional SSA for a verified destruction with coalescing". In: *25th International Conference on Compiler Construction*. Barcelona, Spain, Mar. 2016. URL: https://hal.archives-ouvertes.fr/hal-01378393.

[Ema93] Maryam Emami. "A practical interprocedural alias analysis for an optimizing/parallelizing C compiler". MA thesis. Montreal: McGill University, 1993.

[Eng+04] Robert A. van Engelen, J. Birch, Y. Shou, B. Walsh, and Kyle A. Gallivan. "A Unified Framework for Nonlinear Dependence Testing and Symbolic Analysis". In: *ICS*. ACM, 2004, pp. 106–115.

[Ern04] Michael D. Ernst. "Invited Talk Static and dynamic analysis: synergy and duality". In: *Proceedings of the 2004 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering PASTE, Washington, DC, USA, June 7-8, 2004*. ACM, 2004, p. 35.

[Fäh+98] Manuel Fähndrich, Jeffrey S. Foster, Zhendong Su, and Alexander Aiken. "Partial Online Cycle Elimination in Inclusion Constraint Graphs". In: *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation PLDI, Montreal, Canada, June 17-19, 1998*. 1998, pp. 85–96.

[FL11] Paul Feautrier and Christian Lengauer. "The Polyhedron Model". In: *Encyclopedia of Parallel Programming*. Ed. by David Padua. Springer, 2011.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. "The program dependence graph and its use in optimization". In: *ACM Transactions on Programming Languages and Systems TOPLAS* 9.3 (1987), pp. 319–349.

[FP11] Łukasz Fronc and Franck Pommereau. "Towards a certified Petri net model-checker". In: *Proceedings of APLAS'11*. Vol. 7078. LNCS. Springer, Jan. 2011, pp. 322–336.

[Guo+06] Bolei Guo, Youfeng Wu, Cheng Wang, Matthew J. Bridges, Guilherme Ottoni, Neil Vachharajani, Jonathan Chang, and David I. August. "Selective Runtime Memory Disambiguation in a Dynamic Binary Translator". In: *Proceedings of European Joint Conferences on Theory and Practice of Software*. Springer, 2006, pp. 65–79.

[GZH93] Dirk Grunwald, Benjamin Zorn, and Robert Henderson. "Improving the Cache Locality of Memory Allocation". In: *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation PLDI, Albuquerque, New Mexico, USA, June 23-25, 1993*. ACM, 1993, pp. 177–186.

[Haj78] Jan Hajek. "Automatically verified data transfer protocol". In: *4th International Conference on Computer Communication (ICCC)*. 1978, pp. 749–756.

[Hen06] John L. Henning. "SPEC CPU2006 benchmark descriptions". In: *SIGARCH Comput. Archit. News* 34.4 (2006), pp. 1–17.

[HGG06] Sebastian Hack, Daniel Grund, and Gerhard Goos. "Register allocation for programs in SSA-form". In: *Compiler Construction, 15th International Conference CC, Held as Part of the Joint European Conferences on Theory and Practice of Software ETAPS, Vienna, Austria, March 30-31, 2006, Proceedings*. Springer-Verlag, 2006, pp. 247–262.

[HH98] Rebecca Hasti and Susan Horwitz. "Using Static Single Assignment Form to Improve Flow-Insensitive Pointer Analysis". In: *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation PLDI, Montreal, Canada, June 17-19, 1998*. 1998, pp. 97–105.

[HL07] Ben Hardekopf and Calvin Lin. "The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code". In: *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation PLDI, San Diego, California, USA, June 10-13, 2007*. ACM, 2007, pp. 290–299.

[HL11] Ben Hardekopf and Calvin Lin. "Flow-Sensitive Pointer Analysis for Millions of Lines of Code". In: *Proceedings of the CGO 2011, The 9th International Symposium on Code Generation and Optimization CGO, Chamonix, France, April 2-6, 2011.* 2011, pp. 265–280.

[Hol97] Gerard J. Holzmann. "The Model Checker SPIN". In: *IEEE Transactions on Software Engineering* 23.5 (1997), pp. 279–295.

[Hor97] Susan Horwitz. "Precise Flow-Insensitive May-Alias Analysis is NP-Hard". In: *ACM Transactions on Software Engineering* 19.1 (1997), pp. 1–6.

[HT01] Nevin Heintze and Olivier Tardieu. "Ultra-fast Aliasing Analysis using CLA: A Million Lines of C Code in a Second". In: *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation PLDI, Snowbird, Utah, USA, June 20-22, 2001.* 2001, pp. 254–263.

[ISO11] ISO-Standard. *9899 - The C Programming Language.* 2011.

[ISO99] ISO-Standard. *9899 - The C Programming Language.* 1999.

[JM82] Neil D. Jones and Steven S. Muchnick. "A Flexible Approach to Interprocedural Data Flow Analysis and Programs with Recursive Data Structures". In: *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 1982.* ACM, 1982, pp. 66–74.

[LA03] Chris Lattner and Vikram Adve. "Architecture for a Next-Generation GCC". In: *Proc. First Annual GCC Developers' Summit.* Ottawa, Canada, May 2003.

[LA04] Chris Lattner and Vikram S. Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *2nd IEEE / ACM International Symposium on Code Generation and Optimization CGO, 20-24 March 2004, San Jose, CA, USA.* IEEE, 2004, pp. 75–88.

[Lan92] William Landi. "Undecidability of Static Analysis". In: *Letters on Programming Languages and Systems LOPLAS* 1.4 (1992), pp. 323–337.

[LCR15] Huisong Li, Bor-Yuh Evan Chang, and Xavier Rival. "Shape Analysis for Unstructured Sharing". In: *Static Analysis - 22nd International Symposium SAS, Saint-Malo, France, September 9-11, 2015, Proceedings.* 2015.

[LF08] Fancesco Logozzo and Manuel Fahndrich. "Pentagons: a Weakly Relational Abstract Domain for the Efficient Validation of Array Accesses". In: *Proceedings of the 2008 ACM Symposium on Applied Computing (SAC), Fortaleza, Ceara, Brazil, March 16-20, 2008.* ACM, 2008, pp. 184–188.

[LF10] Francesco Logozzo and Manuel Fähndrich. "Pentagons: A weakly relational abstract domain for the efficient validation of array accesses". In: *Science of Computer Programming* 75.9 (2010), pp. 796–807.

[LH06] Ondrej Lhoták and Laurie J. Hendren. "Context-Sensitive Points-to Analysis: Is It Worth It?" In: *Compiler Construction, 15th International Conference CC, Held as Part of the Joint European Conferences on Theory and Practice of Software ETAPS, Vienna, Austria, March 30-31, 2006, Proceedings.* 2006, pp. 47–64.

[LPH05] Donglin Liang, Maikel Pennings, and Mary Jean Harrold. "Evaluating the impact of context-sensitivity on Andersen's algorithm for Java programs". In: *Proceedings of the 2005 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE'05, Lisbon, Portugal, September 5-6, 2005.* 2005, pp. 6–12.

[LR91] William Landi and Barbara G. Ryder. "Pointer-Induced Aliasing: A Problem Classification". In: *ACM Transactions on Programming Languages and Systems TOPLAS*. 1991, pp. 93–103.

[Mas93] François Masdupuy. "Semantic Analysis of Interval Congruences". In: *Formal Methods in Programming and Their Applications, International Conference, Akademgorodok, Novosibirsk, Russia, June 28 - July 2, 1993, Proceedings*. 1993, pp. 142–155.

[Mem+16] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N. M. Watson, and Peter Sewell. "Into the Depths of C: Elaborating the De Facto Standards". In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation PLDI , Santa Barbara, CA, USA, June 13-17, 2016*. ACM, 2016, pp. 1–15.

[Min06] Antoine Miné. "The octagon abstract domain". In: *Higher Order Symbol. Comput.* 19 (1 2006), pp. 31–100.

[Min07] Antoine Miné. "Field-Sensitive Value Analysis of Embedded C Programs with Union Types and Pointer Arithmetics". In: *CoRR* abs/cs/0703074 (2007).

[Min12] Antoine Miné. "Abstract domains for bit-level machine integer and floating-point operations". In: *4th International Workshop on invariant Generation, EWING 2012, Jun 2012, Manchester, United Kingdom*. 2012.

[Moc+01] Markus Mock, Manuvir Das, Craig Chambers, and Susan J. Eggers. "Dynamic points-to sets: a comparison with static analyses and potential applications in program understanding and optimization". In: *Proceedings of Programming and Analysis for Software Tools and Engineering*. ACM, 2001, pp. 66–72.

[Moc03] Markus Mock. "Dynamic Analysis from the Bottom Up". In: *Proceedings of International Conference on Software Engineering*. 2003.

[MP01] Vincenzo Martena and Pierluigi San Pietro. "Alias Analysis by Means of a Model Checker". In: *Proceedings of Compiler Construction*. Springer, 2001, pp. 3–19.

[Naz+14] Henrique Nazaré, Izabela Maffra, Willer Santos, Leonardo Barbosa, Laure Gonnord, and Fernando Magno Quintão Pereira. "Validation of memory accesses through symbolic analyses". In: *Proceedings of Object-Oriented Programming, Systems, Languages & Applications*. ACM, 2014, pp. 791–809.

[Neu93] Arnold Neumaier. "The Wrapping Effect, Ellipsoid Arithmetic, Stability and Confidence Regions". In: *Validation Numerics*. Ed. by R. Albrecht, G. Alefeld, and H. J. Stetter. Springer Vienna, 1993, pp. 175–190.

[NG15] Vaivaswatha Nagaraj and R. Govindarajan. "Approximating flow-sensitive pointer analysis using frequent itemset mining". In: *Proceedings of International Symposium on Code Generation and Optimization*. IEEE Computer Society, 2015, pp. 225–234.

[NL14] Bjørner Nikolaj and de Moura Leonardo. *Applications of SMT solvers to Program Verification*. http://fm.csl.sri.com/SSFT14/smt-application-chapter.pdf. Online; accessed 21 June 2017. 2014.

[NNH05] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of program analysis*. Springer, 2005.

[Nuu95] Esko Nuutila. "Efficient Transitive Closure Computation in Large Digraphs". In: *Acta Polytechnica Scandinavia: Math Computer Engineering* 74 (1995), pp. 1–124.

[Oh+12] Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, and Kwangkeun Yi. "Design and implementation of sparse global analyses for C-like languages". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI, Beijing, China - June 11 - 16, 2012*. 2012, pp. 229–238.

[Oh+14] Hakjoo Oh, Wonchan Lee, Kihong Heo, Hongseok Yang, and Kwangkeun Yi. "Selective context-sensitivity guided by impact pre-analysis". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*. ACM, 2014, p. 49.

[PB09] Fernando Magno Quintao Pereira and Daniel Berlin. "Wave Propagation and Deep Propagation for Pointer Analysis". In: *Proceedings of the CGO 2009, The Seventh International Symposium on Code Generation and Optimization CGO, Seattle, Washington, USA, March 22-25, 2009*. IEEE, 2009, pp. 126–135.

[PKH04] David J. Pearce, Paul H. J. Kelly, and Chris Hankin. "Efficient field-sensitive pointer analysis for C". In: *Proceedings of the 2004 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering PASTE, Washington, DC, USA, June 7-8, 2004*. 2004, pp. 37–42.

[Pop06] Sebastian Pop. "The SSA Representation Framework: Semantics, Analyses and GCC Implementation". Theses. École Nationale Supérieure des Mines de Paris, Dec. 2006. URL: https://pastel.archives-ouvertes.fr/pastel-00002281.

[PP09] Fernando Magno Quintão Pereira and Jens Palsberg. "SSA Elimination after Register Allocation". In: *Compiler Construction, 18th International Conference CC, Held as Part of the Joint European Conferences on Theory and Practice of Software ETAPS, York, UK, March 22-29, 2009. Proceedings*. 2009, pp. 158–173.

[Ram94] G. Ramalingam. "The Undecidability of Aliasing". In: *ACM Transactions on Programming Languages and Systems TOPLAS* 16.5 (1994), pp. 1467–1471.

[Ras16] Fabrice Rastello. *Static Single Assignment Book*. Springer, 2016.

[RCP13] Raphael Ernani Rodrigues, Victor Hugo Sperle Campos, and Fernando Magno Quintao Pereira. "A Fast and Low Overhead Technique to Secure Programs Against Integer Overflows". In: *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization CGO, Shenzhen, China, February 23-27, 2013*. ACM, 2013.

[RQA16] Bruno Rodrigues, Fernando Magno Quintão Pereira, and Diego F. Aranha. "Sparse Representation of Implicit Flows with Applications to Side-channel Detection". In: *Proceedings of the 25th International Conference on Compiler Construction CC, Barcelona, Spain, March 12-18, 2016*. ACM, 2016, pp. 110–120.

[RR00] Radu Rugina and Martin C. Rinard. "Symbolic bounds analysis of pointers, array indices, and accessed memory regions". In: *Proceedings of Programming Language Design and Implementation*. ACM, 2000, pp. 182–195.

[RR05] Radu Rugina and Martin C. Rinard. "Symbolic bounds analysis of pointers, array indices, and accessed memory regions". In: *ACM Transactions on Programming Languages and Systems TOPLAS* 27.2 (2005), pp. 185–235.

[RRH02] Silvius Rus, Lawrence Rauchwerger, and Jay Hoeflinger. "Hybrid analysis: static & dynamic memory reference analysis". In: *Proceedings of International Conference on Supercomputing*. ACM, 2002, pp. 274–284.

[Ryd+01] Barbara G. Ryder, William A. Landi, Philip A. Stocks, Sean Zhang, and Rita Altucher. "A Schema for Interprocedural Modification Side-effect Analysis with Pointer Aliasing". In: *ACM Transactions on Programming Languages and Systems TOPLAS* 23.2 (2001), pp. 105–186.

[SH97] Marc Shapiro and Susan Horwitz. "Fast and Accurate Flow-Insensitive Points-To Analysis". In: *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997*. 1997, pp. 1–14.

[Sin06] Jeremy Singer. "Static Program Analysis Based on Virtual Register Renaming". PhD thesis. University of Cambridge, 2006.

[SRW98] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. "Solving Shape-Analysis Problems in Languages with Destructive Updating". In: *ACM Transactions on Programming Languages and Systems TOPLAS* 20.1 (1998), pp. 1–50.

[Sta09] Stefan Staiger-Stöhr. *Implementing Sparse Flow-Sensitive Andersen Analysis*. Research Report. Feb. 2009.

[Ste96] Bjarne Steensgaard. "Points-to Analysis in Almost Linear Time". In: *Proceedings of Symposium on Principles of Programming Languages*. ACM Press, 1996, pp. 32–41.

[Str+14] Thomas Ströder, Jürgen Giesl, Marc Brockschmidt, Florian Frohn, Carsten Fuhs, Jera Hensel, and Peter Schneider-Kamp. "Proving Termination and Memory Safety for Programs with Pointer Arithmetic". In: *Proceedings of International Joint Conference on Automated Reasoning*. Springer, 2014, pp. 208–223.

[Sui+16] Yulei Sui, XIaokang Fan, Hao Zhou, and Jingling Xue. "Loop-oriented Array and Field-sensitive Pointer Analysis for Automatic SIMD Vectorization". In: *Proceedings of the 17th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, Tools, and Theory for Embedded Systems LCTES, Santa Barbara, CA, USA, June 13 - 14, 2016*. ACM, 2016, pp. 41–51.

[Sur+14] Rishi Surendran, Rajkishore Barik, Jisheng Zhao, and Vivek Sarkar. "Inter-iteration Scalar Replacement Using Array SSA Form". In: *Compiler Construction - 23rd International Conference CC, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*. Springer, 2014, pp. 40–60.

[Tav+14] André Luiz Camargos Tavares, Benoit Boissinot, Fernando Magno Quintão Pereira, and Fabrice Rastello. "Parameterized Construction of Program Representations for Sparse Dataflow Analyses". In: *Compiler Construction - 23rd International Conference CC, Held as Part of the European Joint Conferences on Theory and Practice of Software ETAPS, Grenoble, France, April 5-13, 2014. Proceedings*. Springer, 2014, pp. 2–21.

[VB04] Arnaud Venet and Guillaume P. Brat. "Precise and efficient static array bound checking for large embedded C programs". In: *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation PLDI, Washington, DC, USA, June 9-11, 2004*. 2004, pp. 231–242.

[VR01] Frédéric Vivien and Martin C. Rinard. "Incrementalized Pointer and Escape Analysis". In: *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22, 2001*. 2001, pp. 35–46.

[WBW17] Wei Wang, Clark Barrett, and Thomas Wies. "Partitioned Memory Models for Program Analysis". In: *Verification, Model Checking, and Abstract Interpretation - 18th International Conference, VMCAI 2017, Paris, France, January 15-17, 2017, Proceedings*. 2017, pp. 539–558.

[Wes89] Colin H. West. "An automated technique for communications protocol validation". In: *IEEE Transactions On Communications* (1989), pp. 1271–1275.

[WL04] John Whaley and Monica S. Lam. "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, Washington, DC, USA, June 9-11, 2004*. 2004, pp. 131–144.

[Wol96] Michael Wolfe. *High Performance Compilers for Parallel Computing*. 1st. Adison-Wesley, 1996.

[Wu+13] Jingyue Wu, Gang Hu, Yang Tang, and Junfeng Yang. "Effective dynamic detection of alias analysis errors". In: *Proceedings of European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*. ACM, 2013, pp. 279–289.

[Yan+11] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. "Finding and Understanding Bugs in C Compilers". In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation PLDI, San Jose, CA, USA, June 4-8, 2011*. ACM, 2011, pp. 283–294.

[YH04] Suan Hsi Yong and Susan Horwitz. "Pointer-Range Analysis". In: *Proceedings of Static Analysis Symposium*. Springer, 2004, pp. 133–148.

[YP15] Tomofumi Yuki and Louis-Noël Pouchet. *PolyBench 4.0*. 2015.

[YSX14] Sen Ye, Yulei Sui, and Jingling Xue. "Region-Based Selective Flow-Sensitive Pointer Analysis". In: *Proceedings of Static Analysis Symposium*. Springer, 2014, pp. 319–336.

[YXR11] D. Yan, G. Xu, and A. Rountev. "Demand-Driven Context-Sensitive Alias Analysis for Java". In: *Proceedings of International Symposium on Software Testing and Analysis*. ACM, 2011, pp. 155–165.

[Zha+13a] Qirun Zhang, Michael R. Lyu, Hao Yuan, and Zhendong Su. "Fast Algorithms for Dyck-CFL-reachability with Applications to Alias Analysis". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI, Seattle, WA, USA, June 16-19, 2013*. ACM, 2013, pp. 435–446.

[Zha+13b] Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. "Formal verification of SSA-based optimizations for LLVM". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI, Seattle, WA, USA, June 16-19, 2013*. ACM, 2013, pp. 175–186.

[Zha+14] Qirun Zhang, Xiao Xiao, Charles Zhang, Hao Yuan, and Zhendong Su. "Efficient subcubic alias analysis for C". In: *OOPSLA*. ACM, 2014, pp. 829–845.

[ZR08] Xin Zheng and Radu Rugina. "Demand-driven Alias Analysis for C". In: *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*. ACM, 2008, pp. 197–208.

[ZRW05] Qin Zhao, Rodric Rabbah, and Weng-Fai Wong. "Dynamic Memory Optimization Using Pool Allocation and Prefetching". In: *SIGARCH Comput. Archit. News* 33.5 (2005), pp. 27–32.

Publications

[Maa+17a] Maroua Maalej, Vitor Paisante, Fernando Magno Quintao Pereira, and Laure Gonnord. "Combining Range and Inequality Information for Pointer Disambiguation". In: *Science of Computer Programming* (2017). URL: https://hal.archives-ouvertes.fr/hal-01625402.

[Maa+17b] Maroua Maalej, Vitor Paisante, Pedro Ramos, Laure Gonnord, and Fernando Magno Quintão Pereira. "Pointer disambiguation via strict inequalities". In: *Proceedings of the 2017 International Symposium on Code Generation and Optimization, CGO 2017, Austin, TX, USA, February 4-8, 2017*. 2017, pp. 134–147. URL: https://hal.archives-ouvertes.fr/hal-01387031/document.

[MG15] Maroua Maalej and Laure Gonnord. *Do we still need new Alias Analyses?* Research Report RR-8812. ENS Lyon ; CNRS ; INRIA, Nov. 2015. URL: https://hal.inria.fr/hal-01228581.

[Pai+16] Vitor Paisante, Maroua Maalej, Leonardo Barbosa, Laure Gonnord, and Fernando Magno Quintão Pereira. "Symbolic Range Analysis of Pointers". In: *Proceedings of the 2016 International Symposium on Code Generation and Optimization CGO, Barcelona, Spain, March 12-18, 2016*. ACM, 2016, pp. 171–181. URL: https://hal.inria.fr/hal-01228928/document.