Implementing Andersen Alias Analysis in LLVM

CHAVAN YOGESH LAXMAN

A Thesis Submitted to

Indian Institute of Technology Hyderabad

In Partial Fulfillment of the Requirements for

The Degree of Master of Technology

भारतीय प्रौद्योगिकी संस्थान हैदराबाद
Indian Institute of Technology Hyderabad

Department of Computer Science and Engineering

June 2015

Declaration

I declare that this written submission represents my ideas in my own words, and where ideas or words of others have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be a cause for disciplinary action by the Institute and can also evoke penal action from the sources that have thus not been properly cited, or from whom proper permission has not been taken when needed.

[illegible]

(Signature)

(CHAVAN YOGESH LAXMAN)

CS13M1012

(Roll No.)

Approval Sheet

This Thesis entitled Implementing Andersen Alias Analysis in LLVM by CHAVAN YOGESH LAXMAN is approved for the degree of Master of Technology from IIT Hyderabad

[illegible]

(M.V.P. Rao) Examiner
Dept. of C.S.E
IITH

[illegible]

(N.R. Arvind) Examiner
Dept. C.S.E
IITH

[illegible]

(Dr. Ramakrishna Updrasta) Adviser
Dept. of Computer Science and Engg
IITH

[illegible]

(Manish Singh) Chairman
Dept. of C.S.E
IITH

Acknowledgements

I would like to express my special appreciation and sincere gratitude to my thesis adviser Dr. Ramakrishna Upadrasta, you have been a tremendous mentor for me. Your constant encouragement, patience and immense knowledge help me a lot. A special thanks to my family.

Dedication

I would like to dedicate this thesis to my loving parents.

Abstract

Alias Analysis information is a prerequisite for most of the program analysis and the quality of this information can greatly boost up the precision and performance of the program. Many recent works in this area have focused on high precision and scalability, but still they lack an efficient and scalable alias analysis. The inclusion based or Andersen style alias analysis is widely used to compute such information. Andersen alias analysis is flow insensitive which make it highly precise and motivate many modern optimizing compilers to deploy it. In this work, we have improved and implemented the Andersen alias analysis in the latest version of LLVM.

Contents

List of Figures

List of Algorithms

List of Tables

Chapter 1

Intoduction

Industrial program codes are getting bigger. Programs with billions of lines of source code are no longer uncommon. Compilation of such huge programs typically takes several hours, even some time days. The scalability of compilation is a minimum requirement for current compiler frameworks like gcc [1] and LLVM [2]. Compilation of a program involves several static analysis which analyze and optimize the program. A static analysis automatically analyzes the program without actually running the program [3] because of static analysis we can achieve faster run time execution, less buggy program and better program understanding. Since static analysis is an inseparable component of a compiler, program analysis designer and compiler writers must try hard to make their analysis as efficient as possible.

Alias Analysis is a static analysis which takes a program as input and compute points-to information about the program. To optimize the program, points-to information is used by static analyzer called as *Clients*. The points-to information is depends on how various program elements are modeled, but modeling these program elements in various way gives different points-to information with varying analysis time and memory requirement (program elements are Known as analysis dimension [4]). Some of the analysis dimensions are, context sensitive analysis, which is about calling context of function, the flow sensitive analysis is nothing but control flow in the program, Modeling of aggregates are covered by field sensitivity. The most important terms in alias analysis are analysis time, memory requirement and precision which are the challenging factors of the alias analysis researcher.

In this work we improve and implement the Andersen alias analysis [5] in LLVM. The Andersen alias analysis is deprecated from the current version of LLVM because of buggy code, less efficiency and less scalability. Andersen alias analysis was present in LLVM 2.6 version, but after this version it was removed.

1.1 What is Alias Analysis?

Alias Analysis (aka Pointer Analysis) is a mechanism to statically determine whether two pointers may point to the same memory locations at runtime. Suppose we have a program p to extract pointer specific information and computes an internal representation of aliasing information present in program, which is nothing but *Points-to* information. A *Client* to the alias analysis then queries this information to obtain answers to its specific queries regarding the alias relationship between pointers.

Several *Points-to* analysis algorithms exist in literature (see a survey in Chapter 3). However, two of the most widely used algorithms are Andersen [5] and Steensgaard [6] alias analysis. In this Andersen alias analysis is based on inclusion of points-to set which is more precise, but less efficient. While in Steensgaard analysis, which is unification based is less precise, but more efficient.

1.2 Clients of Alias Analysis

Alias Analysis is not an optimization in itself, i.e., once alias analysis is run, it does not make the program run faster. A Program transformation called *Client* need to query alias information from alias analysis to create an optimized version of the program. Also, there are some other analysis that don't alter the code, but make use of alias analysis to compute better dataflow information which can be used by other transformation. Examples of transformation clients include, Automated bug correction, Parallelization, Common subexpression elimination. Examples of analysis clients include, Slicing, Shape analysis.

1.3 Issues with Scalable Alias Analysis

The alias analysis causes the effectiveness of a *Client's* optimization [4]. The *Client's* execution time will sharply reduced if alias analysis is highly precise. One direction to make alias analysis more precise is to make it context-sensitive but it blowing up the space and time requirements of analysis, the context sensitive version of Andersen alias analysis [5] runs out of memory if optimization is not done. There have been several attempts to achieve a scalable context sensitive alias analysis, but they have been only partially successful. As an example, *Whaley and Lam* [7] proposed a cloning based context sensitive alias analysis using BDD. In another method *Lattner et al.* [8] proposed a cloning based context-sensitive alias analysis (without BDD) which claims significant performance benefits, but give precision only, equal to that of context-insensitive analysis. Considering above aspects for the precision we have chosen Andersen analysis which is highly precise but less scalable. To implement a best alias analysis in LLVM we need both precision and scalability hence, to use Andersen we should have to make it scalable.

In this thesis, we have improved the Andersen alias analysis [5] for that we studied the code base available in LLVM 2.6. The LLVM 2.6 Andersen analysis does not content any of the refinements (such as offline variable substitution or online cycle elimination) to improve its efficiency, so it can be quite slow in common cases, which makes algorithm to take $O(N^3)$ time. Considering these flaws in Andersen analysis, one of the Texas Ph.D student named *Jia Chen* has used *Online Cycle Elimination* and *Offline Variable substitution* functionality in Andersen [5], but still his implementation lack's several things such as External library functions, Exception handling, Mod/Ref, Extract and Insert value, and test cases required for the algorithm also, his implementation is not compatible with latest versions of LLVM. Considering above analysis we implemented Andersen analysis in the latest version of LLVM including the *Online Cycle Elimination* and *Offline Variable substitution* for efficiency. Further, we have included the functionality which was not there in the deprecated version of Andersen.

1.4 Our Contribution

The major contribution part of this thesis is implementing and improving Andersen alias analysis [5] in the latest version of LLVM. In this, we have improved the functionality such as External library functions, Exception handling, Mod/Ref, Extract and Insert value. Also, we have made Andersen implementation compatible with most of the latest version such as LLVM 3.6 and LLVM 3.7. Further, we have added extra test cases in our implementation to test the algorithm.

1.5 Organization of this Thesis

This thesis is arranged in following way. We describe the necessary definitions and terminologies to understand the thesis in Chapter 2. Mainly, we explain various analysis dimensions like flow-sensitivity, context-sensitivity and field-sensitivity with examples. We also mention the alias analysis as a graph problem which gives clear understanding of constraint graph.

In Chapter 3 we survey various alias analysis methods. We divide the survey in *five* categories which explain about the idea and algorithms of alias analysis.

In Chapter 4 we explain about the Andersen alias analysis implementation in LLVM. In this, we covered each phase which we have followed in the code base. Also, we mention the flow of implementation which gives the complete idea of code in LLVM.

In Chapter 5 we explain about the latest added modules and changes required for various versions of LLVM. Further, we added experimental results for our Andersen alias analysis with basicaa and cfl-aa which compares Andersen implementation.

We conclude our work in Chapter 6 and also explain some future directions in which this work can be extended.

Chapter 2

Background

In this chapter we cover all key concepts that are necessary to understand the forthcoming chapters. Here, we first define important terms and give various dimensions of alias analysis that affects the precision of alias analysis algorithm. In section 2.4, we present alias analysis as a graph problem which will give the idea of the structure of points-to information.

2.1 Definition and Important terms

A Pointer in a programming language like C/C++ is declared as *T *ptr* where *ptr* is the name of the pointer variable and *T* is the type of the variable it points-to. Thus, a pointer declared as *int *ptr* can point-to a variable of type integer.

Definitions

Pointer: A pointer is program variable whose *r-value* is either *zero* (*null*) or the *address* of another variable.

Pointee: A Pointee is program variable or a location whose address is the *r-value* of a pointer.

Aliases: Two pointer are aliases if they point to the same pointee.

Aliasing: Relationship between two aliases is called aliasing relationship or *aliasing.* The *aliasing* is defined as two expressions that evaluate to the same memory location.

Aliasing occur due to pointers, array indexing and union variables. *aliasing* is a reflexive relation, i.e., *p* aliases with *p.* Also, *aliasing* is a symmetric relation, i.e., *p* aliases with *q* implies *q* aliases with *p.*

Alias method responses: Must alias, No alias, May alias and Partial alias are the responses return by *alias* method in LLVM.

The *No alias* may be return when there is never an immediate dependence between any memory reference on pointer and memory reference.

The *Must alias* may return if the two memory objects are guaranteed to always start at exactly the same location. This implies that the pointers compare equal.

The *May alias* response is return whenever the two pointers might refer to the same object.

At last the *Partial alias* response when two memory objects are known to be overlapping in some way, but not start at the same address.

Must alias: If it is guaranteed that two pointers p and q always (i.e., in all program execution) alias with each other at a program point, then they are must alias of each other.

May alias: If two pointers p and q alias with each other at some program point in some program execution, then they are said to be may alias. If we can identify must alias it will help in reducing the number of variables tracked during the analysis and hence helps in an efficient alias analysis.

Alias analysis, Pointer analysis: Alias analysis or Pointer analysis is the mechanism of statically finding the aliasing relationship between pointer in a program.

Alias analysis may be exhaustive and compute the aliasing relationship for all possible pointer pairs. Also, alias analysis is demand driven which computes the aliasing relation only for given set of pointers pair. But we have to note here is an exhaustive pointer analysis could be very costly in terms of memory requirement. Hence, alternative representation for storing alias pairs is used.

Points-to pairs, Points-to fact: Points-to pair is a pointer-pointee relationship between two program variables or between a program variable and memory location which need not be distinct.

In this thesis we denote points-to pair using right arrow eg. $ptr \rightarrow \{var\}$ i.e., pointer *ptr* points-to pointee *var*. Also, in some cases like flow sensitive analysis a points-to fact is defined at a program point.

Example 2.1: Consider a program with following statements,
$p1 = \&v;$
$p2 = p1;$
$\therefore$ points-to facts derived from this program statements are,
$p1 \rightarrow v$
$p2 \rightarrow v$
Here we computed aliasing relation by checking a common pointee in points-to set of two pointers.

Points-to Analysis: The mechanism of statically computing the points-to set of pointers in a program.

Typically there are three aspects of performance measuring for points-to analysis, analysis time, memory requirement and analysis precision. *Analysis time* is measured in terms of analysis time complexity (eg., $O(n^3)$). *Memory Requirement* of an analysis is the total amount of memory in bytes used to store the points-to constraints and points-to information. *Precision* of a pointer analysis is a metric to compute the amount of conservativeness in the analysis.

Soundness: A static points-to analysis is sound if every realizable points-to fact (i.e., points-to fact occur at least once in execution of program) is computed by the analysis.

Clients: A client can be a program analysis, a program transformation or an application that quires the points-to information to the alias analyzer.

eg. Clients can be various dataflow optimizations like, common subexpression elimination, copy propagation and analysis like, live variable analysis, Mod/Ref analysis [4], program slicing.

In short a client to points-to analysis X_1 could be another points-to analysis X_2 which could make use of the information from X_1 to improve precision.

Scalar, Non-Scalar variable: A Scalar variable is a variable of basic type. *eg. int, int*, char*

A Non Scalar variable is a variable of an aggregate type. *eg. array and structure*

Intra-procedural analysis: The Intra-procedural analysis processes each function in isolation making conservative assumptions about the external environment (i.e., callers and callees)

Example 2.2: Consider following program statement,

```
main()
{
s1: ptr2=&z;
s2: fun(&x);
s3: fun(&y);
}
fun(int *ptr1)
{}
```

In the example 2.2 the intra-procedural analysis processes function *main* and *fun* in isolation, i.e. it computes the points-to solution for the statement in each function by approximating the effect of external function. Hence, for each pointer that receives points-to information from outside a function through function argument it assumes that the points-to information contains all the address taken variables in the program.

∴ Points-to information computed by intra-procedural analysis is,

$ptr1\colon ptr1 \rightarrow \{x, y, z\}$

As observe from the example the intra-procedural analysis is imprecise. However, all pointers that are accessed outside function have the same points-to information hence such pointers can be merged to reduce total number of variable tracked. Therefore intra-procedural analysis scales on the basis of program size.

Inter-procedural analysis: The inter-procedural analysis is more precise and it includes flow of information from one function to another. Thus, function processed collectively (not in isolation) but it keeps the information of all callers and callees. Caller and Callee represented using *Callgraph.* Each node in *Callgraph* is a function and a call from one function to another is denoted by directed edge. A cycle in a *Callgraph* denotes the recursion.

∴ Thus Points-to information computed by inter-procedural analysis for example 2.2 is,

$ptr1\colon ptr1 \rightarrow \{x, y\}$

As analysis merged the information coming from two calls to *fun* and excluded the other address-taken variable for *ptr1*.

2.2 LLVM

The LLVM[2] is a modern open source compiler infrastructure implemented in *C++* and designed as a set of reusable library with well defined interfaces. LLVM is a acronym for Low Level Virtual Machine. It is designed for link time, run time and compile time optimization of programs written in programming language. It has many supported front ends or languages such as *C, C++, objective-C, Ruby, Python* and many more. Also, it supports many CPU architecture in back-end such as ARM, Alpha, Intel x86, MIPS, PowerPC and SPARC. The LLVM is the framework which is separated in compilation process in frontend or analysis and transformation or backend.

2.3 Analysis Dimensions

The accuracy of a program analysis depends upon how it models an input program's data and control-flow. For modeling, we have some key concepts such as flow-sensitivity, field-sensitivity, and context-sensitivity this is known as analysis dimensions [9] 2.3. Although these dimensions are also applicable to other program analyses and transformations, we explain these in the context of pointer analysis.

2.3.1 Context-sensitivity

A context-sensitive analysis is an inter-procedural analysis. It distinguishes between different calling context in which a function can be called.

A context-insensitive analysis on the other hand does not differentiate between calling contexts of the analyzed function. Context-insensitive analysis allows values to flow from one call through function to return to another caller.

Example 2.3: Consider the program given below,

```
main()
{
s1: ptr2=&z;
s2: fun(&x);
s3: fun(&y);
}
fun(int *ptr1)
{}
```

A context sensitive analysis distinguish between the two calls to function *fun* from function *main* and computes a different points-to information for pointer *ptr1* across each context,

$ptr1 \rightarrow x$ along $s2$,

$ptr1 \rightarrow y$ along $s3$.

2.3.2 Flow-sensitivity

A flow sensitive analysis considers the control flow in the program while computing solution i.e., it differentiates between different program points. Flow sensitive analysis computes points-to set for pointers at each program points hence it is more precise than flow insensitive analysis.

On the other hand flow insensitive analysis ignores the control flow in program and assumes that any basic block can be reached from any other basic block. Flow insensitive analysis computes single points-to set for the entire program. It process all the program statements repeatedly in a sequential order this makes analysis more efficient than flow sensitive analysis.

Example 2.4: Consider the following program, statement *s1* and *s2* denotes the program points.
$s1$: a=&x;
$s2$: a=&y;
Points-to information for the example 2.4:
using flow-sensitive analysis:
at $s1$ $a \rightarrow x$,
at $s2$ $a \rightarrow y$
using flow-insensitive analysis:
$a \rightarrow \{x,\ y\}$

2.3.3 Field-sensitivity

Field sensitive analysis give how to modeled the aggregates. In aggregate we consider *struct, union, and array.* Field sensitivity keeps track of individual members of an aggregate data type i.e., it treats each field of an each aggregate separately [10].

Field insensitive analysis considers all members of the aggregate to be one.

Field based analysis modeled each field of an aggregate type separately and never distinguish between different field instances for different aggregate variable of same type [10].

Example 2.5: Consider following program, where p and q are the aggregate of same type T
$p.f1 =$ &x;
$q.f1 =$ &y;
$p.f2 =$ &z;
field-sensitive analysis computes: $p.f1 \rightarrow x$, $q.f1 \rightarrow y$, $p.f2 \rightarrow z$
field-insensitive analysis computes: $p \rightarrow \{x, z\}$, $q \rightarrow y$
field-based analysis computes: $T.f1 \rightarrow \{x, y\}$, $T.f2 \rightarrow z$

2.4 Alias analysis as a graph problem

The inclusion-based analysis proposed by Andersen [5] iteratively computes the points-to solution by using points-to constraints. However, iteratively going through all the points-to constraints is not the most efficient way to compute the fixed points-to set of an inclusion based analysis. The inclusion based analysis can be solved by using constraint graph (proposed by Fahndrich et. al. [11]). The constraint graph gives a graphical representation to the points-to analysis problem and enable its structural properties to optimize the analysis.

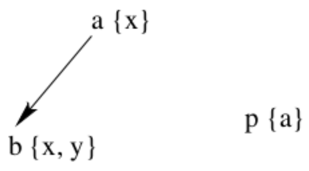

Figure 2.1: Constraint graph for example 2.6

Example 2.6: Consider the program statements,
$a = \&x;$
$b = \&y;$
$p = \&a;$
$b = *p;$

The Constraint graph G is a directed graph where each node represents a pointer and each directed edge $u \rightarrow v$ represents the subset (or inclusion) relationship. Each node also maintains its points-to set which gets updated as points-to information (flows along its incoming edges and propagated along the outgoing edges). A constraint graph is not static and new edges get added to it as complex constraints are evaluated, when no more edges can be added and no more points-to information can be propagated a fixed point of points-to information is computed at the node in graph.

The constraint graph given in the Figure shows the graph example. The graph contains three nodes corresponding to each pointer *a, b, p* each node is associated with its points-to information as shown in graph from *Example 2.6* by curly-braces. A directed edge from *a to b* indicates the flow of points-to information from *a to b* edge is added due to processing of the load constraint $b = *p$.

A points-to analysis using constraint graph is shown in algorithm 1

Algorithm 1 Points-to analysis using constraint graph

```
1. Initialize set $C$ of points-to constraints
2. Process $address - of$ constraints
3. add edge to constraint graph $G$ using *copy* constraints
4. *repeat*
5. Propagate points-to information in $G$
6. Add edges to $G$ using *load* and *store* constraints
7. do *until* fixed point reach
```

Here we have to note that, a naive analysis using a constraint graph does not give any additional benefits over the iterative Andersen analysis. To make analysis efficient using constraint graph the structure of constraint graph need to be changed. This is done using two methods, first by dynamically collapsing cycles in a graph and second propagating points-to information in topological

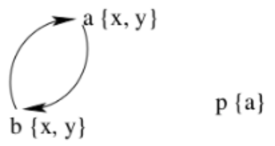

Figure 2.2: Constraint graph for example 2.7

order.

Example 2.7: Consider the following set of points-to constraint,
$a = \&x;$
$b = \&y;$
$p = \&a;$
$b = *p;$
$a = b;$
Final points-to information using constraint graph shown in Figure 2.2,
$a \rightarrow \{x, y\}$,
$b \rightarrow \{x, y\}$,
$p \rightarrow \{a\}$

2.5 Chapter Summary

In this chapter we have studied the background needed to understand the rest of the thesis. We defined various terms necessary to understand the alias analysis. Next, we have discussed analysis dimensions who affect the precision. Finally, we presented alias analysis as a graph constraint problem.

Chapter 3

A Survey of Alias Analysis Methods

In this chapter we present the survey of alias analysis methods. In this, we will try to cover most interesting and important work on alias analysis. Typically, a survey contains a broader perspective than individual algorithms and although it may not add a new innovation to the area, it helps in understanding the higher level picture of the area.

We divide the survey in *Five* categories. Also, we try to cover related work into this category instead of separate section.

1. Existing Surveys of Alias Analysis
2. Two key Points-to analysis methods
3. Algorithms which uses inclusion-based approach
4. Applications of alias analysis
5. Points-to analysis for other languages

3.1 Existing Surveys of Alias Analysis

In this section we discuss the various algorithms who give an idea about higher level picture of the area in the form of trends, pitfalls and lessons.

To understand the overview of alias analysis we have tabulated all the possible alias analysis algorithms on the basis on analysis and type, as shown in Table 3.1

Hind and Pioli's [4] survey on alias analysis is the most cited survey. In this, they have mentioned several dimensions which affect the trade-off between precision and scalability. The survey discusses various dimensions like flow-sensitivity, context-sensitivity, heap modeling (it tells how allocation sites are modeled). Also, it gives various issues like metrics of precision and reproducibility of results. Finally, it gives several directions for future research.

Rayside [12] given a summary of various important aspects of pointer analysis. He classifies several analysis on the basis of context-sensitivity and flow-sensitivity. In future direction he mention demand-driven analysis, incremental analysis and handling incomplete program.

Wu [13] presents a survey of pointer analysis. In this, he discussed various kinds of flow graphs used in literature, namely, context-sensitive call graph, inter-procedural control flow graph, procedural call graph. He also mentions abstract data representation used to model variables inaccessible to a procedure. Further, he mentions various approaches used to model recursive data structure such as 1-level, K-limiting, beyond K-limiting using symbolic access paths.

Ryder [14] given various approaches to reference analysis for object oriented programming based on analysis dimensions. The analysis dimensions are, field-sensitivity, flow-sensitivity, program representation, context-sensitivity and object representation. Further, author discussed various open issues in the context of object oriented languages, reflections, methods, exceptions, dynamic class loading and incomplete programs.

Raman [15] presents three methods, Unification based analysis, Pointer analysis using *BDD* and application of pointer analysis in bug detection.

3.2 Two Key Points-to Analysis Methods

In this section we present two important pointer analysis algorithms which compute approximate points-to information. One of them is Andersen's analysis, is based on set inclusion. It gives a relatively higher precision, but has a higher running time. The other algorithm is Steensgaard, is based on unification. It runs in almost linear time, but has relatively lower precision. Several points-to analysis algorithms available are variants of one of these two algorithms. Therefore, these algorithms are important. Both the algorithms are flow-insensitive.

3.2.1 Andersen Alias Analysis

The Andersen analysis [5]is a subset-based, flow-insensitive, context-insensitive, field-insensitive alias analysis, and implemented for inter-procedural alias analysis hence, widely believed it as a fairly precise.

Andersen [5] computes a single points-to set for each pointer, for the entire program. Let *pts(p)* denote the points-to set of a pointer p. Then, the goal of analysis is to compute $\forall p{:}pts(p)$. It processes each statement in the program to update the points-to sets of one or more pointers or objects as shown in Table 3.2. Since the analysis is flow-insensitive, there is no specific order in which the statements need to be processed. All statements are processed until a fixed point, i.e., the analysis stop when none of the points-to sets change. The order in which statements are processed does not change the points-to sets computed. There are four kinds of statements: address-of, copy, load and store. Load and Store statements are collectively called complex statements.

Table 3.2: Updating points-to sets (adapted from Rupesh Nasre [16])

Type of Statement	Statement	Update	Explanation
Address-of	x=&a	$\{a\} \subseteq pts\,(x)$	pts(x),is updated to include,a
Copy	x=y	$pts(y) \subseteq pts\,(x)$	pts(x),is updated to include,pts(y)
Load	x=*y	$\forall a \in pts\,(y)\,, pts(a) \subseteq pts\,(x)$	For each,a,that,y,may point to,,pts(x),is updated to include,pts(a)
Store	*x=y	$\forall b \in pts\,(x)\,, pts(y) \subseteq pts\,(b)$	For each,b,that,x,may point to,,pts(b),is updated to include,pts(y)

Andersen inclusion-based analysis works by converting each of the four statements into a set-constraint as shown in Table 3.2. Thus, a statement of the form $LHS = RHS$ is handled by adding set-constraint $RHS \subseteq LHS$ i.e., the points-to set of RHS is subset of LHS. Obtaining such a set of n constrains, its solution can then be obtained using a constraint solver in $O(n^3)$ time [10]. The constraint solver iteratively processes the set of constraints and updates the points-to set until a fixed-point.

Example 3.1: Consider following program statement,

$a = \&x;$

$b = \&y;$

$p = \&a;$

$b = *p;$

The points-to information computed for the above example using Andersen's analysis is,

$a \rightarrow \{x\},$

$b \rightarrow \{x, y\},$

$b \rightarrow \{a\}$

Table 3.1: Survey of Alias Analysis

	Flow-sensitive	**Flow-insensitive**	**Equality-based**	**Subset-based**	**CFL-based**		
Context-sensitive	-Landi and Ryder (1992), -Emami et. al. (1993), -Wilson and Lam (1995), -Whaley and Rinard (1999), -Radu and Rinard (2003), -Khedker et. al. (2012)	-Donglin and Mary (1999), -Cheng et. al. (2000)	-Fahndrich et. al. (2000)	-Jakab Fahndrich (2001), -Whaley and Lam (2004)	-Xu, Atanas, Manu (2009)	-Hendren (1999)	-global mod/ref,-no-AA
Context-insensitive	- Choi et. al. (1993)	-Burke and Hind (1994), -Marc and Susan (1997), -Manuvir Das (2000), -Pearce, Kelly et. al.(2007), -David, Liz et. al. (2008)	-Weihl (1980), -Steensgaard (1996)	-Andersen (1994), -Fahndrich (1998), -Heintze and Tardicu (2001), -Berndl et. al. (2003)	-CFL-AA (2013)		
Type-based		-Diwan et. al. (1998) TBAA					
		-Keith and Ken (1989), -Hasti and Susan (1998)					

3.2.2 Steensgaard Alias Analysis

Steensgaard's unification-based analysis [6] employs the bidirectional similarity to merge the points-to sets of LHS and RHS for a statement$LHS = RHS$. In unification based pointer analysis every pointer has a single points-to edge. Thus, if a pointer points-to multiple pointees, all those pointee are merged into the same set. In Andersen analysis multiple points-to edges per pointer are maintain and, therefore, In Andersen analysis [5] it is possible that two pointers may have a comman pointee, but different points-to sets, which is not true in Steensgaard's analysis [6].

An alternative way of looking at unification is from the perspective of the aliasing relation. Aliasing is a reflexive and symmetric relation but, it is not transitive [16], i.e., if pointers p and q are aliases and pointers q and r are aliases, then p and r need not always alias with each other.

For instance, if $p \rightarrow \{a\}, q \longrightarrow \{a, b\}, r \rightarrow \{b\}$ then p aliases with q and q aliases with r, but p does not alias with r. This property is maintained by Andersen's analysis (using the multiple points-to edges per pointer). However, due to a single points-to edge, Steensgaard's analysis [6] conservatively adds transitivity to the aliases represented. Since the relation represented is reflexive, symmetric and transitive, it becomes an equivalence relation forming partitions of alias-sets. Thus, each pointer in an alias-set aliases with all the pointers in the set and does not alias with any pointer outside the alias-set. Due to this key property, alias-sets can be very efficiently implemented using a union-find data structure [17]. In fact, Steensgaard's analysis runs in time $O(m\alpha(m))$ where, m is the number of constraints and $\alpha(m)$ is the inverse Ackermann's function [17] , which is a small constant. Thus, Steensgaard's analysis scales almost linearly with the program size. But due to transitive aliasing, it has significantly less precision compared to Andersen's analysis.

Example 3.2: Consider following program statement,

$a = \&x;$

$b = \&y;$

$p = \&a;$

$b = *p;$

The points-to solution obtained using unification is,

$a \rightarrow \{x, y\},$

$b \rightarrow \{x, y\},$

$p \rightarrow \{a\}$

3.3 Algorithms which uses inclusion-based approach

This section discussed the work which uses the inclusion-based analysis.

Andersen [5] introduced the inclusion based points-to analysis. It is context-insensitive and flow-insensitive points-to analysis that approximates the points-to information based on inclusion of points-to sets. Andersen analysis adds the set inclusion constraints. In set inclusion, A pointer assignment $p = q$ it adds the points-to information of q into that of p. Achieving $p \supseteq q$ which is set inclusion constraint. Finally, it solves those constraints to achieve a fixed point, which is sound approximates to the points-to information.

Hind et al. [18] propose an approximation algorithm for inter-procedural alias analysis. They also propose a method that constructs a program call graph during alias analysis for function pointer analysis. Most important finding is that, flow-insensitive analysis with *kill* information does not improve precision over without *kill* information.

Cheng and Mei [19] propose a modular inter-procedural pointer analysis based on access paths for pointers.

Yong et al. [20] given a points-to analysis to handle type casting and structure. They observe that supporting field-sensitivity can improve the analysis precision.

Pearce et al. [10] propose a field-sensitive points-to analysis for modeling aggregate and function pointers. They observe that field-sensitive analysis is more expensive to compute, but more precise over a field-insensitive analysis.

Lattner et al. [8] Propose a heap cloning based context sensitive points-to analysis to achieve a scalable implementation they are using a unification based analysis and flow-insensitive and left the context-sensitive analysis within strongly connected component.

Fahndrich et al. [11] given a context sensitive flow analysis using instantiation constraint. They show that flow information can be computed efficiently considering only well defined call-return sequence path even for higher order program.

Whaley and Lam [7] use *Heintze and Tardieus* [21]points-to analysis and *Cheng and mei's* [19] access path to develop an efficient reference analysis for JAVA. They shown the effectiveness of their field sensitive and intra-procedural method by computing static call graph for very large JAVA program.

Hardekopf and Lin [22] propose a semi sparse flow sensitive analysis for strong update handling. They convert top-level variables (or non address taken) to static single assignment (SSA) to improve analysis.

3.4 Applications of Alias Analysis

Alias analysis is not an optimization itself, it required a client to use the computed points-to information for performing optimizations over the program, In this section we will see various clients which make use of alias analysis.

1. The extended form of SSA called IPSSA given by *Livshits and Lam* [23] tracks the pointer and use it for format string violation and finding buffer overrun in C program.

2. A call-graph construction given by Milanova et al. [24] uses pointer information.

3. *Wu et al.* [25] propose an element-wise points-to mapping for loop based dependence analysis. An element-wise points-to mapping summaries the relation between a pointer and heap object it points to for every instance of the pointer inside a loop. They mention that, element-wise points-to information can significantly improve the precision of loop-based dependence analysis.

4. To detect security vulnerability in C program Avots et al. [26] use a field-sensitive, context-sensitive points-to analysis.

5. The dynamic points-to information is used by Mock et al. [27] to improve the precision of static slicing in C.

6. The use of pointer information to develop a shape analysis for C programs is given by Ghiya and Hendren [28].
7. *Tonella et al.* [29] use flow-insensitive and context-insensitive points-to analysis for C++ to reaching definitions analysis and slicing.
8. The use client-driven pointer analysis for C programs to several error detection problems *Guyer and Lin* [30].
9. More Clients which uses pointer analysis are, Mod/Ref analysis, live variable analysis, reaching definitions analysis, conditional constant propagation and dead code elimination *Hind and Pioli* [4].

3.5 Alias Analysis for other languages

In this section we will cover various Alias analysis (or Points-to) algorithms for other languages such as Python [31], JavaScript [32], JAVA [33].

The commonly alias analysis is used for C and C++ but its also developed for JAVA. Since pointers in JAVA are known as references hence, the analysis is termed as reference analysis. some of the work which are done for JAVA is given below, *Whaley and Lam* [7] and Berndl et al. [34] who propose variants of points-to analysis algorithm using *BDD* for JAVA. Sridharan et al. [35] propose a demand-driven analysis for JAVA. Also, they build upon their previous work to propose a client driven context sensitive points-to analysis.*Sun et al.* [36] present a probabilistic points-to analysis for object oriented programs. Lhotak and Hendren [37]present a framework of *BDD* based context sensitive points-to analysis.

Tonella et al. [29] propose a context-insensitive and flow-insensitive points-to analysis for *C++*, Which handles various object oriented features.

The JavaScript [32] has dynamic features such as runtime modification of object. *Jang and Choe* [38] propose the first points-to analysis for JavaScript. Their analysis can identify the use of structure field, to improve precision over a traditional Andersen analysis.

Gorbovitski et al. [39] propose alias analysis for a dynamic object oriented language for program optimization by incrementalization and specialization. They instantiate their work flow-sensitive and context-sensitive analysis for *Python* [31].

3.6 Chapter Summary

In this chapter we discussed a survey of various alias analysis methods. We classified them in seven categories and briefly discussed the work in each category.

Chapter 4

Implementation of Andersen Alias Analysis in LLVM

In this chapter we present Andersen alias analysis implementation details in LLVM. The section 4.1 gives the idea of Andersen analysis like what and why Andersen analysis is important. In next section we have explained flow of Andersen code base. Finally, we have given summary of chapter.

4.1 What and Why of Andersen Alias Analysis

What is Andersen Alias Analysis

Andersen's algorithm is a famous pointer-analysis algorithm proposed in L.O. Andersen's 1994 Ph.D thesis. The core idea of his algorithm is to translate the input program with statements of the form $p = q$ to constraints of the form "*q's* points-to set is a subset of *p's* points-to set", hence it is sometimes also referred to as *inclusion based* algorithm. The analysis is *flow insensitive* and *context insensitive*, meaning that it just completely ignores the control-flows in the input program and considers that all statements can be executed in arbitrary order.

Why Andersen Alias Analysis is important

In this implementation, we are reusing the existing code base from LLVM 2.6 by staying in the same inclusion based analysis framework. Because of its high-precision as a *flow insensitive* pointer analysis, Andersen's analysis has been deployed in some modern optimizing compilers.

Andersen's analysis is more precise than Steensgaard's *unification based* analysis. In most of the modern compilers pointer analysis is no longer *unification based*, but rather *inclusion based* and the overall pointer analysis framework remains *context insensitive*. These properties are important, and widely used by industrial-strength implementation available.

4.2 Alias Analysis in LLVM

The LLVM AliasAnalysis class [40] is the primary interface used by clients and implementations of alias analyses in the LLVM system. This class is the common interface between clients of alias analysis information and the implementations providing it, and is designed to support a wide range of implementations and clients (but currently all clients are assumed to be flow-insensitive). In addition to simple alias analysis information, this class exposes Mod/Ref (Modified/Referenced) information from those implementations which can provide it, allowing for powerful analyses and transformations to work well together.

4.2.1 Alias Analysis Class Overview in LLVM:

The AliasAnalysis class defines the interface that the various alias analysis implementations should support. This class exports two important enums: AliasResult and ModeRefResult which represent the result of an alias query or a mod/ref query, respectively.

Methods in AliasAnalysis Class:

The alias method

The alias method is the primary interface used to determine whether or not two memory objects alias each other. It takes two memory objects as input and returns MustAlias, PartialAlias, MayAlias, or NoAlias as appropriate.

The getModRefInfo methods

The getModRefInfo methods return information about whether the execution of an instruction can read or modify a memory location. Mod/Ref information is always conservative: if an instruction might read or write a location, ModRef is returned. The AliasAnalysis class also provides a getModRefInfo method for testing dependencies between function calls.

Other useful AliasAnalysis methods

- The pointsToConstantMemory method: This method returns true if pointer only points to unchanging memory locations (functions, constant global variables, null pointer).
- The doesNotAccessMemory method: This method check whether function reads or writes to memory. If it never read/write to memory then it returns true.
- The onlyReadsMemory method: This method returns true for a function if analysis can prove that the function only reads from non-volatile memory.

4.2.2 Existing alias analysis implementations and clients in LLVM:

Available AliasAnalysis

- The **-no-aa** pass: An alias analysis that never returns any useful information. this pass can be useful if you think that alias analysis is doing something wrong and are trying to narrow down a problem.

- The **-basicaa** pass: This pass is an aggressive local analysis that knows many important facts. such as different fields of a structure do not alias.
- The **-globalsmodref-aa** pass: This pass implements a simple context-sensitive mod/ref and alias analysis for global variables.
- The **-scev-aa** pass: This pass implements AliasAnalysis queries by translating them into ScalarEvolution queries.

Available Clients for AliasAnalysis

- The **-print-alias-sets** pass: This pass is use to print out the Alias Sets.
- The **-count-aa** pass: This pass is useful to see how many queries a particular pass is making and what responses are returned by the alias analysis.
- The **-aa-eval** pass: This pass simply iterates through all pairs of pointers in a function and ask an alias analysis whether or not the pointer alias. This gives an indication of the precision of the alias analysis.
- Memory Dependence Analysis: This is able to answer what preceding memory operations a given instruction depends on, either at an Intra or Inter-block level.

4.3 Flow of Andersen Alias Analysis code base in LLVM

This project is an implementation of Andersen's analysis in LLVM. The entire algorithm is broken down into three phases:

- **Phase 1** Translating from input LLVM IR into a set of constraints:
  - Here we treat *struct* in LLVM-IR *field insensitive.* This will yield worse result, but the analysis efficiency and correctness can be more easily guaranteed.
- **Phase 2** Rewriting the constraints into a smaller set of constraints whose solution should be the same as the original set:
  - In this two constraint optimization techniques called *HVN* and *HU* are used. The basic idea is to search for pointers that have equivalent points-to set and merge together their representation. Details can be found in Ben Hardekopf's [41].
- **Phase 3** Solving the optimized constraints:
  - Two constraints solving techniques called *HCD* and *LCD* are used. The basic idea is to search for *strongly connected components* in constraint graph. Details can be found in Ben Hardekopf's [42].

Implementation details:

This algorithm is implemented as four stages:

1. Object identification.
2. Inclusion constraint identification.
3. Offline constraint graph optimization.
4. Inclusion constraint solving.

Object identification: The object identification stage identifies all of the memory objects in the program, which includes *globals, heap allocated objects, and stack allocated objects.*

Inclusion constraint identification: The inclusion constraint identification stage finds all inclusion constraints in the program by scanning the program, looking for pointer assignments and other statements that affect the points-to graph. For a statement like "$A = B$" , this statement is processed to indicate that A can point to anything that B can point to. Constraint can handle *copies, loads, store and address taking.*

Offline constraint graph optimization: The offline constraint graph optimization portion includes "offline variable substitution algorithm" intended to compute pointer and location equivalences.

Pointer equivalences are those pointers that will have the same points-to sets, and *location equivalences* are those variables that always appear together in points-to sets. It also includes an *Offline cycle detection* algorithm that allows cycles to be collapsed sooner during solving.

Inclusion constraint solving: The inclusion constraint solving phase iteratively propagates the inclusion constraint until a fixed point is reached. This is an $O(N^3)$ algorithm.

Function constraints: Function constraints are handled as if they were *structs* with X fields. Thus, access to argument X of function Y is access to node index $getNode(Y) + X$. This representation allows handling of indirect calls without any issues. To which, an indirect call *Y(a,b)* is equivalent to $*(Y + 1) = a$, $*(Y + 2) = b$. The return node for a function is always located at $getNode(F) + CallReturnPos$. The arguments start at $getNode(F) + CallArgPos$.

4.4 Chapter Summary

In this chapter we have given detail information about Andersen alias analysis in LLVM. This includes Flow and implementation details of Andersen alias analysis in LLVM.

Chapter 5

Improvement in Andersen Alias Analysis in LLVM

This is the most important chapter in our thesis. In this, we have explained all major changes in the section 5.1 here, we have covered required changes with respective to the version of LLVM. Further, we have mentioned newly added functionality in latest version of Andersen analysis in section 5.2. Finally, we have concluded the chapter in summary section.

5.1 Changes for different LLVM versions

The basic idea taken from LLVM 2.6 code base, which is remain same, but we have to make it convenient for latest versions of LLVM. Hence, we have added some of the improved techniques who made it more useful and scalable.

LLVM 2.6:

As mentioned in LLVM 2.6 the Andersen alias analysis [5] was available as a pass, they mentioned that it is implemented for inter-procedural alias analysis. Also, it is a subset-based, flow-insensitive, context-insensitive, and field-insensitive alias analysis that is widely believed fairly precise. Unfortunately, this algorithm is also $O(N^3)$. The LLVM 2.6 implementation does not implement any of the refinements (such as *offline variable substitution* or *online cycle elimination*) to improve its efficiency, so it can be quite slow in common cases.

Considering the flaws in LLVM 2.6 Andersen analysis, one of the Texas Ph.D student named *Jia Chen* has added *Online Cycle Elimination and Offline Variable substitution* functionality to Andersen [5], but still his implementation lack's several things such as External library functions, Exception handling, Mod/Ref, Extract and Insert value, and test cases required for algorithm also, his implementation is not compatible with latest versions of LLVM. We have added mentioned functionality in the Andersen.

LLVM 3.4:

In LLVM 3.4 we have isolated and added important functions as a separate files which make it easier to understand and improve. In this following are the major phases which we have made in separate files.

1. **Constraint Collect**: In this, we scan the program, adding a constraint to the constraints list for each instruction in the program that induces a constraint, and setting up the initial points-to graph.
2. **Constraint Optimization:** In this two constraint optimization techniques called *HVN* and *HU* are used.
3. **Cycle Detector:** Any concrete class that does cycle detection should inherit from this class, specify the Graph Type, and implement all the abstract virtual functions. Also, it offers functionality of detecting *strongly connected components* (SCC) in a graph.
4. **Constraint Solving:** In this, we iteratively processes the constraints list propagating constraints (adding edges to the Nodes in the points-to graph) until a fixed point is reached. We use a variant of the technique called Lazy Cycle Detection which is described in the Hardekopf's paper [42]. The paper describes performing cycle detection one node at a time, which can be expensive if there are no cycles, but there are long chains of nodes that it heuristically believes are cycles (because it will *DFS* from each node without state from previous nodes). Instead, we use the heuristic to build a worklist of nodes to check, then cycle detect them all at the same time to do this more cheaply. This catches cycles slightly later than the original technique did, but does it make significantly cheaper.
5. **External Library:** We have included external library call identification in this implementation.

LLVM 3.6:

We have made many change to move to LLVM 3.6 among them two major changes due to change in LLVM version are given as follows:

1. We have to change some code from *ConstraintOptimize.cpp* as LLVM 3.6 has changed error message format in file *include/llvm-3.6/Support/ToolOutputFile.h*
2. Another major change in *AndersenTest.cpp* at line 134 to 145, which is due to change in LLVM-3.6 file *llvm-3.6/unittests/Analysis/CFGTest.cpp*

LLVM 3.7:

- In LLVM 3.7 the major changes are in DataLayout as in this version LLVM made DataLayout as mandatory part of LLVM framework. For more information please refer LLVM DataLayout [43].
- Also, in LLVM 3.7 we have to change location of *PassManger.h* as it is moved to *llvm-3.7/IR/PassManager.h*

Because of above changes in LLVM 3.7 we have to modify following files in our implementation:

1. AndersenAA.cpp
2. Andersen.cpp
3. AndersenTest.cpp
4. ConstraintCollect.cpp

5.2 Latest Included Modules in Andersen AA

Andersen [5] Implementation available for LLVM 2.6 is buggy and lacks many functionalities which make the code impractical to find out alias pairs efficiently. Hence, to make the implementation powerful and efficient we have added following major parts into the Andersen implementation.

Second important part of our implementation is, it is supported by all latest LLVM versions such as LLVM 3.4, LLVM 3.5, LLVM 3.6 and LLVM 3.7 Hence, it will be useful to all those who are working on alias analysis and want to study Andersen Alias analysis.

1. External Library functions
2. Mod/Ref method
3. Atomic instruction handling
4. Exception Handling (Landingpad)
5. Including functionality of Extract value and Insert value instruction
6. Added Extra Test cases

External Library Functions

What is External Library Functions:

In this, we have added more than 250 external library functions which are useful for instruction to look through and analyze the functions for possible pointers.

What we have added:

In this, we have added more than 250 external library functions which are useful for instruction to look through and analyze the functions for possible pointers.

Mod/Ref Method

What is Mod/Ref:

The Mod/Ref method returns information about an instruction execution whether it can read or modify a memory location. Mod/Ref information is always conservative if an action may read a location.

Mod/Ref also use for testing dependencies between function calls. Like in LLVM method *getModRefInfo* who takes two callsites *cs1* and *cs2* returns,

noModRef if neither call writes to memory read or written by other.

Ref if *cs1* reads memory written by *cs2*.

Mod if *cs1* writes to memory read or written by *cs2*.

ModRef if *cs1* might read or write memory written by *cs2*.

What we have added:

Added Mod/Ref code.

Atomic Instruction

What is Atomic Instruction:

In general atomic operations are useful to use multiple threads to correctly manipulate an object without using mutex lock. To be able to write correct concurrent programs atomic instruction are needed. Atomic instructions are designed to provide readable IR and optimize code generation. LLVM has various atomic instructions which are Load and Store instructions LLVM offers a set of different atomic orderings for these instructions.

LLVM atomic instructions:

load atomic obtains value from atomic object.

store atomic replaces value of an atomic object.

atomicrmw replaces value of an atomic object and obtain the previous value. This instruction is used to automatically modify memory.

atomicmpxchg replaces value of an atomic object and obtains the previous value if equal otherwise performs atomic load. This instruction loads a value in memory and compare it to a given value if they are equal it tries to store a new value into the memory.

What we have added:

Both *atomicrmw* and *atomicmpxchg* code added.

Exception Handling (Landingpad)

What is Exception Handling:

In Exception handling most commonly used technique is "zero cost exception handling". To support exception handling one must change the semantics of a function call and this call fork the execution flow. One branch is taken when everything is fine and second branch is taken in case of an exception. In LLVM IR there is invoke instruction to call functions that may throw. When second branch is taken several kinds of actions may be performed

- call destructor (cleanup)
- continue stack unwinding (resume)
- enforce throw specifications (filter)

- restore normal control flow (catch)

Hence, LandingPad instruction will use to perform these actions. In LandingPad all data like personality routine, catched types, throw specifications and cleanup actions are already specified so, no additional data should passed to the backend to generate exception related information in object file.

The LandingPad instruction is used by LLVM exception handling system to specify that basic block is a landingpad (one where the exception lands) and corresponding to the code found in the catch portion of a try/catch sequence. It defines value supplied by the personality function upon re-entry to the function. The result value has the type *resultty.*

What we have added:

We have added Landingpad code which is available in *cfl-aa* analysis implementation.

Extract value and Insert value instruction

What is Extract Value:

LLVM support several instructions for working with aggregate values. Aggregate types are subset of derived types that contain multiple member types. arrays and structures are aggregate types.

Extract Value:

The Extract Value instruction extracts the value of a member field from a aggregate value.

Syntax:

$< result >= extractvalue < aggregatetype >< val >, < idx > \{, < idx >\}*$

The result is the value of the position in the aggregate specified by the index operands.

eg:

$< result >= extractvalue\ \{i32, float\}\ \%agg,\ 0\ \ ; yields\, i32$

What we have added:

Added extract value code.

What is Insert Value:

Insert Value:

The Insert Value instruction inserts a value into the member field in an aggregate value.

Syntax:

$< result >= insertvalue < aggregatetype >< val >, < ty >< elt >, < idx > \{, < idx >\}*; \quad yields < aggregatetype >$

The result is an aggregate of the same type as val. Its value is that of val except that the value at the position specified by the indices's is that of "elt".

eg:

$agg1 = insertvalue\ \{i32, float\}\ undef,\ i32\ 1, 0; \quad yields\, \{i32\ 1, float\, undef\}$

What we have added:

Added Insert value code.

Added Extra Test cases

What is Extra Test Cases:

The extra Test cases are added to the Andersen implementation, which contains all test cases available from LLVM unittest test cases. The result for all these test cases are passed by our Andersen algorithm.

What we have added:

Added Extra test cases from LLVM test cases for analysis.

5.3 Experimental Result Analysis

We present results and analysis for two set of programs one of them contain 17 test programs which are the loops from the GCC [1] vectorizer example given by *Dorit Nuzman* [44] and another one contains 8 C/C++ programs.

The results shown in the Figure 1, 2 and 3 are the graph showing percent of May, No, and Must aliases return by the alias analysis algorithms i.e., andersen, basicaa, and cfl-aa. In alias analysis algorithm we used *alias* method which gives whether two memory object alias to each other or not and returns *May alias, No alias, Must alias or Partial alias.* For more details please see section 2.1.

We have given results for *Eight* common programs which contains external function call, exception handling, extract and insert value instruction. Mentioned examples are taken to test the added functionality in our Andersen algorithm. Like in example 1 we have external function call, named, *get_time* and *put_time* which is useful to test added external function in Andersen analysis same about the remaining examples each of them are taken to test specific functionality which is included in our implementation.

In Figure 5.1, we have shown the percentage of *No alias* response tracked by all three algorithms such as *basicaa, cfl-aa* and Andersen. In the graph all *three* algorithms are tested under all examples in this, the results shown by cfl-aa are the best among the reaming two algorithms as the response return by algorithm is not able to find out exact alias set. If we compare our algorithm with *basicaa* algorithm from LLVM which is assume to be most close analysis algorithm we can say our algorithm is as efficient as *basicaa.*

In Figure 5.2, we have given percentage of *May alias* response generated by all three algorithms such as *basicaa*, *cfl-aa* and *andersen.* As described on the LLVM alias analysis page [40], "A more precise alias analysis algorithm will have a lower number of may aliases". Our algorithm shows less number of aliases as compared to *cfl-aa* and it is nearer to *basicaa.* Which indicates that our algorithm is more precise than *cfl-aa* and equal to *basicaa.*

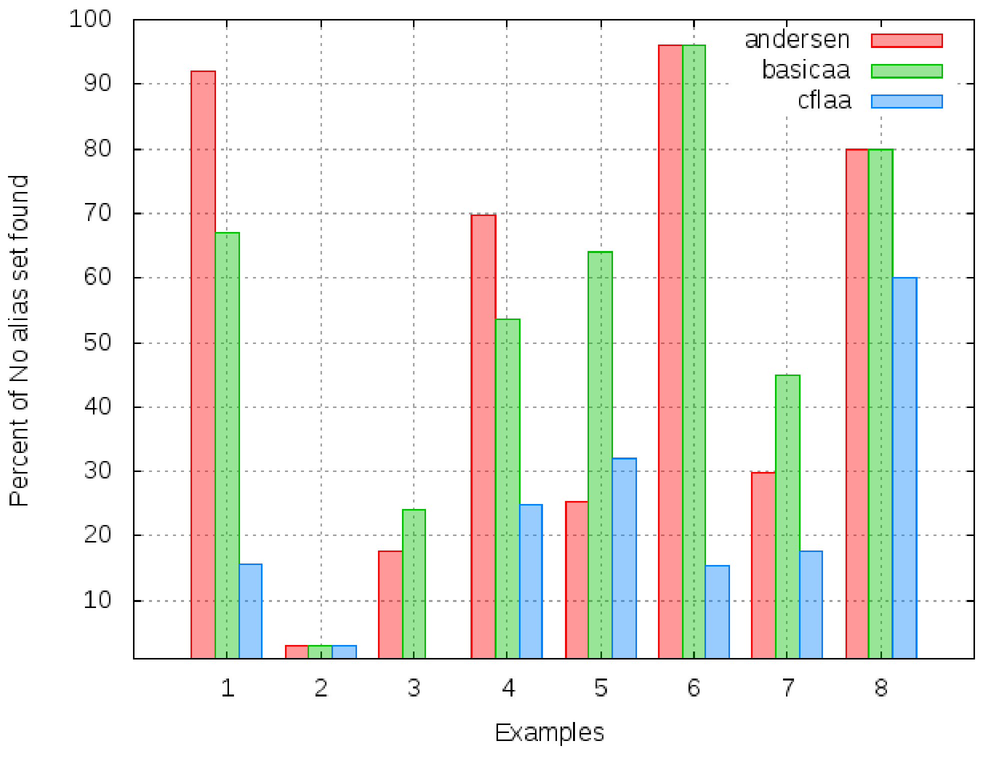

Figure 5.1: No Alias Analysis

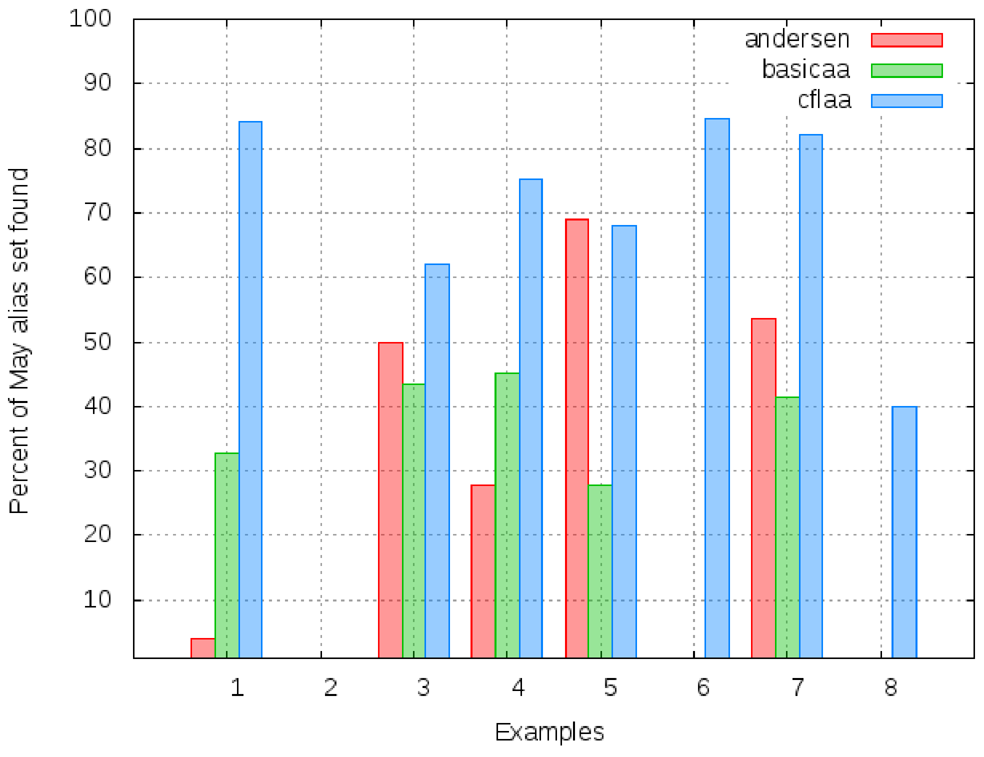

Figure 5.2: May Alias Analysis

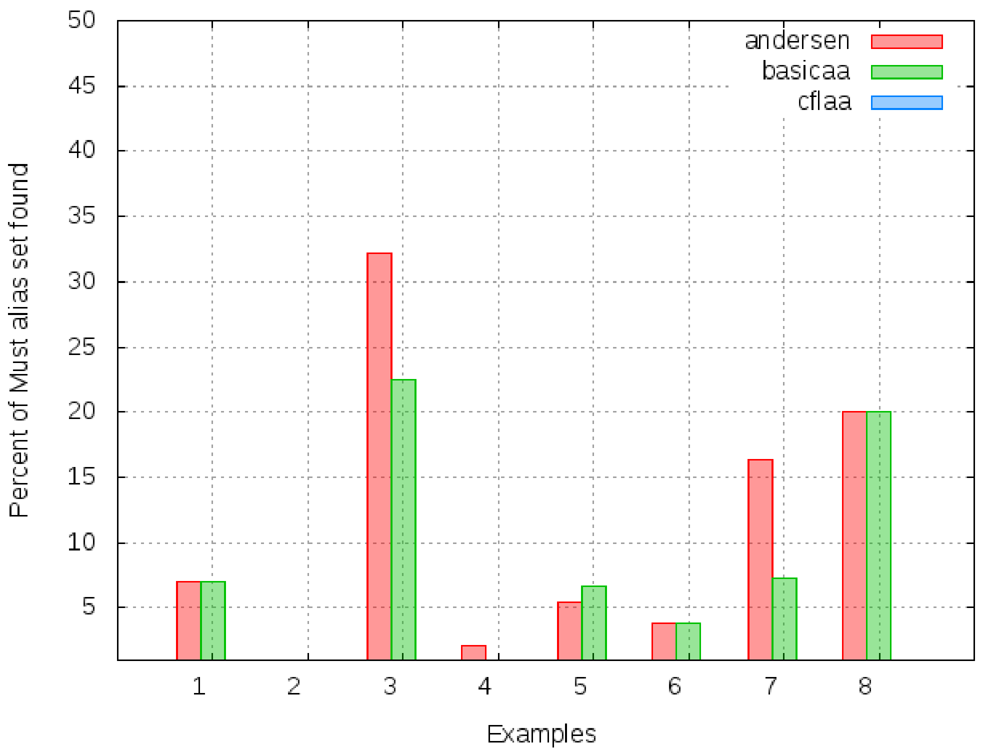

Figure 5.3: Must Alias Analysis

The Figure 5.3 graph shows the percentage alias response for *Must alias*. Here, our implementation shows the response same as *basicaa* algorithm. While *cfl-aa* gives very less Must alias result which indicate that our analysis should check aliases throughly so that our points-to set should be exact.

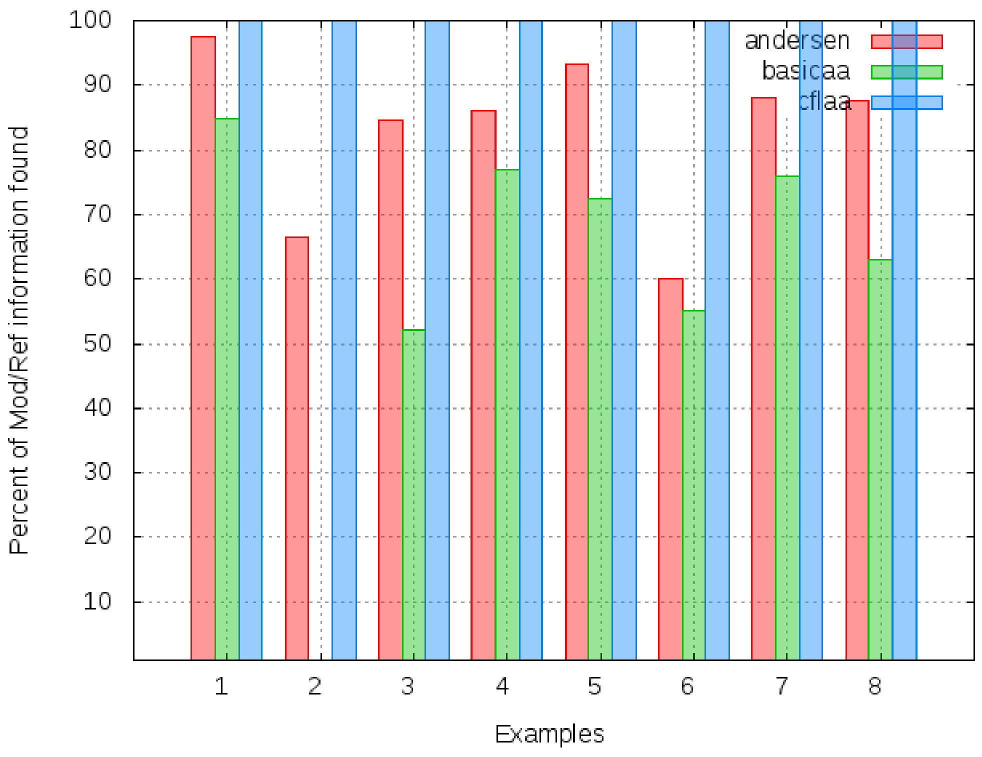

Figure 5.4: Mod/Ref Information

Figure 5.4 mentions the *Mod/Ref* information which we already explain in section 5.2. In this graph our implementation shows the average results compare to remaining two algorithms.

Improvement in vectorization due to alias analysis:

The vectorization example contains total 18 C/C++ test programs given by *Dorit Nuzman* [44]. The vectorizer test cases demonstrate the current vectorization capabilities. In this, we like to mention the basic block vectorization aka SLP, which is enabled by the flag -ftree-slp-vectorize and requires the same platform dependent flag as loop vectorization. The SLP is enabled by default at *-O3* optimization. SLP (Superword Level Parallelism) is a vectorization technique based on loop unrolling and basic block vectorization. The examples included in vectorization are, gfortran, unknown misalignment, multidimension arrays, data type of different sizes, summation reduction, induction, double reduction, basic block SLP, basic block SLP with multiple types, etc.

In Figure 5.5, we have produced the results for vectorization examples with and without enabling *-O3* optimization. Here, while enabling *-O3* option we assume that default alias analysis will call. To demonstrate the effect of alias analysis on the client like vectorization we can see the promising results in the graph.

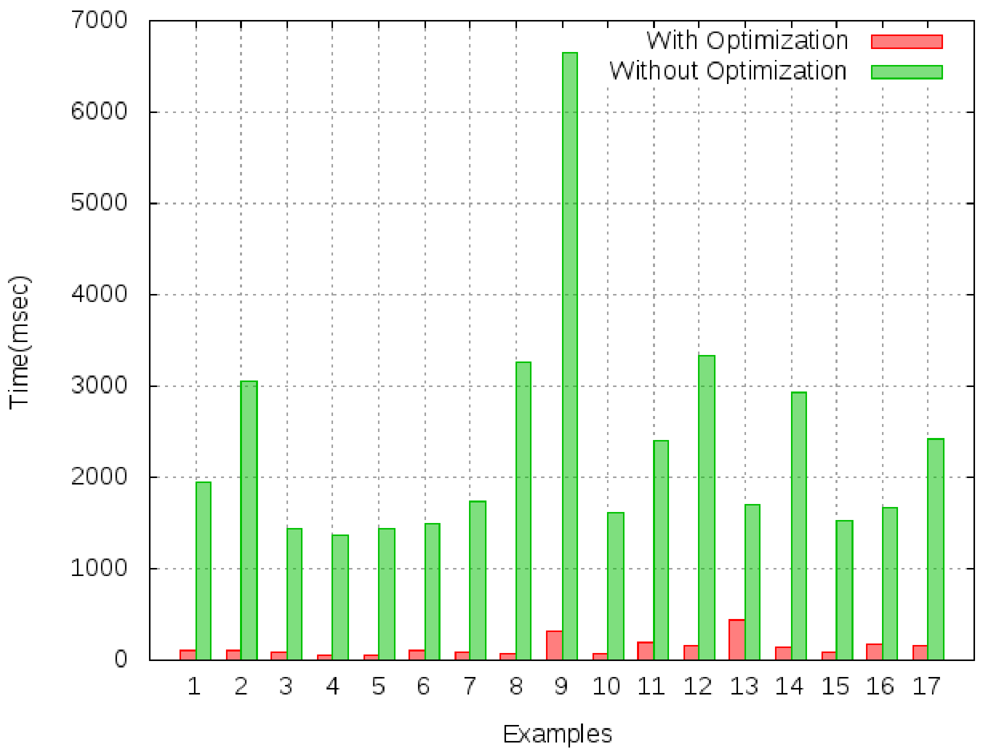

Figure 5.5: Improvement in vectorization due to Alias Analysis

Analysis of all three algorithms with vectorization examples:

As mention in the Figure 5.5, the vectorization examples show the optimization after applying alias analysis. In Figure 5.6, we have shown results for vectorization examples after applying each algorithm such as *Andersen, basicaa* and *cfl-aa*. In this, to produce the results we have enabled optimization level three (*-O3*) for with optimization. While for results using *Andersen*, *basicaa* and *cfl-aa* we have applied separate optimization option such as, for Andersen algorithm we have used *-fresh-andersen-aa* option with optimizer tool *opt-3.7*. Also, for cfl-aa we have used *-cfl-aa* option and for basicaa *-basicaa* option in LLVM.

The result shown for all three algorithms unable to give good results as compare to *with optimization* option, the reason behind this, is we have used *-O3* optimization which also contains other optimizations. Our results are comparable and depict that analysis is nearly equal to *basicaa* and in some cases *cfl-aa*.

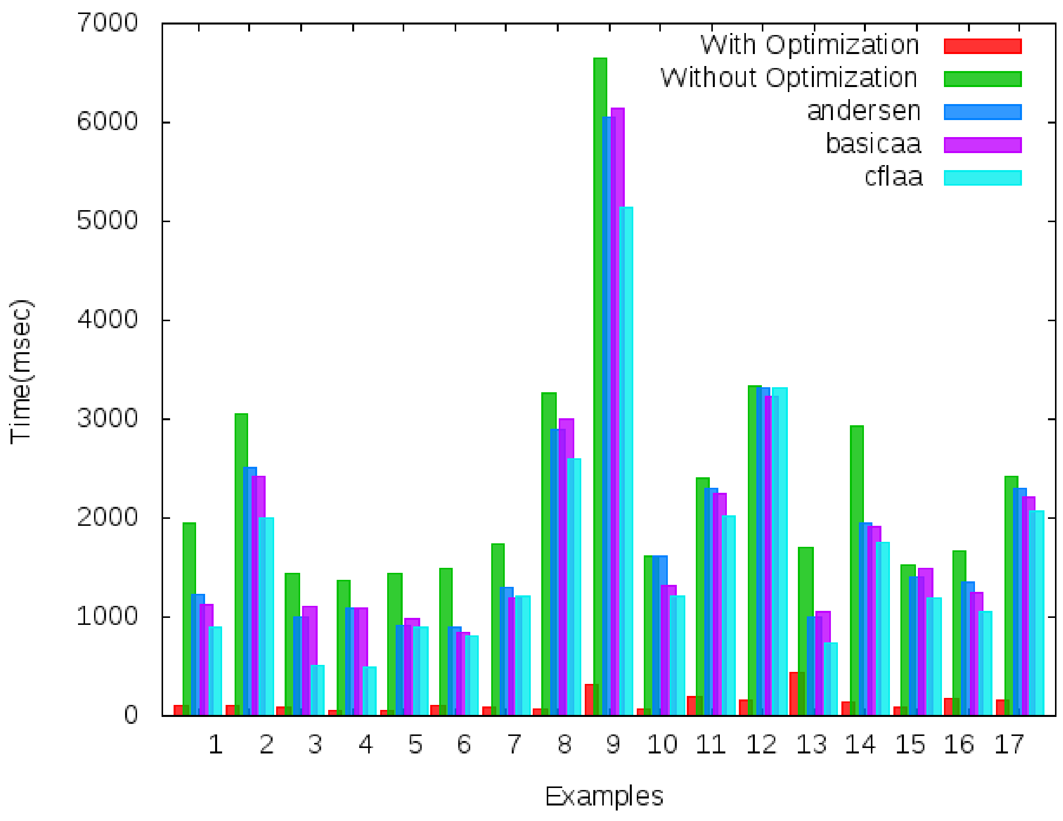

Figure 5.6: Time required for vectorization after applying fresh-andersen alias analysis

5.4 Chapter Summary

This chapter describes the actual improvement and changes in Andersen alias analysis for latest version of LLVM. Further, In this section we have given some result graph who compare fresh-andersen alias analysis with existing algorithms.

Chapter 6

Conclusion and Future Work

Conclusion

In this thesis, we proposed approach to improve the scalability and effectiveness of Andersen alias analysis. For that we have reused Andersen code base available in LLVM 2.6, to improve the scalability and efficiency we have used online cycle detection and offline variable substitution methods in Andersen analysis. Also, we have included some functionality such as, External functions, Exception handling, Mod/Ref, Extract and Insert value, with more test cases, which make it more efficient. In our experimental result analysis section we have given comparison between our improved algorithm with basicaa and cfl-aa algorithms available in LLVM up to some extents. Andersen alias analysis has been removed from LLVM since version 2.7 hence, our implementation will be useful to the researcher who is interested in the alias analysis and want to understand the Andersen alias analysis.

Future Work

In this section, we describe a few directions along which our work can be extended.

- Making Andersen alias analysis context sensitive: The Andersen alias analysis can be moved to context sensitivity which will improve the scalability of the analysis and we can find out more accurate points-to pairs.
- Making Andersen alias analysis scalable to large program: The current implementation of Andersen alias analysis is not scalable to millions of lines of code. The main issue is we are not able to summarize the side effects of procedure. If we are able to find out the way to do this then, we can scale our algorithm to larger programs like millions of lines of code. The context sensitive version of Andersen scalable to millions of lines of code can be deployable in modern optimizing compilers to provide advanced optimization.

References

[1] GCC. `http://gcc.gnu.org/`.

[2] C. Lattner. LLVM Compiler Infrastructure. `http://llvm.org/`.

[3] Static Program Analysis. `http://en.wikipedia.org/wiki/Staticprogramanalysis/`.

[4] M. Hind and A. Pioli. Which Pointer Analysis Should I Use? *SIGSOFT Softw. Eng. Notes* 25, (2000) 113–123.

[5] L. Andersen. Program analysis and specialization for the C programming language 1994. `http://repository.readscheme.org/ftp/papers/topps/D-203.pdf`.

[6] B. Steensgaard. Points-to Analysis in Almost Linear Time. In Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '96. ACM, New York, NY, USA, 1996 32–41.

[7] J. Whaley and M. 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 '04. ACM, New York, NY, USA, 2004 131–144.

[8] C. Lattner, A. Lenharth, and V. Adve. Making Context-sensitive Points-to Analysis with Heap Cloning Practical for the Real World. In Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '07. ACM, New York, NY, USA, 2007 278–289.

[9] B. G. Ryder. Dimensions of Precision in Reference Analysis of Object-oriented Programming Languages. In Proceedings of the 12th International Conference on Compiler Construction, CC'03. Springer-Verlag, Berlin, Heidelberg, 2003 126–137.

[10] D. J. Pearce, P. H. J. Kelly, and C. Hankin. Efficient Field-sensitive Pointer Analysis for C. In Proceedings of the 5th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE '04. ACM, New York, NY, USA, 2004 37–42.

[11] M. Fähndrich, J. S. Foster, Z. Su, and A. Aiken. Partial Online Cycle Elimination in Inclusion Constraint Graphs. In Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation, PLDI '98. ACM, New York, NY, USA, 1998 85–96.

[12] D. Rayside. Points-to Analysis 2005. `http://www.cs.utexas.edu/~pingali/CS395T/2012sp/lectures/points-to.pdf`.

[13] Q. Wu. Survey of Alias Analysis. `http://www.cs.princeton.edu/jqwu/Memory/survey.html/`.

[14] B. G. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. In Proceedings of the 12th International Conference on Compiler Construction, CC'03. Springer-Verlag, Berlin, Heidelberg, 2003 126–137.

[15] Pointer Analysis A Survey, CS203 UC Santa Cruz,2004. `http://www.soe.ucsc.edu/vishwa/publications/Pointers.pdf/`.

[16] R. Nasre, K. Rajan, R. Govindarajan, and U. P. Khedker. Scalable Context-Sensitive Points-to Analysis Using Multi-dimensional Bloom Filters. In Proceedings of the 7th Asian Symposium on Programming Languages and Systems, APLAS '09. Springer-Verlag, Berlin, Heidelberg, 2009 47–62.

[17] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Introduction to Algorithms, Third Edition. 3rd edition. The MIT Press, 2009.

[18] M. Hind and A. Pioli. Assessing the Effects of Flow-Sensitivity on Pointer Alias Analyses. In Proceedings of the 5th International Symposium on Static Analysis, SAS '98. Springer-Verlag, London, UK, UK, 1998 57–81.

[19] B.-C. Cheng and W.-M. W. Hwu. Modular Interprocedural Pointer Analysis Using Access Paths: Design, Implementation, and Evaluation. In Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation, PLDI '00. ACM, New York, NY, USA, 2000 57–69.

[20] S. H. Yong, S. Horwitz, and T. Reps. Pointer Analysis for Programs with Structures and Casting. In Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation, PLDI '99. ACM, New York, NY, USA, 1999 91–103.

[21] N. Heintze and O. Tardieu. Ultra-fast Aliasing Analysis Using CLA: A Million Lines of C Code in a Second. In Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation, PLDI '01. ACM, New York, NY, USA, 2001 254–263.

[22] B. Hardekopf and C. Lin. Flow-sensitive Pointer Analysis for Millions of Lines of Code. In Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO '11. IEEE Computer Society, Washington, DC, USA, 2011 289–298.

[23] V. B. Livshits and M. S. Lam. Tracking Pointers with Path and Context Sensitivity for Bug Detection in C Programs. In Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE-11. ACM, New York, NY, USA, 2003 317–326.

[24] A. Milanova, A. Rountev, and B. G. Ryder. Precise Call Graphs for C Programs with Function Pointers. *Automated Software Engg.* 11, (2004) 7–26.

[25] P. Wu, P. Feautrier, D. Padua, and Z. Sura. Instance-wise Points-to Analysis for Loop-based Dependence Testing. In Proceedings of the 16th International Conference on Supercomputing, ICS '02. ACM, New York, NY, USA, 2002 262–273.

[26] D. Avots, M. Dalton, V. B. Livshits, and M. S. Lam. Improving Software Security with a C Pointer Analysis. In Proceedings of the 27th International Conference on Software Engineering, ICSE '05. ACM, New York, NY, USA, 2005 332–341.

[27] M. Mock, D. C. Atkinson, C. Chambers, and S. J. Eggers. Improving Program Slicing with Dynamic Points-to Data. *SIGSOFT Softw. Eng. Notes* 27, (2002) 71–80.

[28] R. Ghiya and L. J. Hendren. Is It a Tree, a DAG, or a Cyclic Graph? A Shape Analysis for Heap-directed Pointers in C. In Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '96. ACM, New York, NY, USA, 1996 1–15.

[29] P. Tonella, G. Antoniol, R. Fiutem, and E. Merlo. Flow Insensitive C++ Pointers and Polymorphism Analysis and Its Application to Slicing. In Proceedings of the 19th International Conference on Software Engineering, ICSE '97. ACM, New York, NY, USA, 1997 433–443.

[30] S. Z. Guyer and C. Lin. Client-driven Pointer Analysis. In Proceedings of the 10th International Conference on Static Analysis, SAS'03. Springer-Verlag, Berlin, Heidelberg, 2003 214–236.

[31] Python Programming Language. `http://www.python.org/`.

[32] JavaScript Programming Language. `http://www.javascript.com/`.

[33] Java Programming Language. `http://www.java.com/`.

[34] M. Berndl, O. Lhoták, F. Qian, L. Hendren, and N. Umanee. Points-to Analysis Using BDDs. In Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, PLDI '03. ACM, New York, NY, USA, 2003 103–114.

[35] M. Sridharan, D. Gopan, L. Shan, and R. Bodík. Demand-driven Points-to Analysis for Java. In Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA '05. ACM, New York, NY, USA, 2005 59–76.

[36] Q. Sun, J. Zhao, and Y. Chen. Probabilistic Points-to Analysis for Java. In Proceedings of the 20th International Conference on Compiler Construction: Part of the Joint European Conferences on Theory and Practice of Software, CC'11/ETAPS'11. Springer-Verlag, Berlin, Heidelberg, 2011 62–81.

[37] O. Lhoták and L. Hendren. Evaluating the Benefits of Context-sensitive Points-to Analysis Using a BDD-based Implementation. *ACM Trans. Softw. Eng. Methodol.* 18, (2008) 3:1–3:53.

[38] D. Jang and K.-M. Choe. Points-to Analysis for JavaScript. In Proceedings of the 2009 ACM Symposium on Applied Computing, SAC '09. ACM, New York, NY, USA, 2009 1930–1937.

[39] M. Gorbovitski, Y. A. Liu, S. D. Stoller, T. Rothamel, and T. K. Tekle. Alias Analysis for Optimization of Dynamic Languages. In Proceedings of the 6th Symposium on Dynamic Languages, DLS '10. ACM, New York, NY, USA, 2010 27–42.

[40] LLVM Alias Analysis. `http://llvm.org/docs/AliasAnalysis.html/`.

[41] B. Hardekopf and C. Lin. Exploiting Pointer and Location Equivalence to Optimize Pointer Analysis. In Proceedings of the 14th International Conference on Static Analysis, SAS'07. Springer-Verlag, Berlin, Heidelberg, 2007 265–280.

[42] B. Hardekopf and C. Lin. The Ant and the Grasshopper: Fast and Accurate Pointer Analysis for Millions of Lines of Code. In Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '07. ACM, New York, NY, USA, 2007 290–299.

[43] LLVM DataLayout Changes in LLVM-3.7. `https://github.com/llvm-mirror/llvm/commit/c94da20917cb4dfa750903b366c920210c5265ee?diff=split/`.

[44] Loop Example from Vectorizer, by Dorit Nuzman. `http://gcc.gnu.org/projects/tree-ssa/vectorization.html/`.