Alias Analysis in LLVM

by

Sheng-Hsiu Lin

Presented to the Graduate and Research Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Computer Science

Lehigh University

May 2015

This thesis is accepted and approved in partial fulfillment of the requirements for the degree of Master of Science.

_____

Date

_____

Dissertation Advisor

_____

Chairperson of Department

Acknowledgements

Thanks to all fellow researchers in the SOS Lab at Lehigh University including Ben Niu, Shen Liu, Zhiyuan Wan and Dongrui Zeng for their contributions to this research. Special thanks to my advisor Gang Tan for his invaluable guidance.

Contents

List of Tables

List of Figures

Abstract

Alias analysis is a study of the relations between pointers. It has important applications in code optimization and security. This research introduces the fundamental concepts of alias analysis. It explains different approaches of alias analysis with examples. It provides a survey of some very important pointer analysis algorithms. LLVM interface is introduced along with the alias analyses that are currently available on it. This research implementes a Steensgaard's pointer analysis on LLVM. The philosophy of this implementation is explained in detail. Evaluations of rule based basic alias analysis, Andersen's pointer analysis, Steensgaard's pointer analysis and data structure analysis are provided with experimental results on their precision, time and memory usage.

Chapter 1

Introduction to Alias Analysis

The purpose of alias analysis is to determine all possible ways a program may access some given memory locations. A set of pointers are said to be in an alias group if they all point to the same memory locations. Alias analysis is sometimes referred to as pointer analysis. However, it is not to be confused with points-to analysis, which is a sub-problem of alias analysis. Points-to analysis computes sets of memory locations that each pointer may point to. Such result can be useful in some applications, and can be used to derive alias information. Alias analysis and points-to analysis are often implemented by performing static code analysis.

Alias analysis is very important in compiler theory. Some of its most notable applications include code optimization and security. Compiler level optimization needs pointer aliasing information to perform dead code elimination (removing code that does not affect the program's result), redundant load/store instruction elimination, instruction scheduling (rearranging instructions) and more. Program security enforcement at compiler level uses alias analysis to help detect memory leaks and memory related security holes.

Chapter 2

Alias Analysis Survey

There are many varieties of alias analysis. They are often categorized by properties such as *field-sensitivity*, *inter-procedural v.s. intra-procedural*, *context-sensitivity* and *flow-sensitivity*.

2.1 Field-Sensitivity

Field-sensitivity is the strategy that governs the way alias analysis models fields in built-in or user defined data structures. There are three approaches to field-sensitivity – *field-sensitive*, *field-insensitive* and *field-based*. Consider the following code:

```
1      struct { int a, b; } x, y;
```

1. Field-sensitive approach models each field of each struct variable, hence creating four nodes (we use *node* to denote a pointer, variable or memory location) – $x.a$, $x.b$, $y.a$ and $y.b$.

2. Field-insensitive approach models each struct variable, but does not model their fields. This example is modeled by two nodes – $x.*$ and $y.*$.

3. Field-based approach models each field without modeling the struct variables. This example is modeled by two nodes – $*.a$ and $*.b$.

The same principle applies when dealing with arrays. Consider a C integer array *int a[10]*.

Field-insensitive approach models this with only one node: *a[*]*, while field-sensitive approach creates ten nodes: *a[0]*, *a[1]*, ..., *a[10]*.

Clearly a field-sensitive approach provides a more fine grained model and hence better precision. However, the number of nodes increases rapidly when there are nested structs and/or arrays.

2.2 Intra-Procedural v.s. Inter-Procedural

An intra-procedural alias analysis analyzes the bodies of each functions. It does not consider how each function interact with other functions. Specifically, intra-procedural alias analysis does not handle pointer parameter passing or functions that return pointers. On the contrary, inter-procedural alias analysis deals with the pointer behaviors due to function calls.

A pseudo-code where pointer parameter passing is involved:

```
1    void fn1(int* p) { p = ... }
2    void fn2() { int *q; fn1(q); }
```

A pseudo-code where function is returning pointers:

```
1    int* id(int* p){ return p; }
2
3    void fn(){
4      int *q;
5      q = id(q);
6    }
```

Intra-procedural is less expansive to perform, but has lower precision. It is often easier to implement an intra-procedural alias analysis before extending to inter-procedural alias analysis. Intra/inter-procedural property is highly related to context-sensitivity since a context-sensitive analysis has to be an inter-procedural analysis.

2.3 Context-Sensitivity

Context-sensitivity governs how function calls are analyzed. This property yields two types of alias analyses ? *context-sensitive* and *context-insensitive* alias analyses. Context-sensitive analysis considers the calling context (caller) when analyzing the target of a function call (callee). Consider the following code:

```
1     int a,b;
2     int *x;
3
4     void f(void) { *x++; }
5
6     void main() {
7       x = &a;
8       f();
9
10      x = &b;
11      f();
12    }
```

In this code, *function-f* is called twice. It increases the value of *variable-a* the first time it was called, and increases the value of *variable-b* the second time it was called. A context-sensitive alias analyzer needs to have a way to create an abstract description for *function-f* so that every time it is called, the analyzer can apply the calling context to the abstract description.

Context-sensitive provides a finer grain model of the static code hence results in higher precision. However, it increases the complexity of the analysis.

2.4 Flow-Sensitivity

Flow-sensitivity is the principle that governs wether or not an analysis takes the order of code into account. There are *flow-sensitive* and *flow-insensitive* analyses.

A flow-insensitive analysis produces one set of alias result for the entire program it analyzes. This result is the sets of memory locations that pointers may points to at any point of the program.

It does not consider the order of the code. A flow-sensitive analysis computes alias information at every point of the program. Consider the following code:

```
1       int a, b;
2       int *p;
3       p = &a;
4       p = &b;
```

The result of a flow-insensitive analysis would be: *pointer-p* may points to *variable-a* or *variable-b*. A flow-sensitive analysis is capable to determine that between line 3 and line 4, *pointer-p* points to *variable-a*, and after line 4, *pointer-p* points to *variable-b*.

Notice that the complexity of flow-sensitive analysis increases tremendously when a program has many conditional statements, loops or recursive functions. A complete control flow graph is required in order to perform flow-sensitive analysis. Therefore flow-sensitive analysis is much more precise, but is too expansive for most cases to perform on a whole program.

Chapter 3

Alias Analysis Algorithms

Alias analysis is an active research area due to its important applications noted in the introduction. Algorithms for each type of alias analyses have been developed over the past two decades. Each of them has distinct features that make them valuable to certain applications. They are presented in the following sections.

3.1 Andersen's Points-To Analysis

Andersen's points-to analysis was proposed by Lars Ole Andersen in 1994 [1]. It is inter-procedural, flow-insensitive and context-insensitive. The fundamental idea is to transform programs into sets of *subset constraints*, and solve these constraints for the points-to results. The transformation from program to constraint is defined in the following table.

Constraint type	Assignment	Constraint	Meaning
Base	$a = \&b$	$a \supseteq \{b\}$	$loc(b) \in pts(a)$
Simple	$a = b$	$a \supseteq b$	$pts(a) \supseteq pts(b)$
Complex	$a = *b$	$a \supseteq *b$	$\forall v \in pts(b), pts(a) \supseteq pts(v)$
Complex	$*a = b$	$*a \supseteq b$	$\forall v \in pts(a), pts(v) \supseteq pts(b)$

Table 3.1: Four Andersen constraint types

There are two approaches to implementing Andersen's points-to analysis – compute the points-to sets directly as one works through each constraint; or cast into graph and compute the closure.

The complexity of Andersen's points-to analysis is $O(n^3)$ where n is the number of nodes (pointers). The time required to run Andersen's points-to analysis grows exponentially as the

program size grows since the number of nodes (pointers) tends to grow as program size increases. There are many efforts to optimize it. Cycle elimination is one important method as proposed in [4] and [3].

3.2 Steensgaard's Points-To Analysis

Steensgaard's algorithm is very similar to Andersen's approach. It is also inter-procedural, flow-insensitive and context-insensitive. It was proposed in 1996 by Bjarne Steensgaard [8]. Steensgaard's algorithm also transforms programs into constraints, and solves the sets of constraints to obtain points-to results. The main difference is that instead of collecting subset constraints, it collects equivalence constraints.

Constraint type	Assignment	Constraint	Meaning
Base	$a = \&b$	$a \supseteq \{b\}$	$loc(b) \in pts(a)$
Simple	$a = b$	$a = b$	$pts(a) = pts(b)$
Complex	$a = *b$	$a = *b$	$\forall v \in pts(b), pts(a) = pts(v)$
Complex	$*a = b$	$*a = b$	$\forall v \in pts(a), pts(v) = pts(b)$

Table 3.2: Four Steensgaard constraint types

These constraints are much simpler, but produce less precise model. They can be modeled using the *disjoint-set* data structure, and can be solved efficiently using the *Union-Find* algorithm. The implementation detail is covered in Chapter 5. The complexity is nearly linear with $O(n\alpha(n))$ where n is the number of nodes and $\alpha(n)$ is the inverse Ackermann function. This makes Steensgaard highly scalable.

3.3 Data Structure Analysis (DSA)

Data structure analysis is a Steensgaard style (unification based) pointer analysis algorithm. It is flow-insensitive but context-sensitive and field-sensitive. It was first proposed by Chris Lattner in 2007 [7]. Full context-sensitive and field-sensitive pointer analysis were thought to be too expansive to perform for practical uses. DSA is able to achieve a scalable and fast context-sensitive and field-sensitive pointer analysis by giving up context-sensitivity within strongly connected components of the call graph.

Context-sensitivity and field-sensitivity are obtained by performing *heap cloning*. Heap cloning creates an abstract description for each data structure or function, and such abstract description is instantiated when they are called. By doing this, we are able to model different instances of a data structure created at different places in a program.

DSA can be performed in three phases: (1) local analysis phase, (2) bottom-up analysis phase and (3) top-bottom analysis phase. In the local analysis phase, a *Local Data Structure Graph* is computed for each function. It is a summary of the memory objects accessible within the function. The bottom-up analysis phase inlines the caller DS graph with the callee's information. The top-bottom phase fills in incomplete argument information by merging caller DS graphs with callee DS graphs.

Context-sensitivity and field-sensitivity approach leads to significant pointer analysis precision gain. It has been shown that DSA can be as precise as Andersen's for many benchmark cases [7].

Chapter 4

Alias Analysis in LLVM

LLVM is a collection of toolchains of compiler components. LLVM began as a research project at the University of Illinois. It is open source and licensed under the "UIUC" BSD-Style license. It is designed to support compiler implementation of arbitrary language. LLVM has a well defined LLVM intermediate representation (LLVM IR). Any compiler implemented on top of LLVM generates IR code from the source language. Clang and Clang++ are the two most well known and widely used C/C++ compilers implemented on LLVM. LLVM provides a set of libraries and some builtin *passes* that can perform optimizations, code transformation and static analysis. The powerful LLVM Core libraries makes it simple for developers to develop their own passes. For this reason, we are investigating the alias analysis implementations on LLVM, and also developing our own alias analysis implementation on LLVM.

The most recent LLVM release (LLVM 3.6) ships with only one alias analysis pass included. It is the basic alias analysis (-basic-aa). Basic-aa is a rule based alias analysis. It uses the following simple but important rules to compute alias information:

- 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 references stack allocations if they never escape from the function that allocates them (a common case for automatic arrays).

In LLVM's earlier releases, it included a Steensgaard's alias analysis pass, and a DSA pass. However, Steensgaard's algorithm was patented by Microsoft, and DSA algorithm uses Steensgaard's algorithm. They were removed from LLVM in 2006 due to patent issues and lack of maintenance.

Jia Chen, a researcher at the University of Texas at Austin has developed an Andersen alias analysis implementation for the current LLVM release (3.5/3.6). In addition, there is a group of developers who are working to bring DSA alias analysis to current release. In my research, I implemented Steensgaard's algorithm on LLVM 3.5, hich will be presented later in detail.

Chapter 5

Implementation of Steensgaard's Algorithm

Steensgaard's algorithm as mentioned earlier, can be efficiently implemented using the disjoint-set data structure and the Union-Find algorithm. Disjoint-set data structure partitions a set of nodes into disjoint subsets, and Union-Find algorithm provides two operations: union and find. Union operation merges two disjoint sets into one set, and find operation determines the representative node of the subset in which any given node belongs to. When optimization methods are used, Union-Find can achieve $O(\alpha(n))$ which is almost constant in practice.

Using the disjoint-set data structure, we first create a node for each pointer found in the program. We refer to the collection of all nodes as the *universe*. We use the *root node* of each set as the *representative node*. So initially every node is the representative node of its own set, and all sets are disjoint. Every representative node has a pointer pointing to the node that the set represented by it points to. The points-to of each node is initialized to NULL as we do not have any points-to knowledge, yet. We then walk through the program statement by statement. Nodes are then merged and points-to relation updated according to the four types of constraints as described below.

1. *Base constraint* $a = \&b$: merge *node-b* with the set of nodes that *node-a* is pointing to and update the points-to of *node-a* to the new representative node of the merged set. Furthermore, when a merge is performed, we recursively merge downward in the sense that if

node-p and *node-q* are merged, we merge the *points-to of node-p* and the *points-to of node-q*. If *node-a* was originally not pointing to any node, then we add a points-to relation from *node-a* to *node-b*.

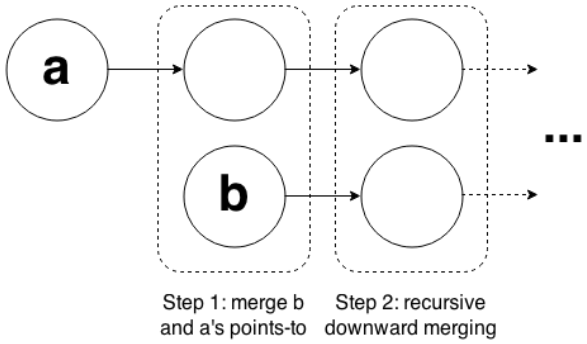

Figure 5.1: Steensgaard's *base* constraint

2. *Simple constraint* $a = b$: merge the points-to of *node-a* and the points-to of *node-b* and perform recursive downward merging.

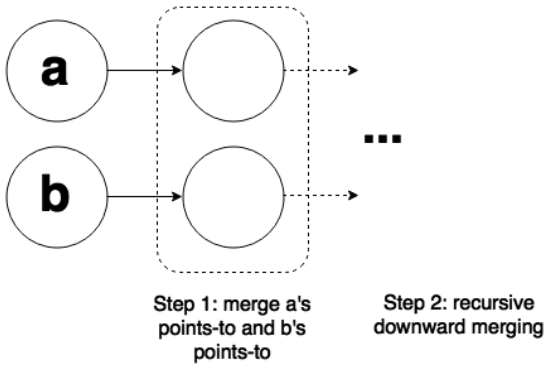

Figure 5.2: Steensgaard's *simple* constraint

3. *Complex constraint type 1* $a = *b$: merge the points-to of *node-a* with the points-to of the points-to of *node-b* and perform recursive downward merging.

   (a) If *node-b* was originally not pointing to any node, then we add a points-to relation from *node-b* to *node-a*.

   (b) If both *node-b*'s points-to node and *node-a* were originally not pointing to any node, then we create a *dummy node* and let both *node-b*'s points-to node and *node-a* point

to this *dummy node*.

(c) If *node-b*'s points-to node was originally not pointing to any node while *node-a* has a points-to, then we add a points-to relation from *node-b*'s points-to to *node-a*'s points-to.

(d) If *node-a* was originally not pointing to any node while *node-b*'s points-to has a points-to, then we add a points-to relation from *node-a* to *node-b* points-to's points-to.

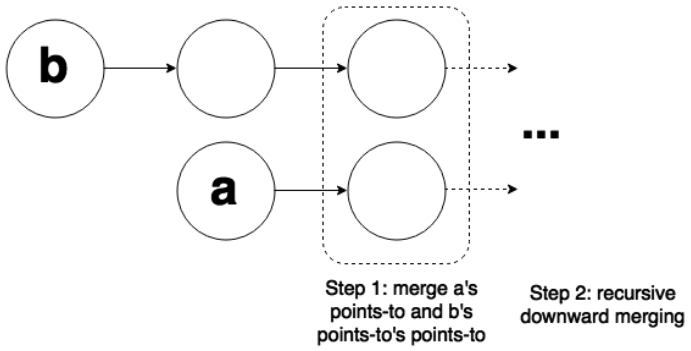

Figure 5.3: Steensgaard's $complex_1$ constraint

4. *Complex constraint type 2* $*a = b$: merge *node-b* with the *points-to set of node-a* and perform recursive downward merging.

   (a) If *node-a* was originally not pointing to any node, then we add a points-to relation from *node-a* to *node-b*.

   (b) If both *node-a*'s points-to node and *node-b* were originally not pointing to any node, then we create a *dummy node* and let both *node-a*'s points-to node and *node-b* point to this *dummy node*.

   (c) If *node-a*'s points-to node was originally not pointing to any node while *node-b* has a points-to, then we add a points-to relation from *node-a*'s points-to to *node-b*'s points-to.

   (d) If *node-b* was originally not pointing to any node while *node-a*'s points-to has a points-to, then we add a points-to relation from *node-b* to *node-a* points-to's points-to.

The challenge of implementing Steensgaard's algorithm on LLVM is that we need to generate constraints from LLVM IR instead of from source languages such as C. We need to find

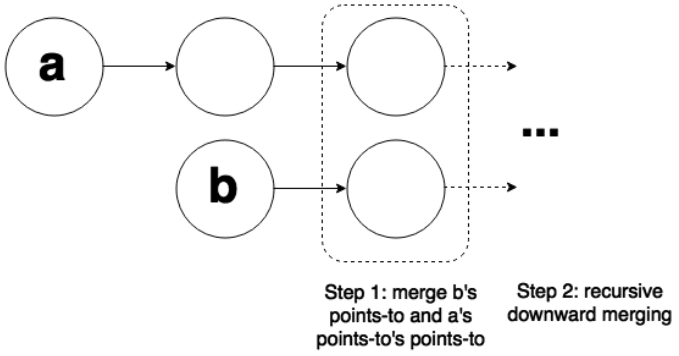

Figure 5.4: Steensgaard's $complex_2$ constraint

a correspondence between LLVM IR instruction and Steensgaard's constraints and perform the appropriate actions for each LLVM IR instruction.

5.1 Store Instruction

An LLVM IR store instruction is used to write value to memory. It has syntax:

```
store <ty> <value>, <ty>* <pointer>
```

The following C source code

```
1    int x;
2    int *p;
3    p = &x;
```

when compiled to LLVM IR using clang generates the following instructions:

```
1      %x = alloca i32, align 4
2      %p = alloca i32*, align 8
3      store i32* %x, i32** %p, align 8
```

On *line 3* of this IR code, the memory address of x is written to $pointer - p$ (i.e. $pointer - p$ now points to x). The *store instruction* is equivalent to Steensgaard's *Complex constraint type 2* $*a = b$. To model it, we merge *value-node* with the set of nodes that *pointer-node* is pointing to and update the points-to of *pointer-node* to the new representative node of the merged set, and

we recursively merge downwards. If *pointer-node* was originally not pointing to any node, then we add a points-to relation from *pointer-node* to *value-node*.

5.2 Load Instruction

An LLVM IR store instruction is used to read value from memory. It has syntax:

```
<result> = load <ty>, <ty>* <pointer>
```

The following C source code

```
1    int *p;
2    *p = 0;
```

when compiled to LLVM IR using clang generates the following instructions:

```
1      %p = alloca i32*, align 8
2      %0 = load i32** %p, align 8
3      store i32 0, i32* %0, align 4
```

On *line 2* of this IR code, p's pointing memory address is loaded to %0, and on line 3 the constant value 0 is written to %0. We can treat the load instruction as an one level dereferencing. %0 in this example is conceptually equivalent to $*p$ in the source language. In this sense, the load instruction is equivalent to Steengaard's *complex constraint type 1* $a = *b$. To model the load instruction, merge *result-node* with the *points-to set of pointer-node* and perform recursive downward merging. If *pointer-node* was originally not pointing to any node, then we add a points-to relation from *pointer-node* to *result-node*.

5.3 Call Instruction

Since we are implementing an inter-procedural Steensgaard, we need to handle function call instructions. The call instruction has syntax:

```
<result> = call <ty> <fnptrval>(<function args>)
```

The *result* is the return value of the function call with type *ty*. *fnptrval* is the identifier of the called function, and *function args* is a list of arguments that gets passed in to the function. The following C code:

```
1   int g;
2   void f(int *fp) { *fp =10; }
3   int main(){
4     int *p;
5     p = &g;
6     f(p);
7     return 0;
8   }
```

when compiled to LLVM IR:

```
1   @g = global i32 0, align 4
2
3   ; Function Attrs: nounwind uwtable
4   define void @_Z1fPi(i32* %fp) #0 {
5   entry:
6     %fp.addr = alloca i32*, align 8
7     store i32* %fp, i32** %fp.addr, align 8
8     %0 = load i32** %fp.addr, align 8
9     store i32 10, i32* %0, align 4
10    ret void
11  }
12
13  ; Function Attrs: nounwind uwtable
14  define i32 @main() #0 {
15  entry:
16    %retval = alloca i32, align 4
17    %p = alloca i32*, align 8
18    store i32 0, i32* %retval
19    store i32* @g, i32** %p, align 8
20    %0 = load i32** %p, align 8
21    call void @_Z1fPi(i32* %0)
```

```
22        ret i32 0
23      }
```

In the source language, we have an actual parameter *int *p* (line 6) when calling *function f*, and we have *int *fp* (line 2) as a formal parameter. Function call in C is strictly call-by-value, that is, in the beginning of a function call, the value of the actual parameter gets *copied* to the value of the formal parameter. In this case, at line 20 in the IR, the memory address that p is pointing to is loaded to %0. %0 then gets copied to the formal parameter $\%fp$ (line 21 and 4). Notice that $\%fp$ now has the memory address that p is pointing to, not the memory address of p. At line 6, a pointer $\%fp.addr$ is created, and $\%fp$ is stored to this new memory location.

This parameter passing mechanism is equivalent to Steensgaard's *simple constraint* $a = b$. To model it, we inspect the call instruction, find the function that is being called, merge the actual parameters and formal parameters nodes as we do for simple constraint, and create points to relation according to the load and store instruction.

In this example, we add a points to relation from $\%p$ to %0 (line 20). We merge %0 with $\%fp$. Then we add a points to relation from $fp.addr$ to fp. As result, $\%p$ and $\%fp.addr$ are aliases because they both point to %0 and $\%fp$.

Now to handle functions that return pointers, consider this example:

```
1       int a, b;
2       int* f() {
3         if (a > 10)
4           return &a;
5         else
6           return &b;
7       }
8       int main(){
9         int *p;
10        p = f();
11        return 0;
12      }
```

This C code when compiled to LLVM IR generates:

```
1   @a = global i32 0, align 4
2   @b = global i32 0, align 4
3
4   ; Function Attrs: nounwind uwtable
5   define i32* @_Z1fv() #0 {
6   entry:
7     %retval = alloca i32*, align 8
8     %0 = load i32* @a, align 4
9     %cmp = icmp sgt i32 %0, 10
10    br i1 %cmp, label %if.then, label %if.else
11
12  if.then:
          ; preds = %entry
13    store i32* @a, i32** %retval
14    br label %return
15
16  if.else:
          ; preds = %entry
17    store i32* @b, i32** %retval
18    br label %return
19
20  return:
          ; preds = %if.else, %if.then
21    %1 = load i32** %retval
22    ret i32* %1
23  }
24
25  ; Function Attrs: nounwind uwtable
26  define i32 @main() #0 {
27  entry:
28    %retval = alloca i32, align 4
29    %p = alloca i32*, align 8
30    store i32 0, i32* %retval
31    %call = call i32* @_Z1fv()
32    store i32* %call, i32** %p, align 8
```

```
33        ret i32 0
34      }
```

Note that, at line 31 in the IR code, in opposed to the previous example, the function return type is now $i32*$ (i.e. a 32-bit integer pointer). In this example, *function f* has two return statements (line 4 and 6) in the C code. In the IR code, however, there is only one return instruction (line 22) for *function f*. The IR return node %1's value depends on the control flow. In reality a function may have any number of return statements. Because we are implementing a flow-insensitive alias analysis, without knowing anything about the control flow of the program, we have to treat every return statement as a possible candidate. This causes imprecision in our alias analysis result, however, it is inevitable for flow-insensitive alias analyses.

The call instruction of function returning pointer is also equivalent to the Steensgaard's *simple constraint*. To model it, we first find all return nodes in the called function by linearly scan through all instructions. We then merge the result node of the call instruction with every return nodes in the function being called.

In this particular example, we merge $\%call$ with %1. Combining with previously described modeling method for load and store instructions, we have: $\%p$ pointing to $\%call$ (line 32), $\%retval$ pointing to {@a, @b} (line 13 and 17), and $\%retval$ pointing to {%1, $\%call$} (line 21, 22 and 31). Furthermore, since {@a, @b} and {%1, $\%call$} are both pointed by $\%retval$, we merge the two sets so that $\%retval$ points to {@a, @b, %1, $\%call$}.

In result, we know that $\%p$ and $\%retval$ both point to the set {@a, @b, %1, $\%call$}, i.e., they are aliases.

5.4 GetElementPtr Instruction

The LLVM IR GetElementPtr (Get Element Pointer) instruction is used to get the address of a subelement of an aggregate data structure (i.e. *self-defined struct* or *C array*). It performs address calculation only and does not access memory. GetElementPtr has syntax:

```
<result> = getelementptr inbounds <ty>* <ptrval>{, <ty> <idx>}*
```

The first <*ty*> defines the type of the aggregate data structure <*ptrval*>. The following list of <*ty*> <*idx*> indicates the indexes of elements that we are interested in. The calculated address is then stored in <*result*>. Consider the following C source code:

```
1       int a[10];
2       int *p;
3       p = &a[2];
```

when compiled to LLVM IR, it becomes:

```
1       %a = alloca [10 x i32], align 16
2       %p = alloca i32*, align 8
3       %arrayidx = getelementptr inbounds [10 x i32]*
          %a, i32 0, i64 2
4       store i32* %arrayidx, i32** %p, align 8
```

At line 3, notice that the aggregate data structure we are interested in is the *integer array a*. It has type *[10 x i32]** (i.e. a pointer to ten 32-bit integers). The first term in the list at the end *i32 0* indexes the first term of $\%a$. The second term in the list *i64 2* indexes the term that we are interested (i.e. $a[2]$). As a result, this instruction returns the address of $a[2]$.

While it is possible to implement a field-sensitive Steensgaard, we started with a field-insensitive version for simplicity. Field-insensitivity allows us to ignore fields in aggregate data structures and different indexes in arrays. For this reason, dealing with GetElementPtr is simple. It is equivalent to the Steensgaard's *simple constraint* where we simply merge the <*result*> *node* with the <*ptrval*> *node*.

In this example, we merge $\%arrayidx$ with $\%a$. From line 4 of the IR code, we know that $\%p$ points to $\%arrayidx$. But because we do not distinguish $\%arrayidx$ and $\%a$, our result would be that $\%p$ may point to any element in the *a array*.

5.5 PHI Instruction

LLVM IR uses static single assignment form (SSA) to represent variables. SSA form requires that every variable be defined before its use, and that each variable is assigned exactly once. This is achieved by splitting existing variables into many variations. Suppose given this code:

```
1       x = 1;
2       x = 2;
3       y = x;
```

when SSA is enforced, the code can be represented as:

```
1       x1 = 1;
2       x2 = 2;
3       y1 = x2;
```

SSA allows compilers to perform many kinds of optimizations such as constant propagation, value range propagation, sparse conditional constant propagation, dead code elimination, partial redundancy elimination, etc. However, when there are control branches in the program, we need some kind of mechanism to determine which branch was executed. Consider this code:

```
1       x = 0;
2       if (x > 0)
3         y = 1;
4       else
5         y = 2;
6
7       z = y;
```

when translated to SSA form, we have:

```
1       x1 = 0;
2       if (x1 > 0)
3         y1 = 1;
4       else
```

```
5         y2 = 2;
6
7       z1 = y?;
```

There is no way to statistically determine which branch was executed, hence the compiler would not know which y to use at line 7. The solution is to create a PHI node right before line 7. It creates a new variable $y3$ by choosing either $y1$ or $y2$ depending on the executed control path.

PHI node in LLVM IR has syntax:

```
<result> = phi <ty> [ <val0>, <label0>], ...
```

where <*result*> is the new SSA variable, <*ty*> is the type of the variable, and [<*val*0>, <*label*0>] is the list of SSA variables with labels of the corresponding control flow branches of each SSA variable.

When dealing with PHI instruction, we do not know which branch we arrive from because we are implementing flow-insensitive alias analysis. The PHI instruction is equivalent to the Steensgaard's *simple constraint*. To model the PHI instruction, we simply merge all SSA variables in the list with the *result node*.

5.6 BitCast Instruction

The BitCast instruction is used to convert variable of one type to another without changing any bit. The syntax is:

```
<result> = bitcast <ty> <value> to <ty2>
```

where <*result*> is the variable of new type, <*ty*> is the original type, <*value*> is the variable to cast from, and <*ty*2> is the new type. BitCast can handle, for example, pointer type castings in C programs. The following code:

```
1       char c = '0';
2       char *p = &c;
3       int *q = (int *)p;
```

when compiled to LLVM IR becomes:

```
1       %c = alloca i8, align 1
2       %p = alloca i8*, align 8
3       %q = alloca i32*, align 8
4       store i8 48, i8* %c, align 1
5       store i8* %c, i8** %p, align 8
6       %0 = load i8** %p, align 8
7       %1 = bitcast i8* %0 to i32*
8       store i32* %1, i32** %q, align 8
```

Steensgaard's alias analysis does not deal with data types. The BitCast instruction is equivalent to Steensgaard's *simple constraint*. To model the BitCast instruction, we simply merge <*value*> with <*result*>.

5.7 IntToPtr Instruction

IntToPtr (Integer To Pointer) is also a type casting instruction. As its name suggests, it converts an integer to a pointer. The syntax for IntToPtr is:

```
<result> = inttoptr <ty> <value> to <ty2>
```

where <*result*> is the returned pointer, <*ty*> is the original type, <*value*> is the integer, and <*ty*2> is the new type. IntToPtr can be generated when certain pointer type castings in C programs are compiled. The following code:

```
1     int a = 0;
2     int *p;
3     p = (int *)a;
```

when compiled to LLVM IR becomes:

```
1       %a = alloca i32, align 4
2       %p = alloca i32*, align 8
3       store i32 0, i32* %a, align 4
4       %0 = load i32* %a, align 4
```

```
5       %conv = sext i32 %0 to i64
6       %1 = inttoptr i64 %conv to i32*
7       store i32* %1, i32** %p, align 8
```

Similar to BitCast, IntToPtr is equivalent to Steensgaard's simple constraint. We model it by merging <*value*> and <*result*>.

5.8 Select Instruction

The Select instruction is used to choose one value base on condition without branching. The syntax for Select is:

```
<result> = select selty <cond>, <ty> <val1>, <ty> <val2>
```

If <*cond*> is *true* then choose <*val*1>, otherwise choose <*val*2>.The following C statement:

```
1     p > 10 ? 1: 2
```

when compiled to LLVM IR becomes:

```
1     %cond = select i1 %cmp, i32 1, i32 2
```

Since we do not know the result of the condition, it is equivalent to Steensgaard's *simple constraint*. We merge <*result*>, <*val*1> and <*val*2>.

5.9 Other Pointer Related Instructions

There are several other LLVM IR instructions that perform operations on pointers. They are *VAArg*, *ExtractValue*, *InsertValue*, *LandingPad*, *Resume*, *AtomicRMW* and *AtomicCmpXchg*. They are seldom used comparing to the instructions introduced earlier. These instructions do effect pointer relations, but they are not yet supported by our Steensgaard's implementation. Some of them are simple to handle while others are more challenging to handle (such as VAArg). We hope to add support for these instructions in the future.

Chapter 6

Evaluation

There are mainly three areas of interest when evaluating a pointer analysis: precision, time and memory usage. This chapter presents our evaluation results for the following four algorithm implementations:

1. Basic alias analysis implemented on LLVM 3.5
2. Andersen's pointer analysis implemented on LLVM 3.5
3. Data Structure alias analysis implemented on LLVM 3.5
4. Our own Steensgaard's pointer analysis implemented on LLVM 3.5

Source codes from the standard benchmark suite SPEC CPU2006 as well as some other well known open source projects were used for evaluation. Specifically, the benchmarks we used are shown in Table 6.1.

Benchmark	*Bitcode size*	*Number of pointers*
mcf	43KB	1014
astar	90KB	2182
gzip	171KB	1813
bzip2	187KB	3947
nginx	4.4MB	27663
gcc	5.7MB	231778

Table 6.1: Evaluation benchmarks

6.1 Precision

There exists many different metrics for evaluating the precision of a pointer analysis. Some intuitive metrics include

1. *Number of pointer-equivalent classes*: higher number means better precision since more pairs of pointers are determined to be non-aliases.

2. *Average size of the points-to set of all pointers*: the points-to set is the set of variables that a pointer may point to. Larger size of such sets means that the analysis is unable to determine precise points-to information hence lower precision.

3. *Percentage of may point to pointer pairs over all pointer pairs*: it is similar to the average size of points-to set metric. It computes the number of pointer pairs that result in *may-point-to* and divides this number with the number of all possible pointer pairs. Lower percentage means higher analysis precision.

We choose to use the may-point-to percentage as our precision metric because it is a built-in pass in LLVM 3.5 (the *aa-eval pass*). This allows us to compare different alias analysis implementations on a fair perspective since all of our alias analyses are implemented on LLVM 3.5.

Our experiment result is shown in Table 6.2[1]. We found that basic alias analysis can achieve pretty decent precision when analyzing smaller programs. This is mainly contributed by its pseudo context sensitive and pseudo field sensitive rules. However, the effect of basic alias analysis' rules diminishes as the size of the program grows.

Notice that in Table 6.2, the precision of Andersen's pointer analysis is strictly better than the precision of Steensgaard's pointer analysis. This is somewhat an indication that both the Andersen's and Steensgaard's are correctly implemented. Andersen's subset based approach is more precise than Steensgaard's unification based approach by definition. In addition, notice that DSA also performs strictly better than Steensgaard's in all benchmark cases. This is the case because DSA is an Steensgaard's style (unification based) alias analysis with added context sensitivity and field sensitivity.

[1]Some values are missing because they took too long to compute.

Furthermore, DSA is shown to be slightly more precise than Andersen's pointer analysis on all benchmarks. This result confirms the result obtained by [7] in which the author claims DSA can be as precise as Andersen's for many benchmark cases and slightly more precise than Andersen's for some other cases.

Benchmark	*Bitcode size*	*# pointers*	*Basic-AA*	*Andersen's*	*Steensgaard's*	*DSA*
mcf	43KB	1014	85.7%	74.9%	91.6%	70.8%
astar	90KB	2182	50.6%	49.3%	90.1%	38.8%
gzip	171KB	1813	39.3%	35.6%	52.4%	31.9%
bzip2	187KB	3947	72.7%	88.1%	96.1%	86.3%
nginx	4.4MB	27663	89.5%	87.9%	88.3%	87.2%
gcc	5.7MB	231778	95.9%	—*	—*	82.9%

Table 6.2: May-alias percentage comparison

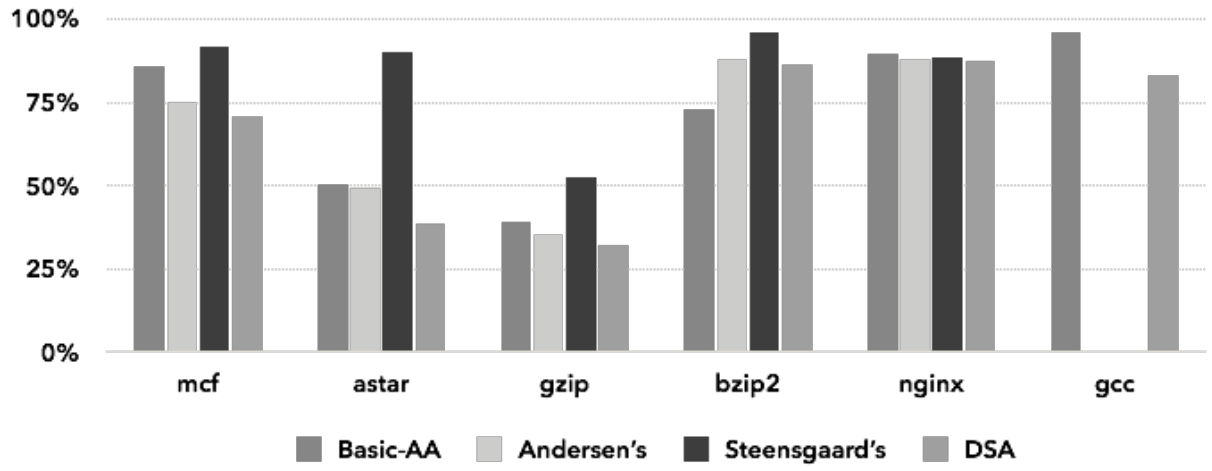

Figure 6.1: May-alias percentage comparison

An advantage of implementing alias analysis on LLVM is that it is easy to chain multiple passes. We can perform one alias analysis followed by a different alias analysis in the sense that whenever the first alias analysis reports *may alias*, we pass the pair of pointers to the next alias analysis and see if it can derive further information about the pair of pointers.

In this research, we evaluated the precisions when chaining basic alias analysis with Andersen's, basic alias analysis with Steensgaard's and basic alias analysis with DSA. The result is presented in Table 6.3[2].

[2]Some values are missing because they took too long to compute.

Benchmark	*Bitcode size*	*# pointers*	*Basic+Anders*	*Basic+Steens*	*Basic+DSA*
mcf	43KB	1014	65.9%	80.4%	62.5%
astar	90KB	2182	27.7%	47.1%	20.4%
gzip	171KB	1813	14.5%	21.0%	11.3%
bzip2	187KB	3947	69.1%	69.5%	67.4%
nginx	4.4MB	27663	82.0%	80.2%	81.0%
gcc	5.7MB	231778	—*	—*	81.3%

Table 6.3: Chained may-alias percentage comparison

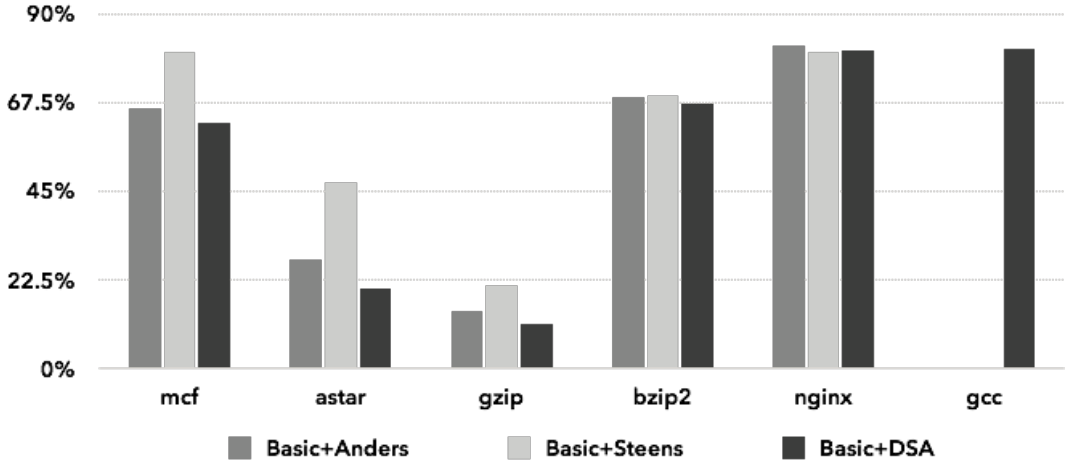

Figure 6.2: Chained may-alias percentage comparison

By chaining basic alias analysis, we see an average precision gain of 12 percent for DSA, 15 percent for Andersen's and 24 percent for Steensgaard's. This result shows that Andersen's, Steensgaard's and DSA can achieve similar precision when chained with basic alias analysis (especially on larger benchmarks). In the next section, we evaluate the execution time of each analysis for analyzing the benchmarks.

6.2 Time

We use the LLVM built-in pass *-time-passes* to measure the execution time of each of our alias analysis passes on all benchmarks.

We compare the number of nodes versus execution time instead of bitcode size because the theoretical time complexity of the algorithms are expressed in terms of number of nodes. Also note that the y-axis in Figure 6.2 is in logarithmic scale for readability. Our experiment shows that the

Benchmark	*Bitcode size*	*# pointers*	*Basic-AA*	*Andersen's*	*Steensgaard's*	*DSA*
mcf	43KB	1014	0.01 sec	0.03 sec	0.08 sec	0.03 sec
astar	90KB	2182	0.04 sec	0.15 sec	0.51 sec	0.08 sec
gzip	171KB	1813	0.05 sec	0.05 sec	0.46 sec	0.13 sec
bzip2	187KB	3947	0.08 sec	0.16 sec	1.9 sec	0.17 sec
nginx	4.4MB	27663	0.5 sec	47 min	1.2 min	11.7 sec
gcc	5.7MB	231778	2.8 sec	1320 min	50 min	4.0 min

Table 6.4: Analysis time comparison

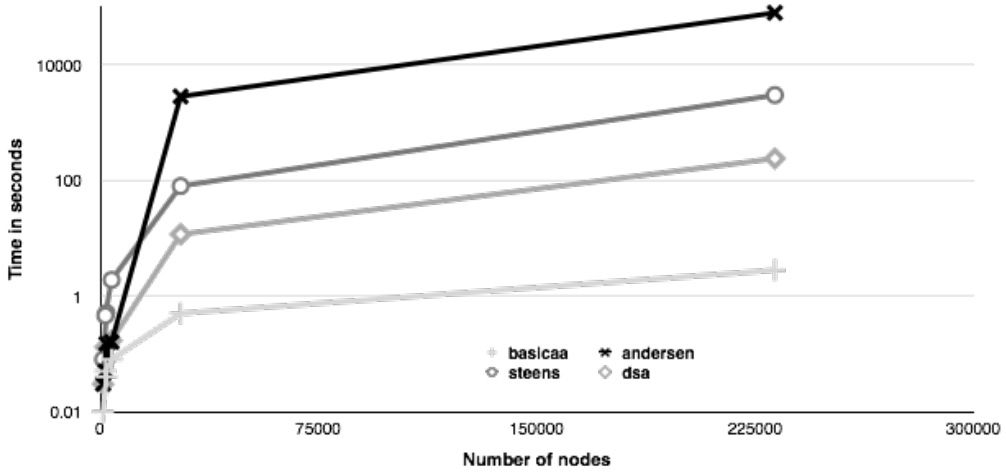

Figure 6.3: Analysis time comparison

time complexity of Andersen's implementation in practice can be as fast as $O(N^2)$ (better than its theoretical complexity $O(N^3)$). Our implementation of Steensgaard's shows almost linear time complexity and so does the DSA while basic alias analysis shows sub-linear time performance.

6.3 Memory

We use the analysis tool *Valgrind* to measure the memory usage. We record the total heap allocation size of each algorithm when analyzing the benchmarks. The result is shown in Table 6.5.[3]

As a result, DSA uses the most memory. However, the memory usage of DSA on all benchmarks are shown to be practical on average modern computing environment. Andersen's pointer analysis uses less memory than DSA, Steensgaard's pointer analysis uses less memory than Andersen's, and basic alias analysis uses the least amount of memory when analyzing programs. Figure 6.3 shows that even in the case of DSA, memory usage grows linearly with the number of

[3]Some values are missing because they took too long to compute.

Benchmark	*Bitcode size*	*# pointers*	*Basic-AA*	*Andersen's*	*Steensgaard's*	*DSA*
mcf	43KB	1014	1.53 MB	2.34 MB	1.72 MB	2.75 MB
astar	90KB	2182	2.83 MB	4.66 MB	3.26 MB	5.98 MB
gzip	171KB	1813	4.96 MB	6.03 MB	5.45 MB	9.19 MB
bzip2	187KB	3947	6.18 MB	9.69 MB	7.01 MB	9.73 MB
nginx	4.4MB	27663	118 MB	—*	123 MB	323 MB
gcc	5.7MB	231778	172 MB	—*	—*	2336 MB

Table 6.5: Memory usage comparison

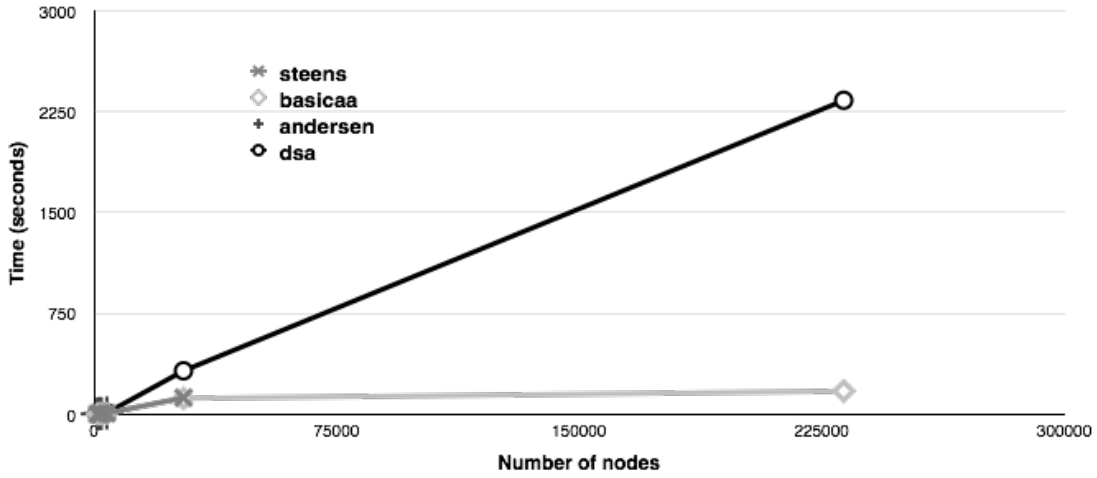

Figure 6.4: Memory usage comparison

nodes.

Chapter 7

Conclusion

This research provides a comprehensive study of alias analysis. We implemented a Steensgaard's pointer analysis at LLVM IR level on LLVM 3.5. We evaluated the precision, time and memory usage of rule based alias analysis, Andersen's and Steensgaard's pointer analysis and data structure analysis. As a result, we found that DSA combined with basic alias analysis provides the best precision among the algorithms we studied. In addition, we found DSA to be the fastest algorithm (except basic alias analysis) among all. Even though DSA uses the most memory when performing analysis, the amount of memory needed is acceptable for most modern computers. Our implementation of Steensgaard's pointer analysis performed worse than DSA in terms of precision and speed, but it is the only implementation of this theoretically very important algorithm in the current LLVM release. It provides a good point of reference for future studies on LLVM alias analysis.

Bibliography

[1] L. Andersen. Program Analysis and Specialization for the C Programming Language. Master's thesis, DIKU, University of Copenhagen, Copenhagen, Denmark, 1994.

[2] M. Das. Unification-based pointer analysis with directional assignments. *ACM Programming language design and implementation*, 35(5):35–46, 2000.

[3] B. Hardekopf and C. Lin. Exploiting Pointer and Location Equivalence to Optimize Pointer Analysis. *International Static Analysis Symposium*, pages 265–280, 2007.

[4] B. Hardekopf and C. Lin. The Ant and the Grasshopper: Fast and Accurate Pointer Analysis for Millions of Lines of Code. *ACM Programming language design and implementation*, 42(6):290–299, 2007.

[5] M. Hind. Pointer Analysis: Haven't We Solved This Problem Yet? *ACM Program analysis for software tools and engineering*, pages 54–61, 2001.

[6] M. Hind, M. Burke, P. Carini, and J. Choi. Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages and Systems*, 21(4):848–894, 1999.

[7] C. Lattner, A. Lenharth, and V. Adve. Making Context-sensitive Points-to Analysis with Heap Cloning Practical For The Real World. *PLDI*, 2007.

[8] B. Steensgaard. Points-to Analysis in Almost Linear Time. *Principles of programming languages*, pages 32–41, 1996.

Biography

Sheng-Hsiu Lin received his Bachelor of Science in Mathematics from Lehigh University in 2011. He received his Master of Engineering in Energy Systems Engineering from Lehigh University in 2013. He is expected to receive his Master of Science in Computer Science from Lehigh University in 2015.