NORTHWESTERN UNIVERSITY

A Better Memory Understanding for Program Dependence Graph through Static Value-Flow Analysis

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

MASTER OF SCIENCE

Field of Computer Science

By

Yian Su

EVANSTON, ILLINOIS

June 2020

ABSTRACT

A Better Memory Understanding for Program Dependence Graph through Static Value-Flow Analysis

Author: Yian Su

Advisor: Simone Campanoni

Committee Members: Peter Dinda

Modern compiler engineers actively seeking ways to achieve performance and energy efficiency by making aggressive code transformation on the source program based on the information extracted and derived through both static and dynamic code analysis. *Program Dependence Graph (PDG)*, an representation which captures both the control and data dependence of the source program, is a powerful representation to support various code transformation techniques.

The construction of program dependence graph requires several code analyses. One of the analysis, *Pointer Analysis*, is a technique to statically analyze the behavior of pointer operations in the source program, whose result will then be used to determine the memory data dependence of the program dependence graph.

Given the complexity of pointer analysis, most pointer analysis applications provide intraprocedural, flow-insensitive, context-insensitive and field-insensitive pointer analysis results. These results are correct but with less precision, therefore introduces many false positive memory data dependence in the program dependence graph, which prohibits many powerful code transformation.

This thesis describes the integration of *Static Value-Flow Analysis (SVF)*, a research project that leverages the advance in sparse pointer analysis that provides both scalable and more precise pointer analysis results, into a parallelizing framework – *NOELLE* and demonstrates how false positive memory data dependence can be successfully removed.

Furthermore, to preserve the program dependence graph constructed, we implement *PDG Embedding* and *PDG Loading*. This allows the program dependence graph constructed through a time-consuming analysis to be stored and attached to the LLVM bitcode. Further construction of program dependence graph can now be read directly from the bitcode.

We apply static value-flow analysis into *NOELLE* which requires program dependence graph construction with interprocedural, flow-insensitive, context-insensitive and field-sensitive pointer analysis, and evaluate and compare on the number of memory data dependence exists before and after. The result shows by average, 25% of false memory data dependence gets removed. We also measure the time elapsed to embed and load the program dependence graph to and from the bitcode. The result shows that for a large program with over $950K$ nodes and $18M$ edges, these two processes can be completed within 1 and 2 minutes respectively.

Acknowledgements

I want to sincerely express my appreciation to my advisor, Professor Simone Campanoni for his support, patience, trust and respect. He led me into the world of research and convincingly guided and encouraged me to be professional. Without his constantly help, this thesis would not be realized.

I wish to acknowledge my girlfriend, Yulin, who has been a continuous source of support, encouragement and understanding. Particularly, for the time when I was stressful and frustrated, she has not just tolerated me, but loved me with humor and joyousness.

Thanks to my parents, who provide the opportunity for me to study abroad and trust me to follow my originality into research. This work would not be possible without the support from you both financially and emotionally.

I also want to thanks to all my friends, for their warm regards during this pandemic and quarantine time. Especially gratitude to Phillip and Cindy for their constantly checking in to make sure everything goes well.

Finally, thank you to my two cats, Mew and Kelly, for all the fun and happiness they provided during this journey.

Table of Contents

List of Tables

List of Figures

CHAPTER 1

Introduction

In this chapter, We first elaborate the goal for compiler engineers, we describe what a compiler does and the concept of *Code Transformation* and *Code Analysis* and how code analysis and transformation helps compiler engineers to achieve their goals. In the end, the organization of the thesis is given.

1.1. Ultimate Goal for Compiler Engineers

Performance Efficiency and *Energy Efficiency* are the two ultimate goals for compiler engineers. Given a program typically performs computation and generate the output. The two goals can then defined as follows:

- *Performance Efficiency*: for the program compiled, the less use of time to compute the output is expected.
- *Energy Efficiency*: for the program compiled, the less use of computation resources, powers to compute the output is expected.

1.2. Code Transformation

In order to achieve both performance and energy efficiency, compiler engineers perform various of aggressive code transformations and optimizations on the source program to generate output faster and use less energy.

```
{                    {
  ...                  ...
  a = 10;              a = 10;
  return a;   =>       return 10;
}                    }
```

Figure 1.1. Example of Constant Propagation

```
{                    {
  ...                  ...
  *p = 10;             *p = 10;
  return *p;  =>       return 10;
}                    }
```

Figure 1.2. Example of Constant Propagation through memory

An example, *Constant Propagation*, in Figure 1.1, substitute the value of variable with known constants compiler time, can be helpful since less registers are needed at the run time of program. Therefore, we can achieve energy efficiency since less computation resources are used.

1.3. Code Analysis

The decision to whether it is safe to make a code transformation or not, so that the output of a program can be preserved, depends on how much we understand about the program. The activity to collect information about the behavior of a program is called *Code Analysis*.

An advanced code analysis result can provide valuable insight of the source program to help making powerful code transformation. For instance, the constant propagation problem can be complicated when pointers and memory locations are introduced. In Figure 1.2, the decision to whether substitute $*p$ with value 10 is determined by the result

given by *Pointer Analysis*. An advanced pointer analysis can conclude that p points to only one memory location and the value stored is 10. Therefore the constant propagation can be performed.

However, the cost to perform an advance code analysis can be expensive in both time and memory usage. As a trade-off inside code analysis between precision, latency and memory consumption, a less precise pointer analysis may return that p can point to many other memory locations, which prohibits the code transformation.

1.4. Thesis Organization

The thesis is organized as follows, in chapter 2, we introduce *Program Dependence Graph*, an important representation of the source program to support various code transformation. In chapter 3, we show *Pointer Analysis*, an analysis used by program dependence graph and show how the false positive dependence gets introduced. In chapter 4, we introduce *Static Value-Flow Analysis* and dive deep into its workflow and show how the false positive dependence in the program dependence graph can be removed. In chapter 5, the integration of static value-flow analysis with a parallelizing framework – *NOELLE* is introduced. In chapter 6, we describe the concept and implementation of *PDG Embedding* and *PDG Loading* to LLVM bitcode. We evaluate the false positive memory dependence removed after leveraging static value-flow analysis and the time elapsed to perform PDG Embedding and PDG Loading on test suite *SPEC2017* in chapter 7. Finally, the conclusion and future works are given in chapter 6.

CHAPTER 2

Program Dependence Graph

This chapter introduces *Program Dependence Graph*, an intermediate graph representation of the original program. Firstly, data dependence and control dependence are explained respectively. Secondly, the usage of program dependence graph is described. We further discuss the correctness and precision of program dependence graph. Finally, the problem with existing program dependence graph is raised.

2.1. What is it?

Program dependence graph (PDG), which first introduced by Ferrante, Ottenstein and Warren [**1**], is an intermediate graph representation of the source program that makes both the control and data dependence explicit for each operation. The nodes are instructions or or function arguments and the edges incident into a node represent either the control condition on which the execution of the operations depends or the data value on which the node's operation depend. Therefore, PDG represents two types of dependence, namely, control dependence and data dependence.

2.1.1. Control Dependence

Control dependency is a situation in which a program instruction executes if the previous instruction evaluates in a way that allows its execution. As in Figure 2.1, the execution of $a = 20$ depend on the evaluation of *condition*. We then assert that $a = 20$ is control

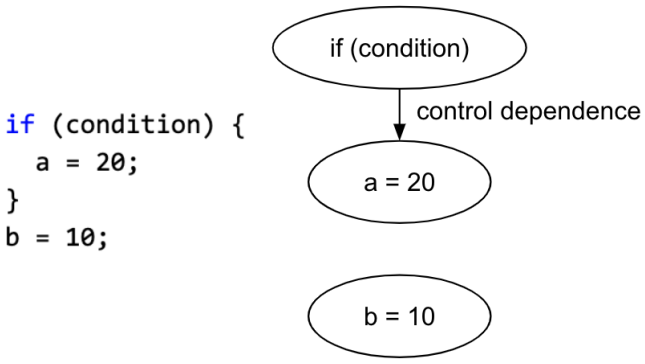

Figure 2.1. Control Dependence

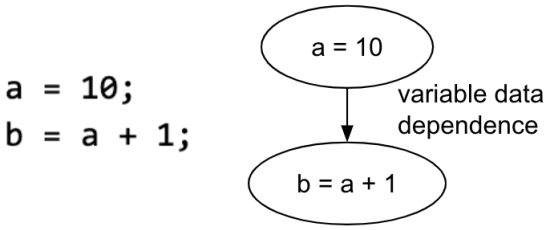

Figure 2.2. Variable Data Dependence

dependent on $if(condition)$. However, no matter what *condition* is evaluated to, $b = 10$ always gets executed, so there's no control dependence between $if(condition)$ and $b = 10$.

2.1.2. Data Dependence

Data dependency is a situation in which a program instruction refers to the data of a preceding instruction. There are two types of data dependence. *Variable Data Dependence* and *Memory Data Dependence*. Variable data dependence refers to data dependent on each other through variable. Memory data dependence, which refers to the data in a memory cell will be accessed in the following program.

In Figure 2.2, $b = a + 1$ variable data dependent on the variable of a whose value is 10.

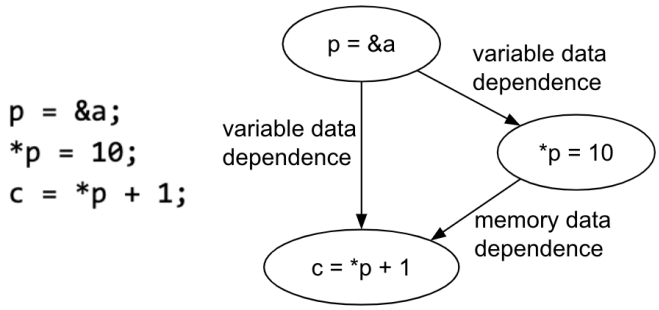

Figure 2.3. Memory Data Dependence

There are three data dependence in Figure 2.3, two variable data dependence and one memory data dependence. $c = *p + 2$ memory data dependent on $*p = 10$ since the content in the memory location pointed by p gets updated.

Data dependence provides an explicit representation of the definition-and-use relationships implicitly present in the source program.

2.2. Usage of Program Dependence Graph

PDG unifies both control and data dependence, so it can be used by code transformation techniques that requires control dependence info or data dependence info or both.

Program Parallelization, which splits the source program to be running on multiple cores instead of a single core, gains more and more attention these days. Parallelization requires both control and data dependence information and parallelizes the source program in a way that all the dependence are preserved. PDG helps the parallelization process to be performed in a way that respect the dependence of the source program, thus achieving the goal of performance efficiency.

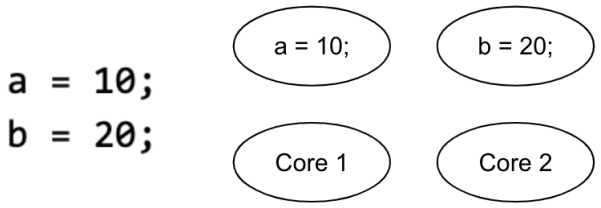

Figure 2.4. Example of Program Parallelization

As shown in Figure 2.4, since there's no dependence between $a = 10$ and $b = 20$, these two instructions can then be parallelized. If constant number of cycles is assumed to run every instruction, we can then achieve a 50% increase in performance after parallelization.

2.3. Correctness v.s. Precision

The *Correctness* and *Precision* of PDG needs to be differentiated. We consider a PDG constructed to be correct if it has all the actual dependence. The precision of PDG means given all the dependence identified within a PDG, what is the proportion of the true dependence. The true dependence should at least be a subset of the dependence identified. Otherwise, the PDG constructed wouldn't be considered as correct.

It would be easier to conservatively assume all the dependence exists. As shown in Figure 2.5, in which the true dependence is represented by solid line while the false positive dependence is represented by dotted line. Given this PDG constructed, we consider it to be correct but only has a precision of 60%.

2.4. Problem

The PDG generated introduces many false positive dependence due to the trade-off of between precision, latency and memory consumption within the code analysis process.

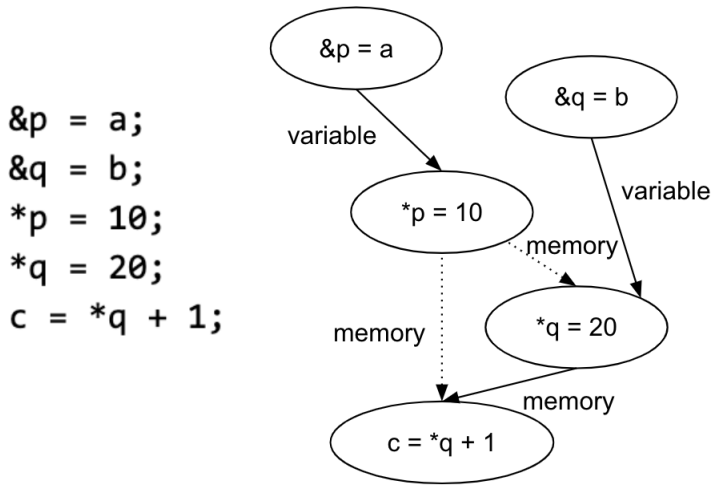

Figure 2.5. Correctness v.s. Precision

Though the PDG constructed is correct, the less of precision prohibits further code transformation which originally is able to be applied. In the example of Figure 2.5. The false positive memory data dependence introduced between $c = *q + 1$ and $*p = 10$ prevents the originally applicable constant propagation of $*q$, who can be substituted with constant 20.

Identifying control and variable data dependence is relatively simple, as efficient and scalable algorithm to find control dependence is available and variable data dependence is explicit in the program. However, identifying memory data dependence is complicated since memory data is generally hidden and hard to track. Therefore, there's a large portion of false positive memory data dependence exists in the constructed PDG. The code analysis technique to identify memory data dependence called *Pointer Analysis*. In the next chapter, we discuss more about pointer analysis and understand why its complexity. Then, in the chapter after next chapter, we show how by leveraging the *Static Value-Flow*

Analysis, these false positive memory data dependence can be efficiently and successfully removed.

CHAPTER 3

Pointer Analysis

This chapter describes *Pointer Analysis*, also known as alias analysis, points-to analysis, is an static code analysis technique that reasons the behavior of pointers at compile time. We first give an example to answer the question what pointer analysis is trying to solve. Then several variants of pointer analysis is introduced. LLVM pointer analysis is explained in the last and we show how the false positive memory data dependence is introduced due to LLVM's less precise pointer analysis.

3.1. An Example

Listing 3.1. An Example Program

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4     int foo(int *p1, int *p2, int value) {
5         *p1 = value;
6         return *p2;
7     }
8
9     int bar(int value1, int value2) {
```

```
10          return value1 + value2;
11      }
12
13      int main() {
14          int *p = (int *)malloc(sizeof(int));  // integer object 1
15          int *q = (int *)malloc(sizeof(int));  // integer object 2
16
17          int value1 = foo(p, q, 10);
18          int value2 = bar(1, 2);
19          int value3 = foo(q, p, 20);
20
21          return 0;
22      }
```

A pointer analysis statically reasons the behavior of pointers at compile so as to help making code transformation of source program involves pointer operations. It answers at least the following questions:

- In a given point of the program, can two pointers point to the same memory location (alias with each other)?
- What memory locations does a pointer points to at a given point of the program?

At the end of pointer analysis, a points-to set will be given, which maps a pointer to the memory locations (objects) it points to. For instance, in this example, the points to

set for pointer p in function *main* is $\{obj1\}$ due to the *malloc* at the beginning of the function. Pointer p has $obj2$ in its points-to set likewise.

For this program, we more interested at function foo, does $p1$ and $p2$ alias with each other? And there are generally three answers to this question:

- *NoAlias*. Pointer $p1$ and $p2$ cannot alias with each other.
- *MustAlias*. Pointer $p1$ and $p2$ must point to the same memory location (object). A constant propagation to substitute $*p2$ with value can be applied.
- *MayAlias*. Pointer $p1$ and $p2$ may or may not alias with each other because we don't know.

Since a *MayAlias* introduces uncertainty, we have to conservatively assume there exists a memory data dependence between $*p1 = value$ and $*p2$. This is how the false positive memory data dependence is introduced. Though after inspecting this simple example. We can know for sure that there should be no memory data dependence between $*p1 = value$ and $*p2$ since $p1$ and $p2$ can never alias with each other.

3.2. Variants of Pointer Analysis

Pointer analysis has to be correct. This can simply be achieved by assuming all pointers may alias with each other and pointer points to all the memory locations. However, this makes a pointer analysis to be less useful since the precision will be very low. The approach to have a highly precise pointer analysis is hard and complex, all points-to information will be propagated along with the control-flow graph and are likely to be updated as pointer operations get involved, as described by Hind [**2**]. Since it's relative unable to

scale to get a highly precise pointer analysis. Typically, pointer analysis is designed based on various needs and has a lot of variants.

3.2.1. Intraprocedural v.s. Interprocedural

An intraprocedural pointer analysis performs on a function level whereas an interprocedural pointer analysis is performed on the whole program (module) level.

An intraprocedural pointer analysis will return *MayAlias* between $p1$ and $p2$ as they are function parameters, pointer analysis has to conservatively assume they can point to any memory location when this invoked. On the other hand, a interprocedural pointer analysis will propagate points-to information from the callsite. Pointer analysis can then decide with they alias comparing their points-to sets.

Though interprocedural pointer analysis provides a higher degree of precision, it is expensive in time and memory usage. An interprocedural pointer analysis requires the computation of call graph and a lot of points-to information will get propagate along the call graph edges.

3.2.2. Flow-insensitive v.s. Flow-sensitive

A flow-insensitive pointer analysis generates one copy of points-to set for the whole program since the execution order of instructions don't matter. Flow-sensitive pointer analysis generates the points-to set for every point of the program as the execution order of instructions are taken into account.

In Figure 3.1, a flow-sensitive pointer analysis will returns that after the execution of instruction j, the points-to set for p is $\{a\}$, for q is $\{b\}$. However, since the order of

```
i: p = &a;
j: q = &b;
k: p = &c;
```

Figure 3.1. Flow-Insensitive and Flow-Sensitive

execution doesn't matter and only one copy of points-to set is generated for flow-insensitive pointer analysis. The points-to set at any point for p is $\{a, c\}$, for q is $\{b\}$.

Flow-sensitive pointer analysis has notoriously been hard to scale. Though flow-sensitive pointer analysis provides a higher degree of precision, imagine a program with thousands of pointers and thousands of memory locations, every instruction in the program have to maintain at least two copies of thousands of points-to information, before and after the instruction is executed. Given these shortcomings, a lot of pointer analysis is performed in a flow-insensitive manner.

3.2.3. Context-insensitive v.s. Context-sensitive

Given an interprocedural pointer analysis with context-insensitivity, several invocation of the same function get merged and all the points-to information will be unified and propagate along the call graph edge. A context-sensitive pointer analysis stores the context of points-to information at a callsite and treat calls to the same function respectively.

As in the example of function foo, a context-insensitive pointer analysis will determine that pointers $p1$ and $p2$ can all points to both $obj1$ and $obj2$, while a context-sensitive pointer analysis treats every callsite respectively and decide the first invocation of function foo has $p1$ points to $obj1$ and $p2$ points to $obj2$. The second invocation has $p1$ points to $obj2$ and $p2$ points to $obj1$.

Implementing a context-sensitive pointer analysis can be hard since the context information has to be preserved, which becomes challenging when call graph is huge and recursion occurs.

3.2.4. Field-insensitive v.s. Field-sensitive

Field insensitive and field-sensitive pointer analysis treats the field of a compounded struct or class object differently. A field-insensitive pointer analysis treats all the fields in a struct or class object as a whole, thus pointers point to different field of the struct or class object will be considered as alias. On the other hand, a field-sensitive pointer analysis returns no alias if two pointers point to different field of a struct or class object.

3.2.5. LLVM Pointer Analysis

Our current construction of program dependence graph uses *LLVM Pointer Analysis Infrastructure* to determine memory data dependence. Most of LLVM's pointer analysis is intraprocedural, flow-insensitive, context-insensitive and field-insensitive. This generates a lot of false positive memory data dependence. As in the first example of this chapter. LLVM pointer analysis return *MayAlias* result between pointer $p1$ and $p2$ in function foo, which the true dependence should be *NoAlias*.

In the next chapter, we present *Static Value-Flow Analysis*, a research tool leverage sparse pointer analysis to make scalable interprocedural flow-senstive, context-sensitive and field-senstive pointer analysis.

CHAPTER 4

Static Value-Flow Analysis

Static Value-Flow Analysis (SVF) is a research tool leverage the recent advance in sparse pointer analysis and supports refinement-based interprocedural program dependnece analysis, which is designed ant implemented by a research group at University of Technology Sydney lead by Sui [**1**]. In this chapter, We first give an overview of the workflow of SVF. Then we dive deep into its two major modules, *Andersen's Pointer Analysis* and *Value-Flow Construction*. Finally, we demonstrate how the false positive memory data dependence can be successfully removed after applying SVF with a context-sensitive pointer analysis in a scalable way.

4.1. Static Value-Flow Analysis Workflow

SVF work flow is scheduled as follows. The source program is first compiled and transformed into LLVM intermediate representation (bitcode), in Figure 4.2. An interprocedural, flow-insensitive and context-insensitive pointer analysis is performed on the bitcode to generate a less precise points-to result. This points-to result is used by the next module, value-flow construction. The value-flow construction makes both the register variables and memory objects explicit in the LLVM intermediate representation. A value-flow graph can be generated. Further flow-sensitive and context-sensitive pointer analysis can then be performed on this value-flow graph in a scalable way thanks to a more precise points-to result.

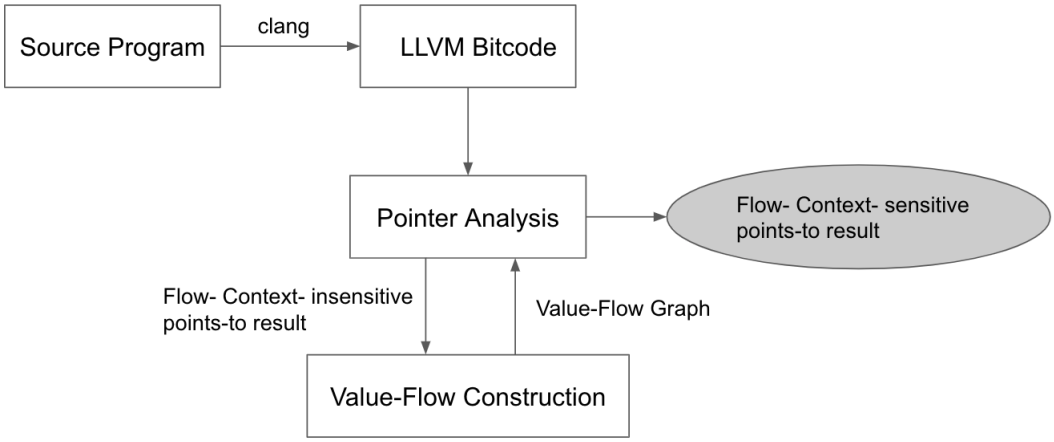

Figure 4.1. Workflow of Static Value-Flow

```
define dso_local i32 @_Z3fooPiS_i(i32*, i32*, i32) #0 {
  store i32 %2, i32* %0, align 4
  %4 = load i32, i32* %1, align 4
  ret i32 %4
}

define dso_local i32 @_Z3barii(i32, i32) #0 {
  %3 = add nsw i32 %0, %1
  ret i32 %3
}

define dso_local i32 @main(i32, i8**) #1 {
  %3 = call noalias i8* @malloc(i64 4) #3
  %4 = bitcast i8* %3 to i32*
  %5 = call noalias i8* @malloc(i64 4) #3
  %6 = bitcast i8* %5 to i32*
  %7 = call i32 @_Z3fooPiS_i(i32* %4, i32* %6, i32 10)
  %8 = call i32 @_Z3barii(i32 1, i32 2)
  %9 = call i32 @_Z3fooPiS_i(i32* %6, i32* %4, i32 20)
  ret i32 0
}
```

Figure 4.2. LLVM Intermediate Representation (Bitcode)

4.2. Andersen's Pointer Analysis

Andersen's Pointer Analysis is a well-known flow-insensitive pointer analysis that can easily be extended to support interprocedrual and field-sensitive. Andersens's pointer

```
p = &a;     // l_a ∈ pts(p)
q = &b;     // l_b ∈ pts(q)
r = *p;     // ∀ o ∈ pts(p), pts(r) ⊇ pts(o)
p = q;      // pts(p) ⊇ pts(q)


Points-to Set
pts(p) = {a, b}
pts(q) = {b}
pts(r) = {a, b}
```

Figure 4.3. An example of Andersen's Pointer Analysis and Points-to Set

analysis transforms all the pointer operations in the source program into a set of constraints use the rules as follows:

- $p = \&x \Longrightarrow l_x \in pts(p)$
- $p = q \Longrightarrow pts(p) \supseteq pts(q)$
- $*p = q \Longrightarrow \forall o \in pts(p), pts(o) \supseteq pts(q)$
- $p = *q \Longrightarrow \forall o \in pts(q), pts(p) \supseteq pts(o)$

A constraints solver will then be invoked to solve all of these constraints and generates the global points-to set. The solver converges when all the constraints are met.

4.3. Value-Flow Construction

With the flow-insensitive global points-to set available, the value-flow construction module will be invoked. Value-flow construction model consists of three steps, *Mod-Ref Analysis*, *Memory SSA Construction* and *Value-Flow Graph Construction*. These three steps will be further explained in the following sections.

4.3.1. Mod-Ref Analysis

Mod-Ref Analysis captures both the modification and reference side-effect for memory locations of a function. We say a function has a modification side-effect on a memory location if this memory location is modified either rewritten by a store instruction or escape to be written through a function call inside this function. Reference, on the other hand, refers that a memory location is accessed either read by a load instruction of escape to be read through a function call.

As in function foo and bar in the example in Chapter 2, we conclude that foo has modification side effect on the memory locations pointed by $p1$ and reference side effect on the memory locations pointed by $p2$. Function bar doesn't have any side effect as no memory store or write operations get involved.

4.3.2. Memory SSA

Given the global points-to set and the mod-ref analysis result. A memory SSA can then be constructed, as shown in Figure 4.4. The idea to construct such a memory SSA is by making all the data dependence including both variable and memory dependence explicit in the LLVM IR. The memory SSA can be constructed through annotation on the LLVM IR with respect to the following rules.

- Load: $p = *q$ is annotated with $LOADMU(o)$ for any object pointed by q to represent a use of object o.
- Store: $*p = q$ is annotated with $o2 = STORECHI(o1)$ for any object pointed by p to represent both a definition and use of object o.

```
define dso_local i32 @_Z3fooPiS_i(i32*, i32*, i32) #0 {
  (obj1_v1, obj2_v1) = ENTRYCHI(obj1_v0, obj2_v0)
  store i32 %2, i32* %0, align 4
  (obj1_v2, obj2_v2) = STORECHI(obj1_v1, obj2_v1)
  LOADMU(obj1_v2, obj2_v2)
  %4 = load i32, i32* %1, align 4
  ret i32 %4
  RETMU(obj1_v2, obj2_v2)
}

define dso_local i32 @_Z3barii(i32, i32) #0 {
  %3 = add nsw i32 %0, %1
  ret i32 %3
}

define dso_local i32 @main(i32, i8**) #1 {
  %3 = call noalias i8* @malloc(i64 4) #3    // obj1
  obj1_v1 = CALLCHI(obj1_v0)
  %4 = bitcast i8* %3 to i32*
  %5 = call noalias i8* @malloc(i64 4) #3    // obj2
  obj2_v1 = CALLCHI(obj2_v0)
  %6 = bitcast i8* %5 to i32*
  CALLMU(obj1_v1)
  CALLMU(obj2_v1)
  %7 = call i32 @_Z3fooPiS_i(i32* %4, i32* %6, i32 10)
  obj1_v2 = CALLCHI(obj1_v1)
  obj2_v2 = CALLCHI(obj2_v1)
  %8 = call i32 @_Z3barii(i32 1, i32 2)
  CALLMU(obj2_v2)
  CALLMU(obj1_v2)
  %9 = call i32 @_Z3fooPiS_i(i32* %6, i32* %4, i32 20)
  obj2_v3 = CALLCHI(obj2_v2)
  obj1_v3 = CALLCHI(obj1_v2)
  ret i32 0
}
```

Figure 4.4. Memory SSA after Annotations

- CallSite: callsites are annotated with $CALLMU$ and $CALLCHI$ based on the mod-ref results of the callee function.
- Function entry/exit: $ENTRYCHI$ and $RETMU$ are annotated at the beginning/exit of function to represent the definition/future use of memory objects.

4.3.3. Value-Flow Graph

With memory SSA in hand, a value-flow graph can be constructed by connecting both the def-use of register variable and memory objects. Since the register variables' def-use relationship is already explicit in the LLVM bitcode SSA form, simply connecting all the def and use of memory objects within memory SSA can get a value-flow graph consisting of both register variables and memory objects.

4.4. Leverage Static Value-Flow Analysis

With the value-flow graph available, any flow-sensitive or context-sensitive pointer analysis now can then be performed using traditional iterative algorithm on this graph sparsely in a scalable manner.

For example, from memory SSA in Figure 4.4, we notice that the invocation to function foo will have no side-effect on any memory locations. Therefore the points-to set before and after the execution of this callsite remains the same. In fact, when performing a flow-sensitive pointer analysis, the points-to information only needs to be propagated along the value-flow edge of memory objects as its where the pointer operations takes place and update to the points-to set happens.

Now, when we query the alias result between pointer $p1$ and $p2$ in function foo using any context-sensitive pointer analysis. We will get the *NoAlias* result. In other words, the false positive memory data dependence introduced before due to *MayAlias* result can now be successfully removed.

CHAPTER 5

Leverage Static Value-Flow Analysis

In this chapter, we describe how to leverage the result of SVF into a parallelizing framework – *NOELLE*. We introduce the workflow of *NOELLE* and the integration process with SVF.

5.1. NOELLE

NOELLE is a parallelizing compilation framework aiming at irregular workloads designed at implemented by Simone Campanoni and Angelo Matni at Northwestern University. NOELLE focuses on parallelizing loops in the source program as loops spend the most proportion of time during a program's execution. Several loop parallelizing techniques are used within NOELLE, including *DOALL*, *DSWP* [**3**] and *HELIX* [**4**], etc. NOELLE also has regression, unit, and performance test to measure the correctness and performance improved for the source program after being parallelized.

5.1.1. Workflow of NOELLE

The workflow of NOELLE is shown in Figure 5.1. A source program is first compiled into LLVM bitcode by clang. Next, transformation such as function inlining and loop hoisting, loop distribution and loop unrolling are performed on the bitcode to explore more parallelizing opportunities. Program profiling will then be used to collect information during the execution of the program to determine the hot loops. After that, the process

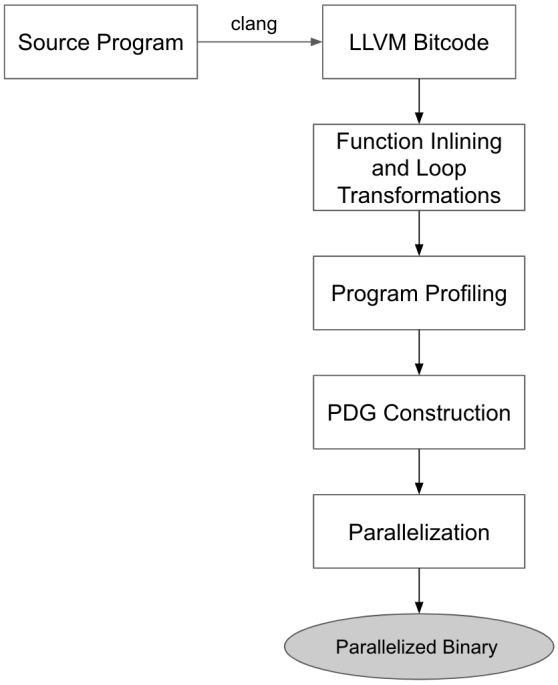

Figure 5.1. Workflow of Noelle

to construct program dependence graph gets invoked to identify both the control and data dependence within the program. Finally, several parallelizing techniques will be applied on the source program based on the program dependence graph to generate the parallelized binary targeting hot loops.

5.2. Integration with Static Value-Flow Analysis

The integration of SVF with NOELLE has three steps. Building SVF as a shared library, initializing SVF client for querying pointer analysis and the actual alias querying.

5.2.1. Build SVF

We use SVF-1.8 based on LLVM 9.0.0 is for the integration process. By default, SVF is built as a static library, which limits the integration with NOELLE since adds-on libraries are normally shared libraries. This limit has been removed by modifying SVF's default build script and CMakeLists. A compiler flag *-fPIC* should also be specified to generate a place independent code of SVF.

5.2.2. Initialize WPAPass

SVF provides an pass called *WPAPass* to perform a whole program analysis. The program dependence graph gets constructed resides in *PDGAnalysis* pass of NOELLE, the *WPAPass* then is declared as a dependent of *PDGAnalysis*.

SVF has been designed to perform interprocedural and field-sensitive pointer analysis. Therefore, in order to reduce the overall complexity, SVF modifies the bitcode in the following ways. For interprocedural analysis, SVF unifies all the exit nodes and ensures every function can only have one exit to simplify the propagation of points-to set. For field-sensitivity, SVF breaks GEP constant expressions into GEP instructions inside LLVM bitcode, making the operations to field objects explicit.

To ensure the bitcode is not modified within the workflow of NOELLE, a normalization process is performed at the beginning of NOELLE. Therefore, two extra transformation passes, *-mergereturns* and *-break-constgeps* are added to ensure the LLVM bitcode is not modified through SVF.

5.2.3. Invoke alias()

Once the *WPAPass* finishes, we are able to query alias info between pointers through SVF. *WPAPass* provides *alias()* APIs for clients to query this information.

Multiple pointer analysis can be asked to perform on the bitcode inside SVF. NOELLE initially uses LLVM's pointer analysis, they are -globals-aa, -cfl-steens-aa, -tbaa, -scev-aa and -cfl-anders-aa. The alias result is determined by the least conservative one. That is, if one pointer analysis returns a Must result, it will then be accepted. SVF provides eight extra pointer analysis: -nander, -hander, -sander, -sfrander, -wander, -ander, -lander and -hlander. They are all interprocedural, flow-insenstive, context-insensitive, field-sensitive Andersen's pointer analysis solving by different solvers. The less conservative result between LLVM's pointer analysis and SVF's pointer analysis will be adopted.

CHAPTER 6

Combine Program Dependence Graph with Bitcode

In this chapter, we show how to couple the constructed PDG into bitcode to make pdg expicit. Firstly, *PDG Embedding* and *PDG Loading* is presented, we describe the reason why PDG Embedding and PDG Loading can save us tremendous time. A PDG can further be dumped into an *JSON* object, which supports PDG comparison and other uses. The scheme of this PDG JSON object is also given in the chapter.

6.1. Program Dependence Graph Embedding and Loading

Besides providing an intermediate representation of the source program, LLVM bitcode also let users to store various information. This is realized through adding *Metadata* to the bitcode. Metadata can be attached to the bitcode in both module, function and instruction levels. The benefits of embedding PDG in the bitcode is that it not only allows a single construction of PDG through analysis, but also gives you a bitcode with PDG that can be restored at any time. As analysis to determine dependence can be expensive, for a system that requires to construct PDG multiple times without modifying the bitcode, embedding PDG in the bitcode can save a tremendous of time and memory usage.

```
define dso_local i32 @main(i32, i8**) #0 {
  %3 = call noalias i8* @malloc(i64 1) #3
  %4 = call i64 @atoll(i8* %3) #4
  %5 = trunc i64 %4 to i32
  ret i32 0
}
```

Figure 6.1. LLVM Bitcode before PDG Embedding

6.1.1. PDG Embedding

The embedding of PDG first needs a PDG constructed through analysis. At the beginning, all the nodes are added to the PDG. These nodes are either function arguments or instructions. All the nodes are assigned a globally unique ID. Edges in PDG represent dependence, and they are added between nodes through analysis for control dependence, variable data dependence and memory data dependence.

The PDG nodes are first embedded to the bitcode, an instruction node is embedded to the corresponding instruction with its ID. Since LLVM doesn't support adding metadata for function arguments, argument nodes are therefore embedded to the function with IDs for all its arguments.

The PDG edges are next to be embedded to the bitcode on the function level. We store the following information for every edge. The source node and destination node, which are represented by IDs, the dependence information indicates whether this is a control, variable data or memory data. And the information to determine whether the dependence is loop-carried and whether it can be removed or not.

With all the PDG nodes and edges embedded to the bitcode, a named metadata is embedded to the module indicating whether this bitcode has PDG embedded. As shown in Figure 6.2.

```
define dso_local i32 @main(i32, i8**) #0 !noelle.pdg.args.id !3 !noelle.pdg.edges !6 {
  %3 = call noalias i8* @malloc(i64 1) #3, !noelle.pdg.inst.id !8
  %4 = call i64 @atoll(i8* %3) #4, !noelle.pdg.inst.id !9
  %5 = trunc i64 %4 to i32, !noelle.pdg.inst.id !16
  ret i32 0, !noelle.pdg.inst.id !26
}

!noelle.module.pdg = !{!2}
!2 = !{!"true"}
!3 = !{!4, !5}
!4 = !{i64 0}
!5 = !{i64 1}
!6 = !{!7, !13, !15, !17, !19, !13, !20, !21, !22, !23, !24, !25}
!7 = !{!8, !9, !10, !2, !11, !10, !10, !10, !12}
!8 = !{i64 2}
!9 = !{i64 3}
!10 = !{!"false"}
!11 = !{!"RAW"}
!12 = !{}
!13 = !{!8, !8, !2, !10, !14, !10, !10, !10, !12}
!14 = !{!"WAW"}
!15 = !{!9, !16, !10, !2, !11, !10, !10, !10, !12}
!16 = !{i64 4}
!17 = !{!8, !8, !2, !10, !18, !10, !10, !10, !12}
!18 = !{!"WAR"}
!19 = !{!8, !8, !2, !10, !11, !10, !10, !10, !12}
!20 = !{!8, !9, !2, !10, !18, !10, !10, !10, !12}
!21 = !{!8, !9, !2, !10, !14, !10, !10, !10, !12}
!22 = !{!9, !8, !2, !10, !11, !10, !10, !10, !12}
!23 = !{!9, !8, !2, !10, !14, !10, !10, !10, !12}
!24 = !{!9, !8, !2, !10, !18, !10, !10, !10, !12}
!25 = !{!8, !9, !2, !10, !11, !10, !10, !10, !12}
!26 = !{i64 5}
```

Figure 6.2. LLVM Bitcode after PDG Embedding

6.1.2. PDG Loading

The loading of PDG is a reverse process to construct the PDG from the metadata without querying any analysis. If the named metadata indicating the PDG has been embedded can be found, we then construct PDG from metadata instead of from analysis.

The PDG nodes are first constructed by going through all the function arguments and instructions in the bitcode. An ID to node map will also be constructed using the ID number in the metadata.

Then, by going through every edge set embed to every function, we reconstruct PDG edges by going through every edge metadata inside the edge set of that function. The edge attribute can be restored through reading the metadata and the source and destination node can be identified by querying the ID to node map.

6.2. Dump Program Dependence Graph to JSON

Besides embedding and loading PDG into the bitcode, we also provide mechanism to dump the PDG to an JSON object. So the difference between two PDGs can easily be detected by comparing two JSON objects. Moreover, an JSON object can store extra and more structured information and therefore can be leveraged by many other applications.

6.2.1. Schema

The JSON object has two keywords, *nodes* and *edges*. *nodes* maps to a nested JSON object whose keywords are the function names. Each function name maps to an array object storing the nodes data of that function. *edges*, likewise, maps to a nested JSON object whose keywords are also the function names, which then map to an array object storing the edges data. An example is given in Figure 6.3.

```
{
  "hasPDG": true,
  "nodes": {
    ...
    "foo": [
      {
        "type": "instruction",
        "id": 10,
        "value": "store i32 %2, i32* %0, align 4"
      },
      {
        "type": "instruction",
        "id": 11,
        "value": "%4 = load i32, i32* %1, align 4"
      }
    ],
    ...
  },
  "edges": {
    ...
    "foo": [
      {
        "source": 10,
        "destination": 11,
        "isControlDependence": false,
        "isMustDependence": false,
        "isMemoryDependence": true,
        ...
      }
    ],
    ...
  }
}
```

Figure 6.3. PDG JSON Object

CHAPTER 7

Evaluation

In this chapter, we do the evaluation on test suite *SPEC2017*. We first measure the false positive memory data dependence removed after integrating SVF and then measure the time elapsed for *PDG Embedding* and *PDG Loading*.

7.1. Measure memory data dependence

SPEC2017 is a CPU intensive suite to provide a comparative measure of compute-intensive performance across the widest practical range of hardware using workloads developed from real user applications.

We integrated SVF into NOELLE with interprocedural, flow-insensitive, context-insensitive and context-sensitive pointer analysis to generate the intermediate PDG and compare the memory dependence generated before as baseline and after SVF gets integrated as shown in Table 7.1. Several benchmarks are not included in the table because SVF run for too long and didn't finish within 2 hours. These benchmarks are *imagick*, *omnetpp*, *parest* and *perlbench*.

As shown in the table, an average of 25% false positive memory data dependence get successfully removed.

BENCHMARK	baseline	svf_integrated	% of memory dependence removed
deepsjeng	300,088	213,082	29.0%
lbm	17,164	12,764	25.6%
leela	1,909,848	1,730,526	9.4%
mcf	640,182	610,598	4.6%
nab	6,345,124	5,929,618	6.5%
namd	15,709,764	4,845,886	69.2%
x264	14,565,990	12,961,420	11.0%
xalancbmk	26,234,504	16,169,354	38.4%
xz	1,072,660	681,268	36.5%
AVERAGE			25.1%

Table 7.1. Statistics of memory data dependence between baseline and integration of SVF

7.2. Measure PDG Embedding and PDG Loading

We then measure the time elapsed for *PDG Embedding* and *PDG Loading* for benchmarks within *SPEC2017* after integration with SVF. The number of nodes, edges and the time to perform embedding and loading in milliseconds are presented in the Table 7.2.

BENCHMARK	# of nodes	# of edges	PDG embedding	PDG loading
deepsjeng	20,030	258,703	480 ms	1,555 ms
lbm	2,858	20,067	31 ms	108 ms
leela	37,446	1,825,209	3,369 ms	12,422 ms
mcf	8,272	632,530	1,091 ms	4,236 ms
nab	45,250	6,038,519	15,195 ms	44,203 ms
namd	202,683	5,426,533	10,755 ms	35,058 ms
x264	149,345	13,317,229	26,956 ms	93,342 ms
xalancbmk	952,067	18,411,447	40,150 ms	117,817 ms
xz	37,080	774,979	1,425 ms	4,925 ms

Table 7.2. Statistics of elapsed time in milliseconds for PDG Embedding and PDG Loading

As we can see, for the largest benchmark, *xalancbmk*, who has over $950K$ nodes and $18M$ dependence edges. A construction of PDG through analysis for such a large benchmark can last for hours. However, by performing *PDG Embedding* and *PDG Loading*, this time-consuming process and its generated output can be preserved and restored in less than 1 and 2 minutes respectively.

CHAPTER 8

Conclusion

In this thesis, we present *Static Value-Flow Analysis*, a code analysis technique leverages the recent advance in sparse pointer analysis so that the flow-sensitive or context-sensitive pointer analysis that previously cannot be easily scaled now can be performed in a scalable manner. The result shows that by leveraging SVF with interprocedural, flow-insensitive, context-insensitive and field-sensitive pointer analysis, an average of 25% false positive memory data dependence can be dropped. We also introduce the implementation of *PDG Embedding* and *PDG Loading*, the idea to embed and restore the PDG information to and from LLVM bitcode through metadata. By performing *PDG Embedding* and *PDG Loading*, the construction of PDG through analysis needs to be performed only once and the following process who requires PDG construction can benefit from it. As shown in the evaluation result. Even for a large code base with over $950K$ PDG nodes and $18B$ PDG edges, the construction of PDG can be completed within 2 minutes.

8.1. Future Work

Although SVF already provides a huge improvement in removing false positive memory data dependence, we believe there still has plenty of room to explore in the future.

Firstly, implementing a flow-sensitive, context-sensitive pointer analysis on the value-flow graph generated to further removing those false positive memory dependence.

Secondly, querying through getModRefInfo APIs to get a better understanding of the mod-ref relation on a specific memory location through SVF. Our three pull requests have already been merged by the author into master branch of SVF to query the mod-ref information. These commits provide a solid base for us to use getModRefInfo APIs in the future.

- delay inclusion of mod to ref to preserve useful mod-ref analysis result (5a1cccd).
- add getModRefInfo APIs support for SVF (525fde5).
- Refine getModRefInfo APIs to accomodate with field-sensitivity of SVF (95eb264).

Finally, using sub edge mechanism in NOELLE to explore more parallelism opportunities. These sub edges provide detailed dependence information. Instead of adding a memory data dependence edge between two function calls who have memory data dependence in between, a sub edge connected from two instructions inside the callee functions where the memory data dependence actually happens allows more parallelization opportunities for instructions who should not be blocked by the memory data dependence.

References

[1] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[2] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '01, page 54–61, New York, NY, USA, 2001. Association for Computing Machinery.

[3] Guilherme Ottoni, Ram Rangan, A. Stoler, and D.I. August. Automatic thread extraction with decoupled software pipelining. pages 12 pp.–, 12 2005.

[4] Simone Campanoni, Kevin Brownell, Svilen Kanev, Timothy M. Jones, Gu-Yeon Wei, and David Brooks. Helix-rc: An architecture-compiler co-design for automatic parallelization of irregular programs. *SIGARCH Comput. Archit. News*, 42(3):217–228, June 2014.

[5] Yulei Sui and Jingling Xue. Svf: interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th international conference on compiler construction*, pages 265–266. ACM, 2016.

[6] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* `http://llvm.cs.uiuc.edu`.