A SYMBOLIC DATA DEPENDENCE ANALYSIS WITH ABSTRACT INTERPRETATION

A Thesis in
Computer Science and Engineering
by
Qingyuan Zheng

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

May 2021
The thesis of Qingyuan Zheng was reviewed and approved by the following:

Gang Tan  
Professor of Computer Science and Engineering  
Thesis Advisor

Danfeng Zhang  
Assistant Professor of Computer Science and Engineering

Chitaranjan Das  
Department Head and Distinguished Professor  
Department Head of Computer Science and Engineering
Abstract

Data dependence analysis is a part of program dependence analysis. It inspects data flows in a program and observes where the value of a particular name in the program is defined. While data dependence analysis in the wild captures a good amount of information for us to reason about a program, they also produce many false positive results due to the imprecise alias analysis. This research introduces a modular data dependence analysis built on a modular symbolic points-to analysis based on Abstract Interpretation and Theorem Prover, which is precise and maintains practical performance. We present the analysis in the context of LLVM intermediate representation programs and provide the evaluation on precision, time and memory consumption.
Contents

List of Figures vi

List of Tables vii

Acknowledgement viii

Chapter 1. Introduction 1

Chapter 2. Background 3

2.1 Control Flow Graph . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
2.2 Alias Analysis . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
2.3 Abstract Interpretation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4

Chapter 3. A Precise and Modular Points-to Analysis 5

3.1 Memory Abstraction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
  3.1.1 Abstract Memory Location . . . . . . . . . . . . . . . . . . . . . . . . . 6
  3.1.2 Points-to Constraints . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
3.2 Analysis Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
3.3 Initialization of Abstract Store . . . . . . . . . . . . . . . . . . . . . . . . . . 9
3.4 Example . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10

Chapter 4. Points-to Analysis in LLVM 12

4.1 Syntax And Terms . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
4.2 Allocation Instruction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
4.3 Store Instruction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
List of Figures

3.1 Analysis Overview ............................................... 8
3.2 Initial Abstract Store ........................................... 10
3.3 Initial Abstract Store of swap Function ....................... 11
3.4 Summary Abstract Store of swap Function ..................... 11

4.1 Abstract Semantics of Allocation Instructions .................. 14
4.2 Abstract Semantics of Store Instructions ....................... 15
4.3 Abstract Semantic of Load Instructions ........................ 17
4.4 Abstract Semantic of Phi/Select Instructions With Two Operands 18
4.5 Abstract Semantic of Pointer Cast Instructions ................. 19
4.6 Summary Abstract Store of swap Function ...................... 22
4.7 Initial Abstract Store of do_swap Function .................... 23

6.1 Analysis run time distribution of functions in Nginx .......... 35
List of Tables

6.1 Test Programs .......................................................... 34
6.2 Analysis Precision Comparison ........................................... 34
6.3 Analysis Run-time Comparison ........................................... 35
6.4 Analysis Memory Consumption .......................................... 35
6.5 Case Study ............................................................... 36
Acknowledgement

Special thanks to professor Gang Tan for his invaluable guidance and tremendous help in the development of this research. Also, I would like to acknowledge the contributions provided by Yongzhe Huang and Shen Wang.
Chapter 1

Introduction

Program dependence graph (PDG) is a program representation that explicitly marks data and control dependencies. It provides valuable information in formal static analysis, one of the essential techniques to reason about a program ahead of run-time that is broadly applied in the area of compilers and computer security[LTJ17]. While modern tool-chains such as LLVM have made capturing def-use dependencies for variables and intra-procedural control dependencies trivial thanks to the static single assignment (SSA) form and control flow graph (CFG), data dependencies behind pointers are much trickier to identify due to the dynamism of program execution.

For an arbitrary pair of store and load instructions in a particular execution of a program, a read-after-write (RAW) relation exists if and only if 1) the load is executed after the store, 2) the same pointer is used, 3) no other store overwrites data under the pointer between two instructions. If there could be such a relation in any possible execution, we say there’s a RAW data dependence between the two instructions. Because addresses of memory allocations are generally unknown until run-time, identification of such relations requires an alias analysis. Precision of alias analysis has a huge impact on data dependence analysis.

```c
int seq(int* p)
{
    *p = 41; *p = 42; return *p;
}
```
As conventional algorithms usually trade flow-sensitivity and strong-update off for performance and soundness, data dependence analysis built upon them is less precise. Application of them in subsequent analysis could lead to insufficient information and conservative decision. For example, an analysis disregarding strong-update on the function above would yield two dependence edges. Pointer dereference from the return statement would depend on the assignment of both 41 and 42, which is undesirable because obviously the first assignment will be overwritten by the second one and have no influence on what value to return.

In this research, we port a precise modular points-to analysis[DDAS11] to LLVM, with some simplification and extensions on the original analysis made to fit the purpose of building data dependence graph from C programs in LLVM intermediate representation (IR). Based on the SSA form, optimizations are crafted to reduce redundant computation in copying the abstract store. And new semantics are added to accommodate instructions like phi nodes. After the implementation of the points-to analysis, we propose a context- and flow-sensitive data dependence analysis on top of that, which is also modular and relies on a theorem prover.

In our evaluation, the technique presents a considerable precision improvement compared to our baseline in term of the number of RAW relations, thanks to flow-sensitivity, more precise alias analysis and manual annotations on library functions. It has a practical performance overhead and memory footprint as our implementation only has up to 24x slow-down in our benchmarks and low memory usage well under gigabytes.
Chapter 2

Background

2.1 Control Flow Graph

Control flow graph is a graph representation of programs[All70], in which instructions are put into basic blocks that are connected by control edges. A basic block has only one path of execution, i.e. a sequence of instructions without any jumps. And connecting them are control edges that represents potential jump directions. To form a proper program, there should also be an entry block that has no incoming edges and an exit block that has no outgoing edges. Such organization of the program could help us traverse instructions along control edges and thus significantly simplify the downstream program analysis and transformations. In our analysis, control flow graph is extensively utilized to compute both points-to and dependence summaries.

2.2 Alias Analysis

Alias analysis is a technique that determines if two names alias, i.e. refer to the same object. The object could be variables, memory regions or even program values. Given a query of two names, an alias analysis should report whether they never/may/must alias depending on the confidence we have on that relation. In terms of precision, an alias analysis is context-sensitive if it distinguishes different call sites of a function, flow-sensitive if it respects control flows, and field-sensitive if it recognizes different fields of an aggregate structure. Typically, a sound data
dependence analysis would require a may-analysis and more precise the alias analysis is, often
the more precise our data dependence analysis would be. Most alias analysis is constructed upon
a points-to analysis[Lin15], which computes the points-to relations among the potential pointers
and memories regions. Intuitively, pointers into the same location are alias.

2.3 Abstract Interpretation

Abstract interpretation is a sound approximation of the program semantics. In formal static
analysis, we design an abstract semantics for the underlying programming language and perform
a partial execution upon the program to gather information about the possible outcomes of real
executions. In the presence of loops and recursions, the interpretation should be defined based on
monotonic functions over a lattice such that it always converges in a finite number of steps. The
reached fixed-point state is the desired approximation.
Chapter 3

A Precise and Modular Points-to Analysis

A major part of this research is to implement the precise points-to analysis[DDAS11] in LLVM as a foundation of our data dependence analysis. It is a technique based on abstract interpretation and constraint solving. We compute summaries of functions bottom-up from the call graph, which are graphs that abstract the run-time memory and capture side effects after program execution. The graph, named symbolic abstract store, displays points-to relations among different abstract locations in all calling contexts.

3.1 Memory Abstraction

The points-to analysis operates upon a store-based memory abstraction, an abstract store, which is a directed graph $S = (V, E)$. Vertices in the graph are abstract memory locations and edges represent points-to relations. Based on that, a symbolic abstract store is a function $f : C \rightarrow S$ where $C$ is a set of premises that encodes some calling context and $S$ is the consequent memory abstraction deduced from that context. The process of function application is called an instantiation. It could also be presented as a graph $S' = (V, E')$ where each edge $e \in E'$ is associated with a constraint $c$. The specific edge only exists after instantiation when the attached constraint $c$ is satisfied, and should be eliminated otherwise.
3.1.1 Abstract Memory Location

The nodes in our abstract store is an abstract memory location, or \textbf{abstract location} in short. It could be classified as either \textbf{location constant} or \textbf{location variable}.

A location constant is some location with a deterministic definition, typically locations allocated locally or obtained in invoked subroutines. It may represent a program name, non-overlapping concrete memory regions or program terms. In the context of LLVM, a program name is the name of a variable, in most cases, the produced value of an instruction. A concrete memory region represents static/stack/heap memory from allocation. And a program term is a physical value such as 42. If we have two different location constants, either they don’t alias or the fact of aliasing does not contribute to different analysis outcomes. It is worth noting that a single location constant could correspond to multiple concrete memory regions, e.g. an array or some recursive data structure, or multiple program terms. Such locations are also referred as \textbf{summary locations}. For example, a \texttt{malloc} call in a loop always returns one location constant, though we may not get the same memory in different iterations. Aside from the particular instruction, the same location constant cannot be created.

A location variable, however, has no known definition and often represents those memories passed in by function arguments. In a bottom-up analysis, apparently, we do not know where the definition of arguments reside until call sites are analyzed. We use location variables as placeholders during the summary computation and substitute them with actual abstract locations at the moment of summary instantiation.

3.1.2 Points-to Constraints

In a symbolic abstract store, each edge is associated with a \textbf{points-to constraint}, indicating under which condition a points-to relation holds. This practice allows us to create a single summary for all possible incoming aliasing environment and also act like a cache that helps avoid redundant computation for different call sites. Instead of a simple logic expression, we are using a bracketing constraint\cite{DDA10}, which is a pair of two logical terms $\langle \varphi_{\text{may}}, \varphi_{\text{must}} \rangle$ that respectively shows when the points-to relation may/must hold. This is essential for us to
construct a sound may-analysis while taking some advantage of must-analysis in situations such as strong-update. Following operations are defined on $\phi = \langle \varphi_{\text{may}}, \varphi_{\text{must}} \rangle$:

\[
\neg \varphi = \langle \neg \varphi_{\text{must}}, \neg \varphi_{\text{may}} \rangle \\
\varphi \lor \varphi' = \langle \varphi_{\text{may}} \lor \varphi'_{\text{may}}, \varphi_{\text{must}} \lor \varphi'_{\text{must}} \rangle \\
\varphi \land \varphi' = \langle \varphi_{\text{may}} \land \varphi'_{\text{may}}, \varphi_{\text{must}} \land \varphi'_{\text{must}} \rangle \\
\text{join}(\varphi, \varphi') = \langle \varphi_{\text{may}} \lor \varphi'_{\text{may}}, \varphi_{\text{must}} \land \varphi'_{\text{must}} \rangle
\]

Intuitively, we have $\varphi_{\text{must}} \Rightarrow \varphi_{\text{may}}$ because $\varphi_{\text{must}}$ is a stronger proposition. We define $\lfloor \varphi \rfloor = \varphi_{\text{must}}$ and $\lceil \varphi \rceil = \varphi_{\text{may}}$, and inner term $\varphi$ is a composition of conjunction, disjunction and negation of logic terms $T$ for truth, $F$ for falsehood or alias$(l_i, l_j)$ which indicates two location variables $l_i$ and $l_j$ alias. In some context, $\varphi$ is also used as an abbreviation of $\langle \varphi, \varphi \rangle$.

### 3.2 Analysis Overview

The analysis is modular; that is, we inspect one function at a time and build its points-to summary. A summary is the consequent symbolic abstract store after interpretation of the program with the approximating abstract semantics. In the form of a hoare triple, it is put as $\{S_0\} f \{S_1\}$ where $S_0$ is the initial store, $S_1$ the resulting store and $f$ the function.

Like normal program semantics, our analysis is defined by a set of transformation rules $\{S\} I \{S'\}$ with $S$ being the current program state before instruction execution. Those rules approximate the runtime behaviors of various instructions and describe how they would update the abstract store. By chaining the instructions in a basic block $B$, we could easily derive the semantics of the block as $\{S\} B \{S'\}$. Supposing our block $B$ has predecessors $B_1, \ldots, B_n$, an initial store $S = \text{join}(S'_1, \ldots, S'_n)$ is computed where $S'_i$ is the outcome of $B_i$. Additionally, one special rule is added to build the very first abstract store for the entry block that has no predecessor at all.

In the presence of back edges in the control flow graph, typically from branches and loops in higher-level languages, we define a bottom store, which is a fully-connected graph with all abstract locations and every edge is associated with the $F$ constraint; that is, there is no edge
effectively. This is used as a default analysis result for all basic blocks. The resulting abstract store of each block is repetitively computed until a fixed point is reached, thus yielding the summary for the entire function, i.e. that of the exit block. A similar mechanism should be applied for recursive functions, where each strongly connected component (SCC) in the call graph is analyzed at a time, computing an over-approximation for the points-to summary of each function in the SCC.

The combination of two abstract stores is defined as a join operation. Given two stores $S_1 = (V_1, E_1)$ and $S_2 = (V_2, E_2)$, consequent $join(S_1, S_2) = (V, E)$ has vertices $V = V_1 \cup V_2$. The points-to relation is a little more complicated. We define the extended edge set:

$E'_1 = \{ (u, v, c) \mid (u, v, c) \in E_1 \} \cup \{ (u, v, F) \mid u \in V, v \in V, (u, v, _) \notin E_1 \}$

$E'_2 = \{ (u, v, c) \mid (u, v, c) \in E_2 \} \cup \{ (u, v, F) \mid u \in V, v \in V, (u, v, _) \notin E_2 \}$

Then we have $E' = \{ (u, v, join(c_1, c_2)) \mid (u, v, c_1) \in E'_1, (u, v, c_2) \in E'_2 \}$ and $E = \{ (u, v, c) \mid (u, v, c) \in E', c \neq F \}$. Basically, we first extend the edge set by adding edges with constraint $F$ such that the graph is fully connected on $V$, reduce the two graphs with a join operation of bracketing constraints on edges with the same source and target locations, and finally normalize the graph by filtering out edges that have unsatisfiable constraints. This operation is obviously associative, so $join(S_1, \ldots, S_n) = join(join(S_1, S_2), \ldots)$. 

Figure 3.1. Analysis Overview
3.3 Initialization of Abstract Store

Analysis for each function starts with an initial abstract store, which depicts shape of the memory before function execution. It shows how our parameter registers point to external locations and how they alias with each other. Moreover, for all global variables that are used in the current function or the subsequently called functions, they are also viewed as function parameters. In LLVM IR, they are pointers that point to some static memory regions. Suppose we have a function defined in the syntax:

```
define <ret_ty> @func(<ty> %p1, ..., <ty> %pn) ...
```

It has parameters `%p1, ..., %pn`, which could be pointers to arbitrary external memory locations and even more complicated they could be recursive, pointing to other pointers. Aliasing condition may be very complicated on a trivial construction of abstract store.

First, we have to collect required location variables to represent all external locations. Consider the points-to chain `%p1 → %p2 → *%p3 → ...`, the location variables `*%p3` are created for every level of pointers, including that in aggregate types. Because location variables are essentially placeholders for possibly multiple location constants. Those locations may actually alias with each other. When the proposition `alias(%p1, %p2)` holds, `p1` and `p2` could be instantiated by the same location constants in a certain condition. Such relations are captured by many pointing edges into the aliased locations, constraints of which encode different call sites. In order to reduce the number of aliasing edges, we define a total order on all location variables and require an alias edge only goes to the locations of the smaller. On top of that, many alias rejection strategies like different types, different global variables, different read-only variables are also applied to simplify the initial abstract store, which could have a major performance impact on subsequent analysis as the size of the abstract store generally determines the analysis run time.

Figure 3.2 is an example initial store for a function of three parameters, assuming all of them are pointers. First of all, we have location variables `p1, p2, p3` and so on, formulating an order `p1 < p2 < p3`. Then, we set up the points-to edges. Starting from `%p1`, we have only one choice, `p1`. The constraint is `T` because of reflexivity. For `%p2`, we have two memory configurations, pointers in `%p1` and `%p2` alias or the opposite. When two locations alias, obviously, `%p2` should
point to $p_1$ and the constraint is $\text{alias}(p_1, p_2)$. Oppositely, negation of the constraint is used for the non-alias edge. Similarly, we connect $\%p_3$ to all of $p_1, p_2, p_3$ with the corresponding constraints.

### 3.4 Example

In this section, we illustrate how the points-to analysis works in a simple example. The function $\text{swap}$ below takes two pointers of integers and swap the content. Depending on whether two input pointers alias or not, we show different conclusions can be drawn.

```c
void swap(int * p, int * q) {
    int tmp = *p; *p = *q; *q = tmp;
}
```

Figure 3.3 is the initial store for the function. Note $\%p$ and $\%q$ actually have type $\text{int **}$ and are pointers pointing to the stack memory where the arguments are stored. There are two types of input environments, $p$ alias with $q$ or not, reflected in the constraints associated with edges from $\%p$ and $\%q$. Because the pointed memory regions by $p$ and $q$ save non-pointer values, they are denied from being alias and we have $T$ constraint for edges from the two locations.

Figure 3.4 is the resulting summary. Based on the given initial store, we see two outcomes.
for the function 

\(\text{swap}\). When two input pointers are not alias, the content of two pointed memory regions are swapped as intended; \(p\) now points to \(*q\) and \(q\) to \(*p\). However, when aliasing pointers are provided, both \(%p\) and \(%q\) possesses pointer \(p\) and it still points to its original value \(*p\). Note the edge from \(q\) to \(*q\) still exists in this case, but it is isolated from starting point \(%p\) and \(%q\) and would be removed in summary instantiation.
Chapter 4

Points-to Analysis in LLVM

In this chapter we introduce the points-to analysis implementation details in LLVM. In specific, we present the abstract semantics of common LLVM IR instructions and show algorithms about how they should transform the input abstract store. Most of them are translated from the original formal deduction rules but there are also extensions for Phi nodes, cast instructions and other intrinsic function calls. Finally, We also briefly discuss some performance optimizations to speed up the summary computation.

4.1 Syntax And Terms

Before diving into detailed explanations of abstract semantics, we should set up a syntax and term convention for the algorithm description. As we are statically interpreting LLVM programs in our analysis, we see LLVM as a virtual instruction set architecture (ISA) with an unlimited number of registers and one unified memory. An LLVM IR instruction, in the syntax below, would perform some computation and stores the result back in the register \%ret. Because of SSA form, \%ret is a dedicated register for the particular instruction and cannot be reassigned. Redefinition, however, is allowed, which can update the edges when the same instruction is revisited.

\%
ret = inst ...

An abstract location, vertex in the abstract store, is named in \%loc for LLVM registers and
for other terms. Constraints are put in $c_{loc}$ or written in the form of $(T, F)$ for literals. An edge, i.e. a points-to relation, is described as $loc \rightarrow loc' @ c_{loc}$, meaning $loc$ points to $loc'$ if $c_{loc}$ is satisfied.

The following operations on the abstract store are defined:

- **clear**: We remove all edges from the specific source, emulating overwriting or redefinition.
- **add**: We add an edge to the graph. If an edge of the same source and target already exists, a disjunction of old and new constraints are computed.
- **update**: We add an edge to the graph. If an edge of the same source and target already exists, the old one is replaced.

The difference between add and update is about how to deal with old edges. As sometimes edges of the same source and target could be added by different actions, they act like strong- and weak-updates. While add preserves the old relations, update overwrites them.

### 4.2 Allocation Instruction

An allocation instruction allocates a new memory region. Specifically, it could be an `Alloca` instruction for stack memory or an intrinsic call to functions like `malloc` in C’s run-time library for heap memory. Such instructions always return a new location constant as shown in Algorithm 1 and Figure 4.1.

**Algorithm 1: Abstract Semantic of Allocation Instruction for Non-summary Location**

1. clear $%x$;
2. add $%x \rightarrow AllocatedLocation @ T$;

Note that the constraint of the added points-to relation depends on the property of allocation and may not necessarily be a top $T$. If the allocated location is a summary location, i.e. arrays or other recursive structures, because the constructed location constant could be mapped to multiple
concrete memory regions, writes into the location may not erase the old values. We have to perform weak-update on such pointers for a sound analysis and use constraint \((T, F)\) instead. How strong-update and weak-update are enforced will be explained in the following section.

Because a `malloc` call does not actually contain type information, identification of non-summary locations may not be trivial. So we presume summary location allocation as a good default and recognize non-summary counterpart on a specific pattern of instructions that an upstream C compiler generates, which is reasonable as a heap allocation is usually for buffers and compound structures.

Also note the `clear` operation in the beginning of the Algorithm 1. It removes all edges from the source location \(A_x\) in the case of redefinition. Redefinition is triggered when the same basic block is visited again during the fixed point computation. Although keeping the old edges should not affect the analysis result because of monotonicity of abstract interpretation. Removing them helps us to simplify the syntax complexity of our points-to constraints to be fed into the theorem prover.

### 4.3 Store Instruction

The `Store` instruction is the only instruction in LLVM IR that could directly write into a pointer. It looks like:

```
store <ty> %val, <ty>* %ptr
```

For the instruction above, it finds pointer \(ptr\) in the register \(ptr\) and value \(val\) to be written in \(val\), and then writes \(val\) into the pointed memory of \(ptr\). It is worth mentioning that our points-to relations exist only when the associated constraint is satisfied. Considering \(ptr\) only
points to $ptr$ when $c_{ptr}$ is satisfied, in addition to write new values with the constraint $c_{ptr}$, we need also to keep the old values with the constraint $\neg c_{ptr}$. The process is illustrated in Algorithm 2 and Figure 4.2.

**Algorithm 2: Abstract Semantic of Store Instruction**

```plaintext
1 for all \%ptr $\rightarrow$ ptr @ $c_{ptr}$ do
2     forall ptr $\rightarrow$ oldval @ $c_{oldval}$ do
3         update ptr $\rightarrow$ oldval @ $\neg c_{ptr} \land c_{oldval}$;
4     forall \%val $\rightarrow$ val @ $c_{val}$ do
5         add ptr $\rightarrow$ val @ $c_{ptr} \land c_{val}$;
```

For a non-summary location, we have $c_{ptr} = T$, so a strong-update is applied; edges to old values are removed since $\neg c_{ptr} \land c_{oldval} = F \land c_{oldval} = F$.

For a summary location, however, typically $c_{ptr} = (T, F)$. We have the following equation:

$$\neg c_{ptr} = \neg(T, F) = (\neg F, \neg T) = (T, F) = c_{ptr}$$

Location $ptr$ now points to both new and old values on constraint $(T, F)$, which makes it a weak-update. Suppose we have deterministic pointed value where $c_{val} = T$ and $c_{oldval} = (T, F)$.
after the store instruction, now \texttt{ptr} should point to both \textit{oldval} and \textit{val} with the constraint \((T, F)\).

As a generalization, the constraints of a summary location pointer and its existing pointing edges are always in the form \((\varphi, F)\). It reflects indeterminism of dereferencing a summary pointer. By enforcing this rule, the following equations prove that old points-to edges are always preserved with the same constraint while new edges are added.

\[
\neg(\varphi_{ptr}, F) \land (\varphi_{old}, F) = (T \land \neg\varphi_{ptr} \land F) = (\varphi_{old}, F)
\]

\[
(\varphi_{ptr}, F) \land (\varphi_{new}, F) = (\varphi_{ptr} \land \varphi_{new}, F)
\]

### 4.4 Load Instruction

In contrast to store instructions, a \textit{Load} instruction reads from the pointed memory and puts the value into the specific register. It has syntax:

\[
\%x = \text{load } \texttt{<ty>}, \texttt{<ty>\* }\%\texttt{ptr}
\]

Like the situation above, register \(\%\texttt{ptr}\) may contain pointer \(\texttt{ptr}\) on \(c_{ptr}\) which further points to value \(\texttt{val}\) on \(c_{val}\). Obviously, \(\%\texttt{x}\) should point to \(\texttt{val}\) only when both \(c_{ptr}\) and \(c_{val}\) are satisfied. The edge is added on a conjunction of both conditions, shown in Algorithm 3 and Figure 4.3.

**Algorithm 3: Abstract Semantic of Load Instruction**

1. clear \(\%\texttt{ptr}\);
2. \texttt{forall} \(\%\texttt{ptr} \to \texttt{ptr} @ c_{ptr}\) \texttt{do}
3. \texttt{forall} \(\texttt{ptr} \to \texttt{val} @ c_{val}\) \texttt{do}
4. add \(\%\texttt{x} \to \texttt{val} @ c_{ptr} \land c_{val}\);
Phi nodes and Select instructions are two types of instructions that do conditional assignment. Phi depends on which block the control flow came from and Select relies on a run-time boolean condition. Because our analysis is path insensitive, intuitively, we join the points-to constraints of the edges from and into the same location. Disjunction is computed on may-constraint and conjunction on must-constraint for the new points-to relation, because either may-edge could contribute to the new one while the must condition requires all constraints to be satisfied. This is very similar to how we initialize the abstract store for a basic block. When an edge doesn’t exist at all, we behave as if it is there with the constraint $F$. In Algorithm 4 and Figure 4.4 are illustrations that how a binary phi node changes the abstract store.

**Algorithm 4: Abstract Semantic of Phi Instruction with Two Operands**

```
1 clear %x;
2 forall %a → val @ c_a and %b → val @ c_b do
3     add %x → val @ join(c_a, c_b);
```

Phi nodes in real-world can have more than two operands. In fact, a switch statement in C could compiles into a Phi node, whose operand number equals the number of non-fall-through cases. As the joining operation on bracketing constraint is both commutative and associative, join on many constraints is trivially defined.
4.6 Bitcast/GetElementPtr Instruction

In LLVM IR, there are many pointer cast instructions, commonly *Bitcast* or *GetElementPtr*. *BitCast* allows LLVM to reinterpret a value’s binary representation and *GetElementPtr* offsets the pointer for the address of a data type member. Due to the field-insensitivity of our analysis, we treat the cast pointer as an alias of itself. Trivially, those instructions are actually identity operations in our abstract semantics and we could simply copy the points-to edges. For example, if we have a *BitCast* instruction as shown below, Algorithm 5 and Figure 4.5 shows what we should do.

%\( x = \text{bitcast } \langle \text{ty} \rangle \text{ } \%\text{ptr to } \langle \text{ty2} \rangle \)

---

**Algorithm 5**: Abstract Semantic of BitCast Instruction

1. clear %\( x \);
2. forall \( \%\text{ptr} \rightarrow \text{ptr} @ c_{\text{ptr}} \) do
3. \[ \text{add } \%x \rightarrow \text{ptr} @ c_{\text{ptr}}; \]

For a *GetElementPtr* put in the format below, the same algorithm should be applied.

%\( x = \text{getelementptr inbounds } \langle \text{ty} \rangle^{*} \%\text{ptr} \ldots \)
4.7 Call Instruction

Call instructions are the most sophisticated part of our analysis. We have to instantiate the already computed summary of a particular function and integrate its side effects into the current analysis context. Specifically, it could be done in three steps:

1. Rewriting Constraints: Alias terms should be substituted because location variables are instantiated with abstract locations at the call site.

2. Mapping Locations: Location variables should also be replaced for the same reason above.

3. Merging Stores: Abstract store after the call instruction should reflect observable pointer operations inside the called function.
Algorithm 6: Abstract Semantic of Call Instruction: Rewriting Constraints

1 forall pair of inputs \((p_i, p_j)\) where \(p_i \neq p_j\) do
2 \[ \text{newterm} = F; \]
3 forall loc where \(\%a_i \rightarrow loc \odot c_i\) and \(\%a_j \rightarrow loc \odot c_j\) do
4 \[ \text{newterm} = \text{newterm} \lor (c_i \land c_j); \]
5 replace term \(\text{alias}(p_i, p_j)\) with \(\text{newterm};\)

Algorithm 7: Abstract Semantic of Call Instruction: Mapping Locations

1 forall \(\%a_i \rightarrow ai \odot ... \rightarrow ai^k \odot c_{ai^k}\) do
2 forall \(\%p_i \rightarrow pi \odot ... \rightarrow pi^k \odot c_{pi^k}\) do
3 \[ \text{map} ai^k \iff pi^k \odot (c_{ai} \land ... \land c_{ai^k}) \land (c_{pi} \land ... \land c_{pi^k}); \]

The first two steps are relatively simple, as they are shown in Algorithms 6 and 7. Note map is an operation that constructs a bidirectional mapping between parameters and arguments, enabling us to replace location variables in the callee with abstract locations in the current context.

Consider a call instruction in the following syntax:

\[
%\text{ret} = \text{call <ty>} \ \text{func}(%a_1, \ldots, %a_n)
\]

We already know that each register \(\%a_i\) could have different values on different constraints. At the call site, for each \(\%a_i \rightarrow a^k_i \odot c_k\), location variable \(p_i\) could be instantiated into \(a^k_i\) on the particular constraint. Therefore, the alias term \(\text{alias}(p_i, p_j)\) is satisfied on the conjunction of conditions such that both \(\%a_i\) and \(\%a_j\) points to the same location. Also, \(c_k\) is the constraint of the instantiation of \(p_i\) into \(a^k_i\), yielding the mapping \(p_i \iff a^k_i \odot c_k\).

Then we have to reproduce the side effects. In Algorithm 8, for each abstract location that could be passed into the function invocation, we copy the points-to edges from the instantiated summary. Like store instructions, it requires the location be actually passed into the callee so we should update it. If not, old points-to relations are to be preserved with more strict constraints. On top of that, new location constants may be introduced as the callee may request
Algorithm 8: Abstract Semantic of Call Instruction: Merging Stores

1. **forall** loc\_a where loc\_a $\leftrightarrow$ loc\_p exists do
   1. clear loc\_a;
   2. $c_{passin} = F$;
   3. **forall** loc\_a $\leftrightarrow$ loc\_p \_eq do
      4. $c_{passin} = c_{passin} \lor c_{eq}$;
   4. **forall** loc\_p $\rightarrow$ val\_p \_eq \_pt do
      5. if val\_p is location variable then
         6. **forall** val\_a $\leftrightarrow$ val\_p \_eq do
            7. add loc\_a $\rightarrow$ val\_a \_eq \_pt \_valold;
         8. else
            9. add loc\_a $\rightarrow$ rename(val\_p) \_eq \_pt;
      10. **forall** loc\_a $\rightarrow$ val\_valold \_valold do
          11. add loc\_a $\rightarrow$ val\_valold \_eq \_passin \_valold;
      12. **forall** %ret $\rightarrow$ val\_ret \_ret do
          13. if val\_ret is location variable then
             14. **forall** val\_a $\leftrightarrow$ val\_ret \_eq do
                15. add %x $\rightarrow$ val\_a \_ret \_valtest;
             16. else
                17. add %x $\rightarrow$ rename(val\_ret) \_eq \_valtest;
          18. **else**
             19. add %x $\rightarrow$ rename(val\_ret) \_eq \_valtest;
new memory regions from heap. Those locations are renamed, with syntax $\text{rename}(\text{loc})$ in algorithm description, for context-sensitivity and are brought into the abstract store of the current context.

Consider the following function $\text{do\_swap}$, which simply forwards its two parameters into another function $\text{swap}$. From the last chapter, we know $\text{swap}$ has a summary shown in Figure 4.6 and similarly $\text{do\_swap}$ has an initial store like Figure 4.7. Note in Figure 4.6, alias stands for $\text{alias}(p, q)$ while in Figure 4.7 it is $\text{alias}(a, b)$.

```c
void swap(int* p, int* q) {
    int tmp = *p; *p = *q; *q = tmp;
}
void do_swap(int* a, int* b) {
    swap(a, b);
}
```

In LLVM IR, $\text{do\_swap}$ (simplified, not generated by a compiler) has the following definition:

```llvm
define dso_local void @do_swap(i32* %a, i32* %b) #0 {
  entry:
  call void @swap(i32* %a, i32* %b)
  ret void
}
```

First, we rewrite the constraint terms. In $\text{do\_swap}$, arguments $%a$ and $%b$ can point to location $a$ and $b$. To list them, we have:

- $%a \rightarrow a \oplus T$
According to Algorithm 6, we have the equation:

$$alias(p, q) = F \lor (T \land alias(a, b)) \lor (F \land \neg alias(a, b)) = alias(a, b)$$

Second, we map locations in swap and do_swap. Trivially, we have $%a \iff %p @ T$ and $%b \iff %q @ T$. In the context of the caller, three points-to chains are present, $%a \rightarrow a \rightarrow *a$, $%b \rightarrow a \rightarrow *a$ and $%b \rightarrow b \rightarrow *b$. In the summary, we only inspect $%p \rightarrow p \rightarrow *p$ and $%q \rightarrow q \rightarrow *q$. After six iterations in Algorithm 7, we should obtain:

- $a \iff p @ T$
- $b \iff p @ alias(a, b)$
- $b \iff q @ \neg alias(a, b)$
- $*a \iff *p @ T$
- $*b \iff *p @ alias(a, b)$
- $*b \iff *q @ \neg alias(a, b)$
Finally, we enumerate the locations that are passed in, \( a, b, \ast a, \ast b \), and copy the edges in the instantiated summary back to the current store. For example, as we have \( b \iff p @ alias(a, b) \), \( p \rightarrow \ast p @ alias(a, b) \) and \( \ast a \iff \ast p @ T \), the edge \( b \rightarrow \ast a @ c \) is added where \( c = alias(a, b) \land alias(a, b) \land T = alias(a, b) \). After all mappings are visited, our abstract store for function \( \text{do_swap} \) would have exactly the same shape as that of \( \text{swap} \).

### 4.8 Other Instructions

Besides instructions in previous sections, we also have some LLVM intrinsic functions that operate on pointers. Commonly used are:

- \texttt{llvm.memcpy}: Same as \( \%x = \text{load} \%\text{src}; \text{store} \%x, \%\text{dst} \)
- \texttt{llvm.memmove}: Same as \texttt{llvm.memcpy}
- \texttt{llvm.memset}: Same as \texttt{store}

These functions could also affect underlying points-to relations and should be processed like an independent instruction. There are actually more of them and other pointer-related instructions, yet because those are rare in common LLVM IR programs, they are not implemented in this research.

For instructions that don’t work with pointers at all, we could simply assume they produce some unknown values that we don’t care about. As our interest to identify RAW relations behind pointers, what those instructions are doing is ultimately irrelevant to our analysis.

### 4.9 Performance Optimization

Because the analysis is built on fixed point computation on abstract stores, which have a fairly large memory footprint and are expensive to compare in equality due to many theorem prover invocations, a trivial implementation could easily fail to scale. In this section, we introduce some optimizations to tackle the performance issues.
First of all, the output abstract stores among different basic blocks are redundant. Many points-to edges are created in ancestors and not going to be changed or even used ever since. In particular, because LLVM IR uses SSA form, edges starting from a register is read-only after the assigning instruction. In implementation, points-to edges from such locations are stored in a global array per context regarding to the linear layout of function instructions. By distinguishing two types of source locations, we could significantly cut the number of edges in the abstract store to be saved and copied during the analysis.

Second, due to expensive equivalence testing of constraints, we want to minimize the number of edges we compare in practice. We track the convergence of preceding blocks and skip one store comparison if we know that the current basic block won’t be visited again. During a comparison, we also have heuristics that prioritize that from locations whose outgoing degrees are changed, aiming to exit early if it’s an inequality anyway.

Finally, we are building the points-to analysis specifically for the following data dependence analysis. That is, we only care about the aliasing properties of pointers but not values. By eliminating the non-pointer locations in the abstract store and edges targeting them, we could further reduce the size of stores, saving memory usage and lowering equality comparison overhead.
Chapter 5

Data Dependence Analysis

In this chapter, we present the design of data dependence analysis with points-to summaries. First, we are applying a data flow analysis with constraints to trace the sources of assignment for each abstract location. Second, we inspect every instruction that may read from an abstract location and construct dependence edges between it and its assignment sources under the constraint given, producing a dependence summary. Finally, we may instantiate the summary with a context to obtain RAW edges of data dependence graph for the specific program, or we may simply drop the constraints to make an approximation. Like the points-to summary, a dependence summary is also a graph, but instead of modeling points-to relations, it has instructions (or other sources of assignment) as nodes and its edges mark potential RAW dependence.

5.1 Analysis Overview

The first step of our analysis is to build a dependence mapping $M : loc \rightarrow \{(s, c)\}$ that describes sources of assignment into the memory region of $loc$, where $loc$ is an abstract location, probably a pointed memory region, $s$ a source of data assignment (typically instructions, arguments or global variables), and $c$ a bracket constraint. A triple $(loc, s, c)$ indicates that under the constraint $c$ there exists a points-to relation $loc \rightarrow loc'$ due to execution of source $s$, where $loc'$ is an arbitrary abstract location. Notably a source $s$ is not necessarily an instruction, it could also be an argument or a global variable that gives an initial value. We provide a set of rules of how a variety of
sources could update this mapping, and then how such a mapping could be utilized to construct the set of RAW relations.

On the aforementioned dependence mapping, an operation \( M' = update(loc, c, s, M) \) is defined:

\[
M'(l) = \begin{cases} 
M(l) & , l \neq loc \\
\{(s_i, \neg c \land c_i)\} \cup \{(s, c \lor c_{src})\}, (s_i, c_1), (s, c_{src}) \in M(loc), s_i \neq s & , l = loc 
\end{cases}
\]

It modifies the mapping from location \( loc \) to emulate a conditional write only when \( c \) is satisfied. On condition \( c \) that the overwriting actually happens or on \( c_{src} \) that \( s \) is already a source for \( loc \), the provided source instruction \( s \) contributes to the content of \( loc \) after this update. Otherwise, old dependence sources are kept and \( \neg c \) is used to strengthen the constraints of other sources.

Say we have a mapping \( M = \{loc \mapsto \{(s_1, c_1), (s_2, c_2)\}, \ldots\} \). An \( update(loc, c, s_1, M) \) attempts to conditionally overwrite the source from \( loc \) to \( s_1 \), building a new mapping \( M' \). When \( c \) is satisfied, \( M'(loc) = s_1 \) should always hold. However, when \( \neg c \) is satisfied, we have to keep the old sources according to the old mapping. In this case, they are \( \neg c \land c_1 \) for \( s_1 \) and \( \neg c \land c_2 \) for \( s_2 \). As a result, \( M' = \{loc \mapsto \{(s_1, c \lor c_1), (s_2, \neg c_1 \land c_2)\}, \ldots\} \). In our analysis, \( update \) is invoked on the particular instructions that may write into the pointers thus changing the dependence relations afterwards.

On the other hand, an operation \( M' = merge(M_1, M_2) \) is responsible to combine the dependence mappings from different preceding basic blocks. We have the definition:

\[
M'(l) = \{(s, join(c_1, c_2))\}, (s, c_1) \in M_1, (s, c_2) \in M_2
\]

The mechanism underneath is also similar to that of the points-to analysis and quite self-explanatory. We join the mapping from the same location and to the same source where disjunction of may-constraint and conjunction of must-constraint are computed. Intuitively, if a may-dependence exists from preceding block \( B_1 \) on \( c_1 \) and \( B_2 \) on \( c_2 \), it may exist in the current block on \( c_1 \lor c_2 \). And a conjunction on constraints should happen for a must-dependence. Note this operation is also trivially defined for multiple parameters thanks to associativity and commutativity of the \( join \) operation.
5.2 Dependence Mapping Computation

Start from an empty dependence mapping at the entry of the function, we define the state transfer function. Typically for instructions that do not alter the dependence relations, it is an identity operation that carries the same mapping to the exit points of the instruction. Otherwise, it is a composition of many partial applications of the update function where \( s \) is the current instruction (or parameter):

- Function parameters: For all pointed location variables \( \text{loc} \), do \( \text{update}(\text{loc}, T, s) \)
- Allocation instructions: Do \( \text{update}(\text{loc}, T, s) \) where \( \text{loc} \) is the allocated location
- Store instructions: For all \( \%\text{ptr} \to \text{ptr} @ c_{\text{ptr}} \), do \( \text{update}(\text{ptr}, c_{\text{ptr}}, s) \)
- Call instructions: For all \( \text{loc}_{\text{ptr}} \) passed in and written in callee, do \( \text{update}(\text{loc}_{\text{ptr}}, c_{\text{passin}}, s) \)

Most of the time, those sources query the points-to graph and update the mapping accordingly. We do conditional update only when the instruction actually writes into the pointer. Like the points-to analysis, the complex part is how we deal with call instructions. Because a function call could potentially modify any pointed locations depending on its body, the most conservative guess is that all of them would be updated after the call. Although sound as it is, this is imprecise since many locations are never written. To overcome this, we could perform a prior inexpensive and non-symbolic analysis to find if a location could be written in the callee.

\[
%x = \text{load} <\text{ty}>, <\text{ty}>* %\text{ptr}
\]

Now that a dependence mapping is computed for every point of program execution, we inspect all pointer-reading instructions, load instructions in particular. Given the load above and assuming \( %\text{ptr} \to \text{ptr} @ c_{\text{ptr}} \), the load instruction \( l \) actually depends on the last of which source writes into location \( \text{ptr} \) on \( c_{\text{ptr}} \). Looking it up in the dependence mapping for \( \text{ptr} \to (s, c_s) \), we know \( (l, s) \) is a RAW pair when \( c_{\text{ptr}} \land c_s \) is satisfied.

The strong-update advantage comes from the strong-update-aware points-to analysis and the bracketing constraint. Besides the precise points-to graph across the execution, a use of update
with \( T \) condition erases the old dependence while that with \( (T, F) \) preserves them. Thanks to the SSA form, we actually don’t have to save abstract stores for every program point. The points-to edges from LLVM registers, what we inspect in this analysis, are always latest and correct in the points-to summary as only the defining instructions could change them.

\[
\begin{align*}
s1: & \text{ store } i32 \ %val1, \ i32* \ %ptr, \ \text{align 4} \\
s2: & \text{ store } i32 \ %val2, \ i32* \ %ptr, \ \text{align 4} \\
l: & \ %x = \text{ load } i32, \ i32* \ %ptr, \ \text{align 4}
\end{align*}
\]

For example, the code snippet above has a load instruction immediately following two store instructions that operate on the same pointer register \%\text{ptr}.

If \%\text{ptr} contains a pointer to a non-summary location, points-to analysis may report \%\text{ptr} \to \text{ptr} \at \ T, \text{i.e.} \ c_{\text{ptr}} = T. What values \%\text{val1} and \%\text{val2} have does not matter here because we aim to find which instructions write into the pointers instead of what are written. Assuming an empty dependence mapping before \( s_1 \) and \( s_2 \), the interpretation of the first store should do \text{update}(\text{ptr}, T, s_1) \ and \ result \ in \ a \ mapping \ \text{ptr} \to \{(s_1, T)\}. The second store, then, should perform \text{update}(\text{ptr}, T, s_2), which transforms the mapping into \text{ptr} \to \{(s_1, \neg T \land T), (s_2, T \lor F)\}. When the load instruction is visited, we have constraint \( c_{s_1} = F \) for the source \( s_1 \) and \( c_{s_2} = T \) for the source \( s_2 \). After computing the dependence constraint \( c_s \land c_{\text{ptr}}, \) effectively \((l, s_2)\) represents a RAW relation while \((l, s_1)\) does not.

When \text{ptr} is a summary location, however, a weak update should be applied. With \( c_{\text{ptr}} = (T, F) \), for each store instruction \( s_i \), \text{update}(\text{ptr}, (T, F), s_i) \ is performed and the dependence mapping after \( s_2 \) is shown below:

\[
\text{ptr} \to \{(s_1, \neg(T, F) \land (T, F)), (s_2, (T, F) \lor F)\} = \{(s_1, (T, F)), (s_2, (T, F))\}
\]

We see both store instructions are preserved as potential assignment sources for the location \text{ptr}, where \( c_{s_1} = c_{s_2} = (T, F) \). We know both \((l, s_1)\) and \((l, s_2)\) are RAW pairs with the constraint \( c_{s_i} \land c_{\text{ptr}} = (T, F) \).
In real-world situations, loads and stores that possess a RAW relation do not have to access the same pointer register. Different registers may contain aliasing pointers and the constraints may be much more complicated than a simple $T$ or $(T, F)$. Anyway, the existence of any RAW relations is tracked by the bracketing constraints encoding both weakest and strongest conditions.

5.3 Inter-procedural Dependence

Across the procedural boundaries, a RAW relation may hold between load and store instructions distributed in different functions. A pointer could be written in the caller and read from the callee, and vice versa. Directly capturing those relations is hard as the distribution could be arbitrarily faraway through an unlimited number of function calls and their number could explode. In our analysis, instead of directly encoding this type of dependence in load-store pairs, we split them into a chain of other relations. We introduce RAW dependence between stores and calls, calls and loads, arguments and loads. These relations act as an intermediate bridge, so that we could connect them on need for inter-procedural dependence.

For example, if we want to discover all store-in-callee-load-in-caller dependence with only one level of indirection. We can try to connect the dependence mapping from the callee and load-call RAW pairs in the caller. Location mappings are computed like in the points-to analysis for the particular call instruction, which could be cached in need. If when $c_{ptr}$ is satisfied, a load instruction $l$ reads location $ptr$ from the prior call with the constraint $c_{loadcall}$, $ptr$ maps to location variable $loc$ on $c_{map}$ in the callee, and $loc$, at the end of the called function, is assigned by $s$ on $c_{f}$, there is an indirect RAW pair $(l, s)$ on $c_{ptr} \land c_{loadcall} \land c_{map} \land c_{f}$.

define dso_local void @foo(i32* %p, i32 %x) #0 {
  s: store i32 %x, i32* %p, align 4
      ret
}
...

f: call void foo(i32* %ptr, i32 41)
l: %x = load i32, i32* %ptr, align 4
...

The code snippet above shows a store-in-callee-load-in-caller dependence relation. \((l, s)\) is a RAW pair because in this particular context, the value retrieved by the load instruction \(l\) is written by the store instruction \(s\). Consider the points-to relation \(\%ptr \to ptr \at T\), according to the rules explained in Chapter 5, we have \(c_{ptr} = c_{map} = T\). The call instruction passes location \(ptr\) and a value location 41 into the \(foo\) function, where they are respectively mapped to location variables \(p\) and \(x\). As we know that location variable \(p\) is being altered in every path of \(foo\), the call instruction \(f\) should apply \(update(ptr, T, f)\) to the dependence mapping, obtaining 
\[ptr \to \{(ptr, T), \ldots\}\] and thus \((l, f)\) is a load-call RAW relation with the constraint \(c_{loadcall} = T\).
In function \(foo\), at the exit point, we also have dependence mapping \(p \to \{(s, T)\}\) where value in location variable \(p\) is guaranteed to be written by the instruction \(s\), i.e. \(c_f = T\). By removing the intermediate proxy of the call instruction, RAW pair \((l, s)\) is reported with the dependence constraint \(c_{ptr} \land c_{loadcall} \land c_{map} \land c_f = T\).
Chapter 6

Evaluation

In this chapter, we apply the analysis to both crafted examples and real-world applications, showing results and other statistics. On top of that, we present some case studies on how our analysis displays precision advantages over the baseline while retains a practical performance and memory consumption.

6.1 Experiment Setup

All experiment in this chapter is run in Windows Subsystem Linux, Ubuntu 20.04 with processor Ryzen 3600, 16GB RAM and windows 10 version 19042.846. All test programs are compiled in clang 12 with O0 optimization level to preserve some IR patterns from the C language compiler and ease result inspection. Optimizations may impede our analysis to recognize some non-summary allocations and deteriorate the precision. Tests include hand-crafted examples to show superiority of a more precise analysis as well as programs collected from GNU coreutils, SPEC 2006 benchmark suite and nginx. Precision and run-time statistics are measured with a high-resolution system clock inside the implementation. Memory consumption is measured with valgrind.

The call graph that the analysis relies on is constructed by LLVM prior to the analysis, where indirect calls contribute to an edge targeting to a special node. For each function pointer indirection, we make a conservative guess, joining the abstract stores that are produced by all
functions with the same type signature as the summary.

In addition, our baseline is the program dependence analysis implemented in this git repository [LHT] on commit f44b510. Some modifications are made to disable constructions of non-RAW edges in the data dependence analysis and print comparable statistics.

### 6.2 Examples

```c
void swap(int *p, int *q) {
    int tmp = *p; *p = *q; *q = tmp;
}

int seq(int *p) {
    *p = 41; *p = 42; return *p;
}

int alt(int *p, int b) {
    *p = b ? 41 : 42; return *p;
}

int foo(int *p) {
    alt(p, 1); seq(p); return *p;
}
```

First, we evaluate the analysis on the `swap` function above. Among 5 stores and 7 loads in the IR, our analysis reports 5 load-store RAW pairs compared to the 9 from baseline. The improvement comes from the fact that our analysis is flow-sensitive between loads and stores. Four false positive pairs are reported by the baseline where there's actually no control path from store instructions to the corresponding loads.

Also, precision of our analysis is more than flow-sensitivity. Suppose we have a function `seq` that stores 41 into pointer `p` and instantly overwrites it with 42. Our analysis reports 4 load-store RAW pairs from the IR while baseline yields 5. Our analysis recognizes the strong-update behavior and eliminated the dependence edge from assignment of 41 to dereference at the return statement. In `alt`, however, same 6 cases are reported from both ours and the baseline. Across the inter-procedural boundary, because both `seq` and `alt` always write into the pointer `p`, a strong-update in function `foo` could also be identified from the consecutive calls into pointer.
Table 6.1. Test Programs

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>kill</th>
<th>cat</th>
<th>cp</th>
<th>ls</th>
<th>touch</th>
<th>mcf</th>
<th>nginx</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bitcode size</td>
<td>40.6KB</td>
<td>42.6KB</td>
<td>165KB</td>
<td>206KB</td>
<td>508KB</td>
<td>28.0KB</td>
<td>1.74MB</td>
</tr>
</tbody>
</table>

Table 6.2. Analysis Precision Comparison

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>kill</th>
<th>cat</th>
<th>cp</th>
<th>ls</th>
<th>touch</th>
<th>mcf</th>
<th>nginx</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline load-store</td>
<td>6839</td>
<td>9011</td>
<td>29779</td>
<td>68988</td>
<td>1042709</td>
<td>5696</td>
<td>1914253</td>
</tr>
<tr>
<td>Ours load-store</td>
<td>1755</td>
<td>2059</td>
<td>6876</td>
<td>11266</td>
<td>119103</td>
<td>2140</td>
<td>110892</td>
</tr>
<tr>
<td>Ours load-arg</td>
<td>149</td>
<td>152</td>
<td>697</td>
<td>1150</td>
<td>991</td>
<td>179</td>
<td>8832</td>
</tr>
<tr>
<td>Ours load-call</td>
<td>22</td>
<td>48</td>
<td>1203</td>
<td>693</td>
<td>18314</td>
<td>76</td>
<td>12676</td>
</tr>
</tbody>
</table>

manipulating subroutines.

6.3 Real-world Experiments

6.3.1 Precision

Besides crafted examples, we have also run our analysis on a number of real-world programs in various sizes. In Table 6.2 are the precision statistics on ours and the baseline, where a significant improvement in the numbers of load-store RAW pairs is displayed. The advantage is especially notable in huge functions with tens of thousands of instructions and sophisticated control flows.

6.3.2 Time

In Table 6.3, we compared the analysis run-time with the baseline. There is a 1.2x to 24x slow-down depending on what program we are analyzing. While time consumption of both analysis co-relates to the bitcode size, we see a faster growth in the baseline. Note in analysis of nginx, slow-down ratio comes to a mere 3x. The time consumption of our analysis is generally linear to the number of functions and possesses high variance among functions of different scales and levels of complexities. In Figure 6.1, only a number of functions contribute to a
Table 6.3. Analysis Run-time Comparison

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>kill ms</th>
<th>cat ms</th>
<th>cp s</th>
<th>ls s</th>
<th>touch s</th>
<th>mcf ms</th>
<th>nginx s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline time</td>
<td>163</td>
<td>224</td>
<td>2.70</td>
<td>4.36</td>
<td>140</td>
<td>327</td>
<td>230</td>
</tr>
<tr>
<td>Ours time</td>
<td>3.96</td>
<td>4.41</td>
<td>18.0</td>
<td>25.6</td>
<td>174</td>
<td>2.43</td>
<td>11.4min</td>
</tr>
</tbody>
</table>

Figure 6.1. Analysis run time distribution of functions in Nginx

significant performance penalty. They are in most cases huge in terms of number of instructions and recursive.

6.3.3 Memory Consumption

A summary-based analysis allows us to dispose a summary after all callers are visited in the call graph, because it is no longer used for the subsequent analysis. As a result, the memory consumption does not grow linearly in the size of analyzed program, but highly dependent on how the call graph is structured. In most of the cases we run, our analysis could easily fit into main memory of modern computer systems, as shown in Table 6.4.

Table 6.4. Analysis Memory Consumption

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>kill MB</th>
<th>cat MB</th>
<th>cp MB</th>
<th>ls MB</th>
<th>touch MB</th>
<th>mcf MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ours memory</td>
<td>76</td>
<td>76</td>
<td>135</td>
<td>163</td>
<td>158</td>
<td>12</td>
</tr>
</tbody>
</table>

35
### 6.4 Case Study

In this section, we should have a closer look on the statistics of the `send_signal` function in GNU coreutils. It is a function of 69 LLVM IR instructions and 12 basic blocks. There’s a total 19 loads and 11 stores. The data dependence statistics are in the Table 6.5. Our analysis cut the reported RAW numbers almost by half. There’s three major types of false positive RAW edges that are eliminated in our analysis:

- **Control-flow-error**: No control flow detected from store to load
- **Alias-error**: False positive may-alias report
- **Library-assumption**: As our analysis is summary-based, we have some manual annotation for called library functions

The most interesting part is Alias-error. In the body of `send_signal`, there’s a number of calls into `__errno_location` function to retrieve a pointer of global variable `errno` in C’s run-time library. Our baseline reported many false positive RAW pairs where alias analysis considers the returned pointer alias with a pointer to local stack memory, which obviously never escapes from the scope of the current function.
Chapter 7

Future Work

In our discussion of data dependence analysis, we already mentioned the idea of the two-stage analysis. By performing a prior cheap non-symbolic analysis, we collect information about the entire program to assist the following complicated analysis. In addition to determining if an abstract location may/must be read/write in a function, we could narrow down target function candidates from indirect calls and add contexts to filter impossible aliasing environments. Development of such a prior analysis could have a remarkable benefit on both precision and performance of our data dependence analysis.
Chapter 8

Conclusion

This research provides a precise summary-based symbolic data dependence analysis based on abstract interpretation and a theorem prover. We implemented it targeting C programs in LLVM IR, evaluated its precision, time and memory usage and compared it with a baseline data dependence analysis. Our analysis shows a promising result as it is significantly more precise in a number of aspects while retaining a practical performance and memory consumption. We could eliminate many false positive RAW relation reports and thus improve the precision of downstream analysis, which could help us to better reason about our programs.
Bibliography


