AN EMPIRICAL STUDY OF ALIAS ANALYSIS TECHNIQUES

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Andrew Tran

June 2018

COMMITTEE MEMBERSHIP

TITLE:	An Empirical Study of Alias Analysis Techniques
AUTHOR:	Andrew Tran
DATE SUBMITTED:	June 2018
COMMITTEE CHAIR:	Aaron Keen, Ph.D. Professor of Computer Science
COMMITTEE MEMBER:	John Clements, Ph.D. Professor of Computer Science
COMMITTEE MEMBER:	Christopher Lupo, Ph.D. Professor of Computer Science

ABSTRACT

An Empirical Study of Alias Analysis Techniques

Andrew Tran

As software projects become larger and more complex, software optimization at that scale is only feasible through automated means. One such component of software optimization is alias analysis, which attempts to determine which variables in a program refer to the same area in memory, and is used to relocate instructions to improve performance without interfering with program execution. Several alias analyses have been proposed over the past few decades, with varying degrees of precision and time and space complexity, but few studies have been conducted to compare these techniques with one another, nor to measure with program data to confirm their accuracy. Normally, this is out of the scope of alias analyses because these processes are static, and can only rely upon the input source code. We address these limitations by instrumenting several benchmarks and combining their data with commonly used alias analyses to objectively measure the accuracy of those analyses. Additionally, we also gather additional program statistics to further determine which programs are the most suitable for evaluating subsequent alias analysis techniques.

ACKNOWLEDGMENTS

Thanks to:

- Aaron Keen, for providing excellent guidance and feedback throughout this Thesis, and being one of the best faculty members I've met at Cal Poly
- The Computer Science Department of Cal Poly, for the continued focus on student success and knowledge
- My parents, for unquestionably believing in my self-efficacy against all odds
- Alanna Buss, for providing continued advice and encouragement on surviving Graduate School
- Michelle Lam, for continued confidence since Undergraduate days
- Vincy Chow, for providing support sorely needed during the roughest spans of my Graduate career
- Andrew Guenther, for uploading this template

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

Programming has come a very long way from writing basic binary instructions. Over time, the emergence of different programming languages has widened the availability and range of applications of possible software projects. One of the most critical advances provided by newer programming languages is the level of abstraction they offer. Different features of programming languages, such as data types and garbage collection, have helped programmers move past machine-specific details to focus on more complex problems. Although software development has made significant advances over the past several decades, improved performance continues to be one of the chief concerns of both software producers and consumers. Principal concerns regarding performance include program speed and efficient resource usage, such as memory and I/O devices, and are inherent to all programming languages regardless of abstraction level.

Today, large software projects can be expected to contain at least millions of lines of code[MGSB12], written by different developers in relatively isolated settings. Such large codebases provide numerous opportunities for software optimization; although individual developers can attempt to optimize certain sections of the software by hand, this approach is infeasible on a larger scale. Additionally, software development often makes tradeoffs between performance and readability; for certain software teams, code readability may be more useful in some situations than pure efficiency. Thus, the only practical approach for optimizing such large programs is through an automated process, such as an optimization pass through a compiler. One such component of software optimization is alias analysis, which attempts to determine which variables in a program refer to the same area in memory; this is used to move instructions in

a way that improve performance without interfering with program execution.

Several alias analyses have been proposed over the past few decades, having varying degrees of precision and time and space complexity. However, few studies have been conducted to compare these techniques with one another, nor to measure with program data to confirm their accuracy. Normally, this is out of the scope of alias analyses because these processes are static, and can only rely upon the input source code. This thesis addresses the limitations of previous studies by examining data from several benchmarks and comparing this data to commonly used alias analyses to objectively measure their accuracy. Additionally, we also gather additional program statistics to further determine which programs are the most suitable for evaluating subsequent alias analysis techniques.

Chapter 2

BACKGROUND

2.1 Control Flow Graphs

When compilers convert a source language to the underlying machine code, they first organize the program's statements into a form that is useful for subsequent operations. The compiler constructs a Control Flow Graph (CFG) for each function that separates groups of statements based on the language's available control flow constructs, such as conditional statements or loops. Statements are grouped into basic blocks, and are connected to one another based on their corresponding control flow statements; larger blocks are encouraged to provide more opportunities for program optimizations. After each graph is generated, the compiler converts the statements from each block into the appropriate machine code and outputs each converted block.

2.2 Intermediate Representations

Some compilers use an Intermediate Representation (IR) for the source language before converting the input program to the appropriate machine code. The IR provides additional information, such as data types, at a lower abstraction level than the input language, and can be processed more easily than the final machine code. Optimizations are often performed after the program is converted to an IR due to having more opportunities to optimize at this level without having to address platform-specific details.

2.2.1 LLVM

The Low Level Virtual Machine (LLVM) IR is commonly used for compiler construction. This IR features instructions similar to those of assembly languages, but also includes features available in higher-level languages. Additional abstractions provided by LLVM include virtual registers, register and value types, and function headers and calls, removing the overhead needed to maintain calling conventions. Each virtual register in the LLVM IR is unique and can only be defined once, a convention known as Single Static Assignment (SSA). LLVM's virtual registers are later mapped to real registers when the program is converted to binary code.

2.3 Optimizations

After creating the CFG for the input program and producing the corresponding IR, a compiler may take one or more optimization passes on the graph. These optimizations are meant to improve program performance without affecting the semantics of the program, and focus on reducing unnecessary code, execution time, and memory usage. CFG's can also keep track of other information that is useful for later optimizations, such as each basic block's predecessors.

2.3.1 SSA Optimization

Because loading and storing variables from memory can incur time overhead, some compilers minimize the use of memory by storing variables exclusively within registers. This is effective with an IR that enforces SSA because whenever a value is updated, including ones from variables, that value must be assigned to a new virtual register. Optimizing programs to follow SSA form requires recursively searching through a basic block's predecessors to find the last register that contained a desired value.

2.3.2 Constant Propagation

Certain constants may be known at compilation time within a program. A compiler can replace operands within statements and expressions with known constants, potentially collapsing multiple expressions into single values. Conditions that are replaced with constants may change the structure of the CFG by removing basic blocks that are never traversed. By simplifying the structure of the CFG without changing the program's meaning, this optimization reduces potential ambiguities caused by unnecessary branches, which is useful for code generation and subsequent program analyses and optimizations.

2.3.3 Code Removal and Relocation

Instructions that do not affect other instructions or have no effect on the program can be removed. Instructions that produce the same result within loops or conditional statements may be relocated to surrounding basic blocks to reduce the amount of redundant calculation. Certain instructions, such as load and store instructions, may be relocated to improve performance based on hardware-based considerations, such as pipelining. When relocating instructions, additional analysis is required to ensure that these instructions do not have additional dependencies from nearby instructions, either in the form of operands in later instructions or by updating required variables or values. One such analysis that determines whether certain values are related is Alias Analysis.

2.4 Alias Analysis

Two pointers are said to alias if they refer to the same area of program memory. An alias analysis attempts to determine which pointers in a program are aliases. Because

alias analysis is an undecidable problem [Ram94], conventional alias analysis techniques perform some kind of approximation when producing sets of possible aliases. Because the analysis is imperfect, compilers must make conservative assumptions when performing optimizations based on the results of an alias analysis. Alias analyses vary in terms of how effective they can examine programs; the deeper a program can analyze, the more complex it is. Some analyses are intraprocedural, and are limited to analyzing single functions, while other analyses are interprocedural and can analyze entire programs. An alias analysis is flow-sensitive if it accounts for changes in aliasing caused by program flow, such as conditional statements or loops, and is context-sensitive if it accounts for aliases that exist between function calls.

Chapter 3

RELATED WORK

Much work has been done in the area of alias analysis, both in proposing new alias analysis techniques, and for evaluating such techniques. Because alias analysis is an undecidable problem [Ram94], many potential avenues exist for developing more precise or efficient approximations.

3.1 Current Alias Analyses

Several existing alias analyses are currently used as part of contemporary compiler optimizations. These analyses take different approaches in identifying aliases, and thus vary in terms of efficiency and effectiveness. We are interested in the alias analysis implementations that target LLVM instructions. The following example C program is used to demonstrate some of the differences between the following alias analyses.

3.1.1 Andersen Analysis

Andersen's Alias Analysis [And94] is an analysis technique for determining pointer aliases within functions without considering program flow. Andersen analysis provides set notation and type inference rules meant for the C programming language. Pointers are initially stated to be part of specific types of pointers, such as global variables, dynamically allocated memory, and function parameters. Additional type inference rules are used to represent different operations performed with pointers, such as dereferencing, assignment, and type casting. These rules are used to generate set constraints for the pointer values within a function. For alias analysis across

```
1  void foo() {            Andersen                Steensgaard
2     int a, b, c;
3     int *x, *y, *z;
4
5     x = &a;              x : {a}                 x : {a}
6     y = &b;              y : {b}                 y : {b}
7     z = y;               z : {b}                 z : {b}
8     c = 0;
9     *x = c;
10
11    if (c > 1) {
12       y = z - 4;        y : {b, unknown}        y : {b}
13    }
14    else {
15       y = x - &a;       y : {b, unknown}        y : {a, b}
16    }
17
18    z = x;               z : {a, b}              z : {a, b}
19 }
```

Figure 3.1: Example Program for Evaluating Alias Analyses

function calls, Andersen analysis uses static call graphs and additional inference rules to generate context-sensitive constraints. The generated constraints for the aliases are solved by using a set of rewriting rules for type normalization and propagation; the constraint solving algorithm's time complexity is polynomial in terms of program size.

In the example program in Figure 3.1, constraints are generated for the statements on lines 5, 6, 7, 12, 15, and 18, which all involve assignment of pointer values. After resolving these constraints, `x` is found to reference `a`, `y` is found to reference `b` and an unknown pointer, and `z` is found to reference `a` and `b`. The conditional statement only collects constraints within its then and else clauses, and as statements are processed, possible aliases for pointers are only added; this can result in some inaccuracies, such as with analyzing `y` and `z`. Additionally, expressions with binary operators, such as

those in lines 12 and 15, are dependent on functions that generate constraints based on the operator and the operands; for the first statement, the constraint refers to an unknown pointer due to the operator subtracting an integer from a pointer to a single variable, leading into an unknown area, and for the second statement, the constraint refers to an unknown pointer because the difference of two pointers is considered an integer.

3.1.2 Steensgaard Analysis

Steensgaard Alias Analysis [Ste96] is another alias analysis technique that works across function calls without considering program flow. Steensgaard analysis is based off an abstract pointer-based language that includes pointer operations, n-ary operators, dynamic memory, and functions. Type inference rules are used to generate alias sets for the pointer variables in the program. Each statement is initially processed once to generate the initial set of pointers; these values are stored in a union-find data structure, and are combined in alias sets in subsequent join operations. The resulting algorithm was found to be linear in space complexity and almost linear in terms of time complexity.

In the example program in Figure 3.1, the types for `a`, `b`, `c`, `x`, `y`, and `z` are discovered as distinct values in the first pass. After processing each statement once in the second pass of the algorithm, `x` refers to `a`, `y` refers to `a` and `b`, and `z` refers to `a` and `b`. Thus, all three pointer variables are found to alias under this analysis. Program flow is not captured well in this analysis, as possible aliases are always added for pointers instead of changed; in this example, `z` should only refer to `a` at the end of the program, and `y` only has a defined alias if the statically constant value for `c` is accounted for, which is usually not the case for an alias analysis. Additionally, for this analysis, the operands of a primitive operation, such as addition and subtraction,

have the same type as the destination variable. This also results in inaccuracies, as shown in lines 12 and 15.

3.1.3 LLVM Basic Alias Analysis

The LLVM infrastructure features a basic alias analysis implementation available for use with compiler implementations [llv]. This alias analysis is local per function and depends on a series of heuristics to determine which pointers alias. For this analysis, distinct global variables, local variable declarations, and heap memory can never alias. Additionally, such values never alias the null pointer [llv]. Similarly, differing structure fields and array references that are statically different do not alias. Some C standard library functions are assumed to either never access program memory, or only access read-only memory. Pointers that refer to constant global values, such as strings, are said to point to constant memory. Finally, function calls cannot access local variables that never escape from the function that allocates them.

3.1.4 Automatic Reference Counting

Originally developed for the Objective-C programming language, Automatic Reference Counting (ARC) is a system of keeping track of allocated objects within a program [ARC]. Dynamically allocated objects are given a reference count and a class based on its ownership, such as strong or weak ownership. To prevent memory leaks or accidental deallocations, objects are retained to add owners, and released to remove owners; objects with no owners are deallocated, and their pointers are set to null. Operations that refer to object pointers, such as reads, writes, initialization, destruction, and moving, are given different rules depending on the object's ownership type. While objects do not exist in C, ARC-based mechanisms can be applied to track pointer references, and are used in the LLVM infrastructure as part of an

alias analysis that can be used for program optimizations.

3.2 Evaluating Alias Analyses

The nearest analogue to this thesis's work can be found in Michael Hind and Anthony Pioli's research report [HP97], which attempts to measure several different alias analysis techniques under the same conditions and performance metrics. Specifically, Hind and Pioli explore three different techniques with varying degrees of precision and efficiency: Flow Insensitive Analysis, Flow Sensitive Analysis, and Flow Insensitive Analysis with Kill Information. These techniques are performed on input programs that are broken down into Control Flow Graphs (CFGs), and sets of pointer aliases are calculated at varying degrees of granularity. The Flow Insensitive analysis calculates possible aliases for variables across the entire function, with the Flow Insensitive Analysis with Kill including additional information about pointer definition and usage intended to improve the Flow Insensitive Analysis. On the other hand, the Flow Sensitive Analysis creates two alias sets for each CFG node, reflecting possible changes in the program due to control flow constructs. All three alias analyses are run on fourteen benchmark C programs, and precision is defined as the number of possible objects, or values, that a given pointer could refer to. Additional statistics are also collected from running these benchmarks, including the execution time of each analysis technique, distinctions between pointers used for reading or writing, and the type of pointer within the context of the program, such as local variables, global variables, formal parameters, and heap variables.

After running the benchmarks, the authors found that additional kill information did not improve the precision of Flow Insensitive analysis. The authors also found that the Flow Insensitive analysis was at least as precise, if not more so, than the Flow Sensitive analysis in half of the benchmarks used. The authors attribute this

discrepancy to three possible causes: the first is that Flow Sensitive analysis becomes less precise as the size of a CFG increases, the second is that the consideration of formal and actual parameters is the same for Flow Sensitive and Flow Insenstive analyses, and the third is that pointers are often not modified in ways that would require the additional overhead needed for Flow Sensitive analysis. The authors also propose efficiency improvements for alias analysis techniques, namely sharing alias sets between CFG nodes using Sparse Evaluation Graph (SEG) nodes to save space and reduce overhead in traversing the CFG, using sorted worklists to traverse CFG nodes, and only propagating alias relations that can be reached from a given function call. All of these improvements were shown to speed up the alias analyses in varying degrees. While this report does provide a detailed method of evaluating alias analysis techniques, its definition of precision is limited by the static nature of alias analysis techniques. Thus, there is no additional confirmation on whether a given alias is accurate with respect to the actual program.

Hind and Pioli produced another report evaluating various alias analysis techniques that expands on their previous work [HP01]. This time, they examined six different context-insensitive analysis techniques. Four of these are flow-insensitive, one is flow-sensitive, and one is flow-insensitive but with additional kill information. The first technique, Address Taken (AT) analysis, is flow insensitive and computes a single global alias set for all objects in the program that were assigned to another variable. With its linear time complexity and limited precision, AT served as a baseline technique for comparison with the other techniques. The next technique, Steensgaard (ST) analysis, is a flow-insensitive analysis that computes a single union/find alias set in a single pass in almost linear time. The next flow-insensitive technique, Andersen (AN) analysis, implements Andersen's algorithm; normally, this algorithm uses constraint solving, but in the interest of efficiency, the analysis technique used in this report uses an iterative dataflow. Two flow-insensitive techniques proposed by Burke

et al. (B1 and B2) calculate local alias sets for each function call, with B2 including additional kill information for variable definition and usage. The final flow-sensitive algorithm proposed by Choi et al. (CH) operates similarly to the B1 technique, but at the level of SEG nodes instead of at the function level.

For this report, Hind and Pioli used twenty-four benchmark C programs, varying from under 1000 to almost 30,000 lines of code. These benchmarks themselves are compared according to the resulting CFG's that are created for the pointer analyses by measuring the number of CFG nodes, the number of function calls, and the number of heap allocations. As with the previous report, precision for each analysis technique is defined by the number of possible aliases for each given pointer. In addition to precision, execution time, memory usage, and the number of pointer reads and writes are measured for each analysis technique.

After running the benchmarks, both AT and ST were found to be efficient in terms of speed and memory usage. ST was significantly more precise than AT, especially as programs increased in size, with only minimal increases in overhead. AN and B1 varied in comparison to each other, with one significantly outperforming the other, and vice versa, in different benchmarks. The B2 analysis was consistently slower than B1, and the CH analysis was generally significantly slower, save for some benchmarks. The AN, B1, and B2 analyses had the same level of precision as one another, and were comparable with the CH analysis for many of the benchmarks. Thus, the additional kill information in B2 was again, not found to provide any significant benefit in increasing precision. As with the previous report, the benchmarks did not encounter the types of statements that would benefit from flow-sensitive analysis. Additionally, the CH analysis's memory usage was several times higher than that of its flow-insensitive counterparts, even after additional optimizations were implemented to reduce its memory footprint. The speed of pointer analysis was found to be dependent on both program size and the number of propagated alias relations throughout

a program's graph. Because this report is an expansion on a previous experiment, it also possesses the same limitations as the previous experiments, namely the definition of precision being limited by purely static analysis techniques.

3.3 Proposed Analysis Techniques

Much of the overhead behind inclusion-based pointer analysis is due to the size of the generated constraint graph describing the relationship between pointer aliases, thus Hardekopf and Lin [HL11] proposed two new algorithms to detect cycles within constraint graphs to reduce the size of the graph. The first method, Lazy Cycle Detection (LCD), occurs when an alias set is propagated across nodes in the constraint graph; LCD checks the constraint graph for cycles based on two conditions; the first is whether or not two alias sets are identical, and the second is whether or not the graph edge related to the current pointer relation was searched previously. The second method, Hybrid Cycle Detection (HCD) performs a static analysis of the program before the actual pointer analysis to create a constraint graph and collapse any possible cycles; this preprocessing reduces the number of traverals performed by the actual pointer analysis, and provides additional information about which pointers might be part of a cycle, even if its alias sets are incomplete after performing HCD. LCD and HCD are evaluated with other comparative optimization algorithms for inclusion-based pointer analysis in five C benchmarks. In addition to the overall reduced number of constraints for each benchmark, the execution time and memory usage is measured for each algorithm. For the benchmarks, HCD is measured both by itself and in combination with the other algorithms. As individual algorithms, LCD and HCD had execution times comparable to the other algorithms. However, though LCD's memory usage was on par with the other algorithms, due to its preprocessing nature, HCD by itself could not complete all of the benchmarks before running out of

memory. When used in tandem, LCD and HCD significantly outperformed the other algorithms in terms of speed, with minimal decreases in memory usage. HCD also provided similar performance improvements when used in conjunction with the other algorithms.

One of the alias analysis techniques used in Hardekopf and Lin's experiments to compare against their proposed algorithms was a context-insensitive pointer analysis method developed by Pearce et al. [PKH04] to account for fields and function pointers in an efficient, precise manner. Previously, context-insensitive pointer analyses lacked the constraint types necessary to accurately reflect references to fields within user-defined structures; aggregate types were generally treated as a single variable, or treated as a distinct set of fields for either a unique instance of an aggregate or for all aggregates of the same type. Additionally, function pointers lacked any particular notation that could be used in an elegant or efficient manner. To account for these shortcomings, the authors introduced pointer constraints that included integer offsets, along with inference rules that utilize these constraints. These offsets can be used to model aggregate fields, and functions based on their addresses and parameters, and are treated as edge weights in a constraint graph. After running both field-sensitive and field-insensitive versions of their new analysis technique on seven benchmarks, Pearce et al. found that field-sensitive analysis offered more precision, but at the cost of increased execution time. However, the increase in precision was also found to decrease with larger programs.

Chapter 4

IMPLEMENTATION

4.1 LLVM Instrumentation

To measure the effectiveness of various alias analyses, we instrument a series of C programs to obtain data about which memory addresses are accessed. We perform this instrumentation on LLVM source files translated from the original C source code to allow finer granularity of instrumentation; even without explicit pointer operations or variable assignment, individual expressions or statements may contain multiple memory accesses that are not easily captured at the level of the target language. At the LLVM level, pointer values are also stored within virtual registers alongside variables, providing additional aliases to quantify. For program instrumentation, we are primarily concerned with three types of instructions: load, store, and getelementptr. For these instructions, we output the referenced memory address, along with additional information that we use to find the original virtual registers within the corresponding LLVM file.

4.1.1 load instructions

The second operand of a load instruction is the pointer where the desired value is stored; load instructions are generated for all variable references in the input program, along with other pointer operations. This operand can be a global or local variable pointer, or another virtual register. Whenever a load instruction is found, its pointer operand is printed as a hexadecimal value to standard output. Because virtual registers only exist at compilation time in the LLVM file, the instrumented code also outputs the file name and the line number of the corresponding load instruction. This

information is used to map the original operand to the actual pointer values retrieved from running the instrumented code.

4.1.2 store instructions

The second operand of a store instruction is the pointer of the value to be updated. As expected, store instructions are generated from variable assignments and spills in the input program, since these statements update values that are stored in memory. As with load instructions, whenever a store instruction is encountered, the instrumented program prints the memory address, file name, and line number of that instruction to standard output. Only this information is necessary for instrumentation, as alias analysis does not distinguish between loads from and stores to memory addresses.

4.1.3 getelementptr instructions

The getelementptr instruction is used to compute pointers for specific fields contained within a structure, and elements of an array. When structure field references are converted from C to LLVM, the field names are converted to integer offsets based on the order of the fields within the corresponding struct type declaration; the result of the getelementptr instruction is subsequently used as part of a load or store instruction, depending on whether or not that structure field is being read from or written to. The first operand of the getelementptr instruction is the pointer to the original structure or array, and is instrumented in the same way as the load and store instructions; this distinction is made to measure memory accesses to compound types, compared to loads and stores from single variables. This consists of printing the base memory address, file name, and line number to standard output. The computed memory address is not included, as it is instrumented in later memory access instructions that reference it.

4.1.4 memory allocation instructions

To gather data on the sizes and lifetimes of dynamically allocated memory, we instrument calls to malloc, realloc, and free. For calls to malloc and realloc, the instrumented code prints the return value of the call as a memory address, along with the file name, line number, size, and timestamp to standard output; we chose to omit instrumenting calls to calloc, as they were not present in a majority of our examined programs in any meaningful frequency. Because dynamic memory can be allocated and freed within a short period of time, often less than a second, the timestamps are taken from calls to the C standard library function clock. Whenever a call to free allocated memory is found within the program, the instrumented code prints the passed in memory address, the file name, the line number, and the timestamp.

4.1.5 Non-exhaustive instrumentation

When instrumenting the LLVM files, we are primarily concerned with pointer operands that are either variables or virtual registers; alias analyses at the LLVM level tend to focus on such operands as well. However, depending on the generated LLVM IR, some irregular operands are also instrumented. Such operands include, but are not limited to, nested getelementptr instructions, vectorized pointer types, and temporary structure types. While the program instrumentation does account for some of these operands as pointers identified within the program, due to the somewhat flexible nature of the IR, the instrumentation is not exhaustive in identifying such operands; any irregular operands that are not detected are not representative of the operands we are concerned with.

Chapter 5

METHODOLOGY

To gather data for alias analyses, we instrument a series of benchmark C programs. These programs vary in size and complexity, and are designed to represent various realistic workloads. The C source files are converted to the LLVM IR and compiled using clang, without optimizations. Optimizing the benchmarks could potentially remove instructions with aliases that we would like to measure. Because the instrumentation requires additional I/O and time overhead, the benchmarks are run either until completion or until 10 minutes have passed. The data from the benchmarks is used with the results from several implemented alias analyses to measure the effectiveness of those techniques.

5.1 SPEC2000 Benchmarks

Seven of the benchmarks used to gather data come from the SPEC2000 benchmark suite [spe]. These benchmarks are designed to test a platform's various CPU, memory, and I/O capabilities, and consist of multiple C source files, and reference input data with expected outputs. The original benchmarks were compiled with GCC using the -O3 optimization flag, which is omitted when converting their respective source files to the LLVM IR. Because of the large amount of code in these benchmarks, we only instrument functions with two or more pointer parameters. We use this requirement to examine functions where we are more likely to see meaningful aliasing, and to reduce the amount of instrumentation data that is not meaningful for evaluating alias analyses; we expect functions that take in multiple pointers to reference and utilize them in more elaborate ways, potentially resulting in more assumed aliases.

Some benchmarks from the SPEC2000 suite, most notably the GCC benchmark, were not included because they could not be converted to the LLVM IR for instrumentation. These benchmarks were implemented in versions of C that were not supported by clang. Other benchmarks were not tested due to missing runtime dependencies, such as for the perlbmk benchmark or data formatting issues, as in the vortex benchmark.

5.2 PLB Benchmarks

We also instrument three benchmarks from the Programming Language Benchmark (PLB) Suite [plb]. While these benchmarks are used to primarily measure performance differences between programming languages, we are interested in the C implementations of these benchmarks for gathering data about aliases. Specifically, we use the sudoku, matmul, and dict benchmarks from the PLB suite. Due to the two-dimensional nature of Sudoku puzzles and matrix multiplication, we expect meaningful amounts of aliasing to occur within the program. We also expect similar amounts of aliasing for the dict benchmark due to the nature of creating, updating, and traversing dictionaries. In terms of code size, the PLB benchmarks are smaller, consisting of single source files for each benchmark, and are expected to complete before 10 minutes have elapsed. Unlike the SPEC2000 benchmarks, all functions in these benchmarks are instrumented.

5.3 Other Benchmarks

We include three benchmarks that are meant to test implementations of the standard library malloc function [mal]. While the function calls differ, each of the benchmarks repeatedly call malloc and free. These benchmarks are also small, so all functions for each benchmark are instrumented. Two small customized benchmarks were also

written for testing alias analyses, and are instrumented at the same level as the other smaller benchmarks; one is a search tree that stores words based on common prefixes, while the other searches for cycles within a linked list at different rates.

5.4 Alias Analysis

We gather alias data from four alias analyses implemented by the LLVM optimizer, consisting of the Andersen [And94], Steensgaard [Ste96], Objective-C based Automatic Reference Counting (ARC), and Basic [llv] Alias Analyses. The optimizer outputs information for each LLVM file in the form of alias sets, each containing virtual registers or variables that must, or may alias with each other. We omitted the measurement of more recent alias analyses due to the additional time required to implement more complex analyses at the LLVM level.

Chapter 6

VALIDATION

As an empirical study, we gather statistics from the instrumented programs to gather information about their memory access patterns. We organize the runtime data we retrieve and combine it with our alias analyses to produce meaningful metrics for each of the benchmarks.

6.1 Alias Misidentification

Because alias analyses are static, their accuracy cannot be confirmed until the program is run. As each instrumented program runs, we link each outputted memory address back to its original instruction operand based on the file name and line number. By mapping the outputted memory addresses back to the original operands in their respective LLVM files, we can confirm whether or not two pointers alias. We define an alias miss as two operands that are stated by an alias analyses to alias, but have differing memory addresses at runtime.

At the beginning of each program run, all of the operands found by the alias analyses are considered to be unknown. Depending on how many instructions have been executed, some operands may not have a memory address assigned to them, even though they may have been stated to alias with other known operands. To address this, we also consider two reported aliases to miss if one of those aliases has an unknown address at runtime. Thus, the alias miss rate is defined as the number of alias misses over the total number of aliases found.

6.2 Pointer Lifetime

To better examine memory access patterns, we want to measure how long any given operands in a program refer to the same area of memory. We define pointer lifetime as how long, in terms of instrumented instructions executed, an operand, such as a virtual register, refers to the same memory address before being changed. We expect operands within loops or conditional statements to have shorter lifetimes due to rapid updates and reassignment, while variables used throughout the span of functions are expected to have longer lifetimes.

6.3 Allocation Size and Lifetime

We gather the specified sizes, in terms of bytes, when dynamically allocating memory to get a better sense of what is being allocated over time. More consistent allocation sizes may imply repeated use of data structures, such as linked lists, or types, such as common uses of arrays; conversely, variable allocation sizes may reflect more dynamic uses of memory, such as storing strings. Similarly, we measure allocation lifetime in terms of "ticks" specified by the standard library `clock` function call.

Chapter 7

ANALYSIS

After instrumenting the benchmarks to keep track of memory accesses, we found some interesting trends related to the statistics we chose to measure. While many of these trends were what we expected, others were more surprising and warranted further discussion.

7.1 Pointer Identification

Table 7.1 shows the number of pointer operands reported by alias analyses for each benchmark, the total number of pointers, and the identification rate found for each benchmark. For all the benchmarks, all four alias analyses identified the same number of aliases.

As indicated by Table 7.1, all of the alias analysis techniques identified above 90 percent of the pointer operands within the larger benchmarks, namely benchmarks that consisted of five or more source files. We attribute this to the larger number of virtual registers found within these benchmarks; larger programs have a higher number of memory access instructions with single virtual register operands that can be examined by alias analyses, and despite the increased program complexity, there is a lower proportion of irregular operands that are not identified. The smaller benchmarks we used had more variable identification rates, ranging from about 60 to 75 percent. We expected that none of the benchmarks would have 100 percent pointer identification, where every memory address is mapped to a pointer operand that is accounted for in an alias analysis, due to pointer values within the source programs that did not belong to virtual registers, such as nested getelementptr instructions.

	Identified Pointers	Total Pointers	Identification Rate
bzip2	624	630	0.990
gzip	1170	1204	0.972
mcf	676	733	0.922
twolf	9081	9099	0.998
parser	3221	3243	0.993
vpr	3701	3975	0.931
crafty	4587	4598	0.993
sudoku	77	127	0.606
matmul	41	54	0.759
dict	138	232	0.594
libc_malloc	171	177	0.966
libc_malloc2	171	177	0.966
tcmalloc	171	177	0.966
tree	79	131	0.603
cycles	27	31	0.871

Table 7.1: Pointers Identified, Total Pointers, and Identification Rates

This is because the alias analysis techniques examined primarily focus on virtual registers, instead of irregular operands, so not all memory accesses would be covered by these analyses.

7.2 Alias Misses

Table 7.2 shows the number of alias misses for each benchmark, separated by the type of alias analysis used. When calculating alias misses, we only consider the first memory address assigned to that pointer operand. Thus, if two pointers are said

to alias, and one of these pointers has multiple differing memory addresses over the span of the program, this is considered only one alias miss. Two pointers declared as possible aliases are also considered a miss if they have different memory addresses.

	Anders	Steens	ARC	Basic
bzip2	78	78	79	79
gzip	41	41	41	41
mcf	103	104	104	104
twolf	92	92	92	92
parser	23	23	23	23
vpr	218	218	218	218
crafty	38	38	39	39
sudoku	37	37	37	37
matmul	13	13	13	13
dict	106	106	106	106
libc_malloc	9	9	9	9
libc_malloc2	9	9	9	9
tcmalloc	0	0	0	0
tree	42	42	42	42
cycles	1	1	1	1

Table 7.2: Alias Misses per Benchmark

Most of the benchmarks had relatively low alias miss rates, regardless of the alias analysis techniques. Over all of the instrumented benchmarks, low miss rates ranged from 0 to 7 percent. The number of alias misses was the same across all four analysis techniques for almost all of the benchmarks. Only three benchmarks showed differences in the number of alias misses - bzip2, mcf, and crafty, and for these benchmarks, the differences were still small, typically consisting of one alias

miss between different alias analyses. Otherwise, all four alias analysis techniques tested were shown to be equally effective for most of the instrumented benchmarks.

It's important to remember that the alias analyses are run on functions in LLVM source files, which consist of a one-dimensional array of blocks of statements. High-level control flow statements are translated to block separations, and SSA ensures that updates to existing values produce new virtual registers that may be examined in an alias analysis. Although aliases may still exist across blocks, the amount of ambiguity caused by program flow and stateful updates is reduced at this level of the program. This may explain why different analyses are often equally effective at this level, when they would have more pronounced differences in a more abstract programming language.

7.3 Alias Miss Rates

Table 7.3 shows the alias miss rate for each benchmark, separated by the type of alias analysis used.

Some benchmarks had high alias miss rates. In this case, we considered high miss rates as over 20 percent on average across all alias analyses. These benchmarks include the sudoku, matmul, dict, and tree benchmarks. After examining the original source files for these benchmarks, we provide possible reasons for the unusually high miss rates.

7.3.1 Sudoku

The sudoku benchmark calculates the solution for nine example sudoku problems that are considered difficult for machines to solve by brute force. Across all alias analyses, this benchmark has an average miss rate of about 48 percent. The solving

	Anders	Steens	ARC	Basic
bzip2	0.125	0.125	0.127	0.127
gzip	0.035	0.035	0.035	0.035
mcf	0.152	0.154	0.154	0.154
twolf	0.010	0.010	0.010	0.010
parser	0.007	0.007	0.007	0.007
vpr	0.059	0.059	0.059	0.059
crafty	0.008	0.008	0.009	0.009
sudoku	0.481	0.481	0.481	0.481
matmul	0.317	0.317	0.317	0.317
dict	0.768	0.768	0.768	0.768
libc_malloc	0.053	0.053	0.053	0.053
libc_malloc2	0.053	0.053	0.053	0.053
tcmalloc	0	0	0	0
tree	0.532	0.532	0.532	0.532
cycles	0.037	0.037	0.037	0.037

Table 7.3: Alias Miss Rate per Benchmark

algorithm uses a binary matrix, declared as a two-dimensional array, representing a series of constraints that make up a valid sudoku puzzle, and iteratively tries different sets of constraints until a valid solution is found. As expected, the brute-force algorithm often requires significant backtracking, iteration, and conditional statements related to updating the binary matrix, resulting in numerous basic blocks within the associated LLVM files. Because none of the tested alias analysis techniques are flow-sensitive, they would not be expected to easily identify many of the resulting pointers for possible aliasing within the solution function.

For this benchmark, the function `sd_update` has the highest potential for alias

misses; as the name suggests, this function updates the binary matrix used to represent the sudoku puzzle, and is frequently called throughout the benchmark. The C implementation of this function is shown in Figure 7.1.

The statement at the beginning line 6 of Figure 7.1 updates the state vectors and the binary matrix by iterating through the rows and columns. Referencing the matrix itself requires indirection into a structure and a field reference, followed by multiple indexes into an array. Consequently, this is translated into multiple getelementptr and load instructions to access the appropriate entry in the matrix. Accessing the array in this manner is fairly common, and occurs several times at varying levels of nested loops. The statement translated into the LLVM IR is shown in Figure 7.2.

All of the memory accesses in Figure 7.2 are contained within a loop, and cannot be analyzed well by flow-insensitive analyses. Specifically, the Steensgaard analysis cannot determine the aliases well because each statement is only processed once, which does not properly reflect the iteratve nature of this block; any aliases for the matrix entry to values outside of the loop or inside other loops would miss. As an example, for pointer register `%37`, the Steensgaard Analysis gives registers `%57`, `%79`, `%97`, `%103`, and `%158` as possible aliases, but all of these pointers belong to blocks within different nested loops based on their respective branches, resulting in higher miss rates. Similarly, the Andersen analysis is able to gather constraints for statements within loops, but cannot reflect changes before and after due to iteration within the program. Ultimately, because this iterative access pattern occurs so frequently in this function, it contributes to higher miss rates.

7.3.2 Matrix Multiplication

The matmul benchmark multiplies two randomly-generated 100 by 100 matrices together, and has an average miss rate of about 32 percent. The matrices are dynam-

```
1  // update the state vectors when we pick up choice r;
2  // v=1 for setting choice; v=-1 for reverting
3  static inline int sd_update(const sdaux_t *aux, int8_t sr[729],
4     uint8_t sc[324], int r, int v) {
5     int c2, min = 10, min_c = 0;
6
7     for (c2 = 0; c2 < 4; ++c2)
8        sc[aux->c[r][c2]] += v<<7;
9
10    for (c2 = 0; c2 < 4; ++c2) { // update # available choices
11       int r2, rr, cc2, c = aux->c[r][c2];
12
13       if (v > 0) { // move forward
14          for (r2 = 0; r2 < 9; ++r2) {
15             if (sr[rr = aux->r[c][r2]]++ != 0)
16                continue; // update the row status
17
18             for (cc2 = 0; cc2 < 4; ++cc2) {
19                int cc = aux->c[rr][cc2];
20
21                // update # allowed choices
22                if (--sc[cc] < min)
23                   // register the minimum number
24                   min = sc[cc], min_c = cc;
25             }
26          }
27       } else { // revert
28          const uint16_t *p;
29
30          for (r2 = 0; r2 < 9; ++r2) {
31             if (--sr[rr = aux->r[c][r2]] != 0)
32                continue; // update the row status
33
34             // update the count array
35             p = aux->c[rr];
36             ++sc[p[0]];
37             ++sc[p[1]];
38             ++sc[p[2]];
39             ++sc[p[3]];
40          }
41       }
42    }
43    // return the col that has been modified
44    // and with the minimal available choices
45    return min<<16 | min_c;
46 }
```

Figure 7.1: `sd_update` implementation from the Sudoku Benchmark

ically allocated, providing a large number of potential pointers into the matrices to alias with. The classic multiplication algorithm utilizes three nested loops to calculate

```
1    %24 = load i32, i32* %10, align 4
2    %25 = shl i32 %24, 7
3    %26 = load i8*, i8** %8, align 8
4    %27 = load %struct.sdaux_t*, %struct.sdaux_t** %6, align 8
5    %28 = getelementptr inbounds %struct.sdaux_t, %struct.sdaux_t* %27, i32 0, i32 1
6    %29 = load i32, i32* %9, align 4
7    %30 = sext i32 %29 to i64
8    %31 = getelementptr inbounds [729 x [4 x i16]], [729 x [4 x i16]]* %28, i64 0, i64 %30
9    %32 = load i32, i32* %11, align 4
10   %33 = sext i32 %32 to i64
11   %34 = getelementptr inbounds [4 x i16], [4 x i16]* %31, i64 0, i64 %33
12   %35 = load i16, i16* %34, align 2
13   %36 = zext i16 %35 to i64
14   %37 = getelementptr inbounds i8, i8* %26, i64 %36
15   %38 = load i8, i8* %37, align 1
16   %39 = zext i8 %38 to i32
17   %40 = add nsw i32 %39, %25
18   %41 = trunc i32 %40 to i8
19   store i8 %41, i8* %37, align 1
```

Figure 7.2: LLVM instructions from the Sudoku Benchmark

the resulting matrix. As with the sudoku benchmark, the prominent amount of iteration within this program introduced additional program flow that none of the given alias analyses could reliably address due to treating array elements as aliases with the base array. Compared with the sudoku benchmark, the larger amount of memory used within this benchmark did not result in a higher percentage of alias misses; even though matrix multiplication still requires a large amount of memory for the input matrices, the loop body itself features fewer flow-sensitive statements, resulting in a lower overall miss rate. The C implementation of the matrix multiplication function is shown below.

Within the matrix multiplication function in Figure 7.3, there are two areas of interest: the transposing of the second matrix on lines 6 to 9, and the calculation of each entry on lines 13 to 16. The body of the inner loop on line 9, translated to LLVM IR, is shown below.

Each access into the matrix requires two getelementptr instructions, as on lines 4 and 13, and two load instructions, as on lines 5 and 14, to index twice into the matrix.

```
1  // better cache performance by transposing the second matrix
2  double **mm_mul(int n, double *const *a, double *const *b)
3  {
4     int i, j, k;
5     double **m, **c;
6     m = mm_init(n); c = mm_init(n);
7     for (i = 0; i < n; ++i) // transpose
8        for (j = 0; j < n; ++j)
9           c[i][j] = b[j][i];
10    for (i = 0; i < n; ++i) {
11       double *p = a[i], *q = m[i];
12       for (j = 0; j < n; ++j) {
13          double t = 0.0, *r = c[j];
14          for (k = 0; k < n; ++k)
15             t += p[k] * r[k];
16          q[j] = t;
17       }
18    }
19    mm_destroy(n, c);
20    return m;
21 }
```

Figure 7.3: Matrix Multiplication Function

```
1   %30 = load double**, double*** %6, align 8
2   %31 = load i32, i32* %8, align 4
3   %32 = sext i32 %31 to i64
4   %33 = getelementptr inbounds double*, double** %30, i64 %32
5   %34 = load double*, double** %33, align 8
6   %35 = load i32, i32* %7, align 4
7   %36 = sext i32 %35 to i64
8   %37 = getelementptr inbounds double, double* %34, i64 %36
9   %38 = load double, double* %37, align 8
10  %39 = load double**, double*** %11, align 8
11  %40 = load i32, i32* %7, align 4
12  %41 = sext i32 %40 to i64
13  %42 = getelementptr inbounds double*, double** %39, i64 %41
14  %43 = load double*, double** %42, align 8
15  %44 = load i32, i32* %8, align 4
16  %45 = sext i32 %44 to i64
17  %46 = getelementptr inbounds double, double* %43, i64 %45
18  store double %38, double* %46, align 8
```

Figure 7.4: Transpose Loop translated to LLVM

Additionally, the matrix pointer and the indices used are loaded each time, and are updated after every loop iteration. Combined with the single store instruction for updating the matrix, this block alone has twelve memory access instructions out of

the nineteen total instructions, none of which can be analyzed well by flow-insensitive alias analyses because they are within a doubly-nested for loop. As with the sudoku benchmark, the typing rules for both the Steensgaard and Andersen analyses only process the statements in the loop body once, and thus, cannot properly reflect the changes caused by loop iteration. For this function, the Steensgaard Analysis lists registers `%68`, `%78`, and `%88` as possible aliases of pointer register `%37`, where all of these pointers exist in different loops. The loop body from lines 13 to 16 of Figure 7.3 suffers from similar problems. The translated loop is shown in Figure 7.5, and has a similar concentration of memory access instructions as the previous loop body. Note that register `%78` exists in this loop, leading to mismatches with earlier aliases.

7.3.3 Dictionary

The dict benchmark creates a hash table to store a series of input strings, and the resulting alias miss rate is about 77 percent. The significantly higher miss rate can be attributed to the logic related to updating and maintaining the hash table's entries. As with the Sudoku solving algorithm, the logic related to inserting variable-sized entries into the hash table is implemented as several loops and conditional statements, primarily for traversing the hash table's array buckets and updating various attributes for the hash table. Unlike the Sudoku benchmark, accesses to the hash table are also more variable, and updates to the hash table that resize the hash table may occur, potentially invalidating previous aliases. Thus, hash tables are particularly difficult for the selected alias analysis techniques to effectively characterize. Within the main program loop, each occurrence of a string is inserted into the hash table, as shown in Figure 7.6, using the macros `kh_put`, `kh_val`, and `kh_key` to access the hash table.

When lines 10 to 12 in Figure 7.6 are translated to LLVM, `kh_key` and `kh_val` are treated as array accesses. The arrays of keys and values are fields in the hash table

```
1    %81 = load i32, i32* %9, align 4
2    %82 = load i32, i32* %4, align 4
3    %83 = icmp slt i32 %81, %82
4    br i1 %83, label %84, label %101
5
6  ; <label>:84:                                     ; preds = %80
7    %85 = load double*, double** %12, align 8
8    %86 = load i32, i32* %9, align 4
9    %87 = sext i32 %86 to i64
10   %88 = getelementptr inbounds double, double* %85, i64 %87
11   %89 = load double, double* %88, align 8
12   %90 = load double*, double** %15, align 8
13   %91 = load i32, i32* %9, align 4
14   %92 = sext i32 %91 to i64
15   %93 = getelementptr inbounds double, double* %90, i64 %92
16   %94 = load double, double* %93, align 8
17   %95 = fmul double %89, %94
18   %96 = load double, double* %14, align 8
19   %97 = fadd double %96, %95
20   store double %97, double* %14, align 8
21   br label %98
22
23 ; <label>:98:                                     ; preds = %84
24   %99 = load i32, i32* %9, align 4
25   %100 = add nsw i32 %99, 1
26   store i32 %100, i32* %9, align 4
27   br label %80
28
29 ; <label>:101:                                    ; preds = %80
30   %102 = load double, double* %14, align 8
31   %103 = load double*, double** %13, align 8
32   %104 = load i32, i32* %8, align 4
33   %105 = sext i32 %104 to i64
34   %106 = getelementptr inbounds double, double* %103, i64 %105
35   store double %102, double* %106, align 8
```

Figure 7.5: Inner Loop translated to LLVM

struct, so their offsets are fixed when accessing them from the struct via getelementptr and load instructions. However, the array indices are based on the hash value of the string, which results in aliasing issues caused by unpredictable accesses. The LLVM IR for lines 10 to 12 is shown in Figure 7.7.

The hash value is calculated by the function `kh_put_str`, which replaces the macro `kh_put`. This function utilizes multiple conditional statements and memory accesses into the input string to produce the hash value. Unlike the earlier benchmarks, most

```
1       k = kh_put(str, h, buf, &ret);
2       if (ret) { // absent
3          int l = strlen(buf) + 1;
4          if (block_end + l > BLOCK_SIZE) {
5             ++curr; block_end = 0;
6             mem = realloc(mem, (curr + 1) * sizeof(void*));
7             mem[curr] = malloc(BLOCK_SIZE);
8          }
9          memcpy(mem[curr] + block_end, buf, l);
10         kh_key(h, k) = mem[curr] + block_end;
11         block_end += l;
12         kh_val(h, k) = 1;
13      } else {
14         ++kh_val(h, k);
15         if (kh_val(h, k) > max) max = kh_val(h, k);
16      }
```

Figure 7.6: Dictionary Insertion in C

```
1   %88 = load %struct.kh_str_t*, %struct.kh_str_t** %13, align 8
2   %89 = getelementptr inbounds %struct.kh_str_t, %struct.kh_str_t* %88, i32 0, i32 5
3   %90 = load i8**, i8*** %89, align 8
4   %91 = load i32, i32* %12, align 4
5   %92 = zext i32 %91 to i64
6   %93 = getelementptr inbounds i8*, i8** %90, i64 %92
7   store i8* %87, i8** %93, align 8
8   %94 = load i32, i32* %14, align 4
9   %95 = load i32, i32* %10, align 4
10  %96 = add nsw i32 %95, %94
11  store i32 %96, i32* %10, align 4
12  %97 = load %struct.kh_str_t*, %struct.kh_str_t** %13, align 8
13  %98 = getelementptr inbounds %struct.kh_str_t, %struct.kh_str_t* %97, i32 0, i32 6
14  %99 = load i32*, i32** %98, align 8
15  %100 = load i32, i32* %12, align 4
16  %101 = zext i32 %100 to i64
17  %102 = getelementptr inbounds i32, i32* %99, i64 %101
18  store i32 1, i32* %102, align 4
```

Figure 7.7: Hashing calls translated to LLVM

of the memory accesses refer to the same struct using getelementptr, but with different offsets to produce distinct byte pointers. The inability to distinguish between these varying offsets makes this function difficult to analyze effectively using flow-insensitive alias analyses. One of the blocks illustrating these problems is shown in Figure 7.8. Both the Steensgaard and the Andersen analyses treats accesses to pointer register

%4 with different offsets, such as %110 with an offset of 5, and %280 with an offset of 4, across multiple blocks, as aliases based on the original struct pointer, leading to a high number of misses.

```
%4 = alloca %struct.kh_str_t*, align 8

%109 = load %struct.kh_str_t*, %struct.kh_str_t** %4, align 8
%110 = getelementptr inbounds %struct.kh_str_t, %struct.kh_str_t* %109, i32 0, i32 5
%111 = load i8**, i8*** %110, align 8

%279 = load %struct.kh_str_t*, %struct.kh_str_t** %4, align 8
%280 = getelementptr inbounds %struct.kh_str_t, %struct.kh_str_t* %279, i32 0, i32 4
%281 = load i32*, i32** %280, align 8
```

Figure 7.8: Excerpts from kh_put_str

7.3.4 Tree

The tree benchmark creates a search tree from a series of input strings that has branches based on common prefixes of one or more letters, and traverses this tree to retrieve specific strings requested by the user. This benchmark has a miss rate of about 53 percent, which is related to the iteration required to traverse the tree, along with the dynamic allocation of variable-sized amounts of dynamic memory for each tree's corresponding letters. The Andersen and Steensgaard analyses both identify the tree node pointers in registers %14, %26, %66 and %137 as aliases, but the latter three are pointers that are updated within separate loops. Additionally, byte pointers based off of offsets from the struct pointers, such as registers %83, %117, and %156, are treated incorrectly as aliases, in a way that is similar to the previous dict benchmark.

References to recursive data structures, like trees, are first loaded from the appropriate struct field offsets, and are treated as a separate pointer for subsequent loads and stores. When these fields are repeatedly updated, as in loops, this causes aliasing issues. Excerpts from the function for collapsing tree nodes, which involves such tree

pointer traversal and reassignment, is shown below in both C, and the LLVM IR, illustrating these potential issues.

```
1       /* Merge the contents of cur and the single link */
2       if (link) {
3          temp = cur->size;
4          cur->size += link->size;
5
6          cur->letters = realloc(cur->letters, cur->size);
7
8          /* Copy the characters */
9          for (i = temp; i < cur->size; ++i) {
10            cur->letters[i] = link->letters[i - temp];
11         }
12
13         /* Copy the links */
14         /* Collapse the subtrees */
15         for (i = 0; i < NUM_LETTERS; ++i) {
16            cur->nodes[i] = link->nodes[i];
17            collapseTree(cur->nodes[i]);
18         }
19
20         /* Free the link */
21         free(link->letters);
22         free(link);
23      }
```

Figure 7.9: Excerpt from the tree benchmark in C

7.4 Pointer Lifetimes

Table 7.4 shows the mean and standard deviations of the pointer lifetimes for each benchmark, in terms of the number of instrumented instructions, and are independent of any alias analyses.

The average pointer lifetimes were significantly higher for the larger benchmarks. We suspect that this is due to variables remaining throughout the span of the program, such as input data to be processed, or structs maintaining program state. At the same time, the percentiles of the pointer lifetimes appear to increase quickly as the percentiles increase, reflecting varying access patterns. This suggests that a large number of pointers are being used within a short timespan, likely within loops. The

```
1    %133 = load %struct.treeNode*, %struct.treeNode** %5, align 8
2    %134 = getelementptr inbounds %struct.treeNode, %struct.treeNode* %133, i32 0, i32 0
3    %135 = load i32, i32* %6, align 4
4    %136 = sext i32 %135 to i64
5    %137 = getelementptr inbounds [26 x %struct.treeNode*], [26 x %struct.treeNode*]* %134, i64 0, i64 %136
6    %138 = load %struct.treeNode*, %struct.treeNode** %137, align 8
7    %139 = load %struct.treeNode*, %struct.treeNode** %4, align 8
8    %140 = getelementptr inbounds %struct.treeNode, %struct.treeNode* %139, i32 0, i32 0
9    %141 = load i32, i32* %6, align 4
10   %142 = sext i32 %141 to i64
11   %143 = getelementptr inbounds [26 x %struct.treeNode*], [26 x %struct.treeNode*]* %140, i64 0, i64 %142
12   store %struct.treeNode* %138, %struct.treeNode** %143, align 8
13   %144 = load %struct.treeNode*, %struct.treeNode** %4, align 8
14   %145 = getelementptr inbounds %struct.treeNode, %struct.treeNode* %144, i32 0, i32 0
15   %146 = load i32, i32* %6, align 4
16   %147 = sext i32 %146 to i64
17   %148 = getelementptr inbounds [26 x %struct.treeNode*], [26 x %struct.treeNode*]* %145, i64 0, i64 %147
18   %149 = load %struct.treeNode*, %struct.treeNode** %148, align 8
19   %150 = call %struct.treeNode* @collapseTree(%struct.treeNode* %149)
```

Figure 7.10: Excerpt from the tree benchmark in LLVM

higher percentiles are also much higher than the lower percentiles, implying that a large number of these short range pointers exist, while the averages are skewed toward longer-lived pointers.

	Mean	25th PCTL	50th PCTL	75th PCTL	100th PCTL
bzip2	315009.684	1	1	69390	5537346
gzip	258578.033	1	2.5	1634.25	5714959
mcf	195359.596	1	1	5209	11706529
twolf	70494.051	1	1	2.75	684118
parser	3535.069	1	1	5	92426
vpr	4321.077	1	1	5	258496
crafty	282708.395	2	6	150.5	4852228
sudoku	228052.284	1	8.5	119223	3012733
matmul	176101.094	1	100	10000	5020000
dict	63435.690	2	16	78087	1741124
libc_malloc	76924.026	1	1	1	2000000
libc_malloc2	48781.439	1	1	1	1000000
tcmalloc	86958.217	1	2	3	2000000
tree	25.717	1	1	7.5	1696
cycles	1363.871	1	2	109.5	14712

Table 7.4: Average Pointer Lifetime and Percentiles

Chapter 8

FUTURE WORK

While this thesis is an exploratory study into the effectiveness of alias analyses, there are some limitations that, given more time, could be addressed in subsequent studies. Most of the future work related to this thesis involves elaborating upon various aspects of the study to be more specific at gathering and quantifying data, along with exploring additional alias analyses, programs, and interesting program statistics.

8.1 Additional Alias Analyses

Additional studies could be conducted to measure the effectiveness of other alias analyses. The LLVM Optimizer features several other built-in alias analyses that were not included in this thesis that could be examined. These analyses were omitted from the study to focus on more commonly used alias analyses, as the primary goal of this thesis was to test the effectiveness of using runtime data to determine the efficacy of different alias analyses. Given more time, more recently proposed alias analyses could be implemented to work at the LLVM level for similar measurement.

8.2 Additional Benchmarks

The benchmark programs used in this thesis were selected to represent a diverse range of realistic program workloads. Because of this, the list of chosen benchmarks is inherently non-exhaustive, and cannot reflect all types of memory access patterns we are interested in. We suggest the following guidelines for selecting benchmarks for future studies.

8.2.1 Function Complexity

While benchmarks of varying sizes were selected, organizing these programs into categories based on the complexity of their functions might be useful for observing trends in some of the measured program statistics. Function complexity would be best defined by lines of code, and by number and placement of control flow statements, based on the assumption that more complex programs have similar proportions of memory access instructions to those of smaller programs.

8.2.2 Memory Allocation

Only six benchmarks used in this thesis dynamically allocated memory and were instrumented for this allocation, and some of these benchmarks allocated memory in contrived ways to test the memory allocation functions themselves; this is not representative of most programs that allocate memory, and could be improved in future studies by examining programs that use allocated memory in more realistic ways.

8.3 Improved Instrumentation

The instrumentation for this thesis consists of printing out data for memory access instructions. Because of the I/O overhead associated with printing out information, this is not the most efficient method of instrumentation; when considering the large number of memory access instructions within the unoptimized benchmark programs, the instrumented programs are several orders of magnitude slower than their uninstrumented counterparts. More time could be dedicated to developing or modifying a framework for logging a program's memory accesses at the LLVM level, similar to how debuggers handle programs in isolated environments. A dedicated framework

might also be better at gathering data at a finer level, improving the effectiveness of the gathered program statistics.

8.4 Statistics Gathered

Refining the existing program statistics, along with additional statistics, could help provide further insight into the behavior of instrumented programs.

8.4.1 Timing

Due to limitations on the platform used to test the alias analyses, along with the lower-level nature of the LLVM code, finer-grained timers were unavailable for program instrumentation. One possible improvement for measuring allocation lifetimes in subsequent studies is to use CPU-based timers that measure the number of clocks in a program, but this is dependent on the available libraries on the testing platform.

8.4.2 Local Statistics

The program statistics presented by this thesis are global statistics. However, many programs will have different memory access patterns depending on which functions are running. Generating program statistics per function may help clarify such differences in memory accesses.

Chapter 9

CONCLUSION

To measure the effectiveness of several alias analysis techniques, we instrumented several benchmarks from the SPEC2000 benchmark suite, the PLB benchmark suite, the malloc benchmark suite, and some self-implemented benchmarks, to print information about which memory addresses are accessed. We used this information to measure the accuracy of different alias analyses by mapping the memory addresses to the original operands, and comparing this to the alias sets generated by the analyses. We found that the alias analyses were equally effective for most of the benchmarks used. Of the benchmarks we used, we found the bzip2, crafty, and vpr benchmarks from the SPEC2000 suite, the sudoku and matmul benchmarks from the PLB suite, and the tree benchmark, to be the most effective at testing alias analysis techniques while being representative of real-world memory access patterns. We also explored other program statistics that could provide insights into memory access patterns, namely alias lifetimes, allocation sizes, and allocation lifetimes.

BIBLIOGRAPHY

[And94] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.

[ARC] Objective-C Automatic Reference Counting (ARC). `https://clang.llvm.org/docs/AutomaticReferenceCounting.html`.

[fft] Cal Poly Github. `http://www.github.com/CalPoly`.

[HL11] Ben Hardekopf and Calvin Lin. Flow-sensitive pointer analysis for millions of lines of code. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 289–298, Washington, DC, USA, 2011. IEEE Computer Society.

[HP97] Michael Hind and Anthony Pioli. An empirical comparison of interprocedural pointer alias analyses. Technical report, IBM Research Division and Department of Mathematics and Computer Science, State University of New York at New Paltz, 1997.

[HP01] Michael Hind and Anthony Pioli. Evaluating the effectiveness of pointer alias analyses. *Sci. Comput. Program.*, 39(1):31–55, January 2001.

[llv] Llvm Alias Analysis Infrastructure. `https://llvm.org/docs/AliasAnalysis.html`.

[mal] malloc-benchmarks Github. https://github.com/caglar/malloc-benchmarks.

[MGSB12] J. David Morgenthaler, Misha Gridnev, Raluca Sauciuc, and Sanjay Bhansali. Searching for build debt: Experiences managing technical debt at google. In *Proceedings of the Third International Workshop on Managing Technical Debt*, MTD '12, pages 1–6, Piscataway, NJ, USA, 2012. IEEE Press.

[PKH04] David J. Pearce, Paul H. J. Kelly, and Chris Hankin. Efficient field-sensitive pointer analysis for c. In *Proceedings of the 5th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '04, pages 37–42, New York, NY, USA, 2004. ACM.

[plb] Programming Languages Benchmarks. `https://attractivechaos.github.io/plb/`.

[Ram94] Ganesan Ramalingam. The undecidability of aliasing. *ACM Trans. Program. Lang. Syst.*, 16(5):1467–1471, September 1994.

[spe] Spec Cpu2000 V1.3. `https://www.spec.org/cpu2000/`.

[Ste96] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 32–41, New York, NY, USA, 1996. ACM.