

# Evaluating Alias Analysis methods

Anushka Idamekorala  
(anb5km)  
anb5km@virginia.edu

## ABSTRACT

Pointer Alias Analysis (PAA) is a fundamental technique used in compiler optimizations, program analysis and software testing. This approach decides whether two pointers in a program may point to the same memory location - accurately performing this analysis guarantees precise and efficient execution of programs involving pointer manipulation. Here, I provide a detailed evaluation of pointer alias analysis techniques implemented within LLVM optimizer in terms of their accuracy, precision and effect on other optimizations.

## 1 Motivation

Pointer Alias Analysis is an indispensable technique in compiler optimizations, program analysis, and software testing. It is crucial in identifying whether two pointers in a program may point to the same memory location, which has significant implications for program correctness, performance, security, and correctness. Accurate and efficient pointer Alias Analysis is vital when optimizing or analyzing programs involving pointer manipulation which are common in systems programming, embedded systems applications and performance-critical applications.

Pointer alias analysis plays a central role in supporting other program optimizations. Many compiler optimizations, including common subexpression elimination, dead code elimination and loop optimization rely on accurate pointer alias analysis results in order to execute safe and effective transformations. A thorough understanding of aliasing relationships between pointers, enables a compiler to safely remove redundant computations, remove dead code that cannot be accessed via any valid pointer and optimize loops by unrolling or vectorizing them. Such optimizations bring about a drastic improvement in performance, particularly for applications where every bit counts. As mentioned above, Pointer Alias Analysis is also essential in program analysis and software testing. In order to understand program behavior and to detect any bugs, security vulnerabilities or performance bottlenecks in programs, static analysis techniques such as data flow analysis, points-to analysis, program slicing and many more are commonly used. Pointer Alias Analysis can be used to detect potential buffer overflow vulnerabilities by detecting unsafe pointer dereferences or invalid memory accesses. Use it to identify potential race conditions or data races in concurrent programs by detecting shared access to aliased memory locations. Pointer Alias Analysis is also essential in software testing

techniques such as dynamic analysis, fuzz testing and symbolic execution, where precise pointer alias information is crucial for creating accurate test inputs or exploring different paths within programs.

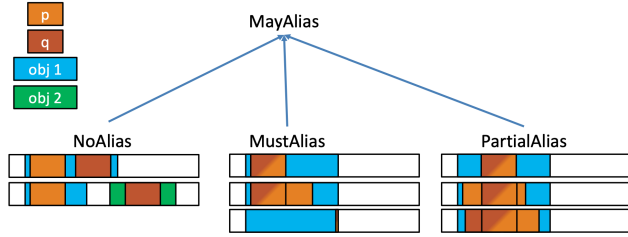
The dynamic and complex nature of pointer operations, together with various programming language features, memory models and optimization levels that must be considered can indeed make Pointer Alias Analysis quite taxing. These facts call for a detailed evaluation on accuracy, precision, and impact of pointer alias analysis techniques which in turn will enhance the current understandings pertaining to their performance, limitations and possible applications.

In this study, I have evaluated pointer alias analysis techniques with the aim of assessing their accuracy, precision, scalability, and robustness. A review of the techniques provided by LLVM to analyze their advantages and drawbacks precedes the experiments. Then, the experiments were conducted using LLVM test suite benchmarks to compare various techniques against each other and gauge their performance against various factors that influence accuracy and precision. Finally, I have also assessed their scalability/efficiency/robustness against potential sources of errors.

The contributions of this research are two fold. First, I offer an in-depth evaluation of pointer alias analysis techniques in terms of accuracy and precision, I then measure their performance combined with various optimizations before outlining future research directions in this field as well as providing recommendations to improve performance, limitations or applications of these techniques.

## 2 Background

Owing to the evolving LLVM community, a large number of compiler optimizations were proposed and implemented. The same applies for alias analysis and nowadays, there is a wide range of proposed alias analysis methods with LLVM supporting several of them. These analysis tools are used to answer the simple question of whether two pointers alias and to what extent they alias each other. The generated answers are no-alias, partial alias, may alias and must alias.



- **No-Alias**: The two locations do not alias at all.
- **PartialAlias**: The two locations alias, but only due to a partial overlap.
- **MustAlias**: The two locations precisely alias each other.
- **MayAlias**: The two locations may or may not alias. This is the least precise result.

Though this atomic function seems elementary, its accuracy, overhead and precision affect the overall optimization process as well as the performance of the program. The process of deducing the alias state, which may be identified using characteristics such as flow sensitivity, context sensitivity and field sensitivity is unique to the implementation. Also the focus of each implementation differs from each other. The following are considered alias analysis methods for the experiment

- **bacis-aa**: Basic alias analysis is a simple and conservative alias analysis that assumes all pointers may alias. It does not perform any complex analysis and is generally used as a fallback when more advanced alias analysis methods are not applicable or not available.
- **tbaa**(Type-Based Alias Analysis): TBAA uses type information to determine aliasing relationships. It annotates memory accesses with type metadata and uses this metadata to determine whether two memory locations can alias or not based on their types. It is a relatively simple and fast alias analysis method that provides precise aliasing information.
- **globals-aa**: Globals alias analysis is a specialized alias analysis that focuses on analyzing accesses to global variables. It takes advantage of the fact that global variables have a global scope and do not change during the execution of a program. It can provide precise aliasing information for global variables, which can be useful for optimizing global variable accesses.
- **scev-aa** (Scalar Evolution-based Alias Analysis): This is a more advanced alias analysis that uses LLVM's scalar evolution framework to determine aliasing relationships. Scalar evolution is a powerful analysis that computes symbolic expressions for loop-based computations, and SCEV-AA leverages this information to perform more precise alias analysis for loop-based memory accesses.

- **obc-arc-aa** (Objective-C Automatic Reference Counting-aware Alias Analysis): This is a specialized alias analysis for Objective-C programs that are compiled with automatic reference counting (ARC). It takes into account the reference counting semantics of Objective-C objects and provides precise aliasing information for ARC-related memory accesses.
- **steens-aa** (Steensgaard Alias Analysis): Steens is a context-insensitive alias analysis that is based on the Steensgaard algorithm. It is a more sophisticated alias analysis that can capture more precise aliasing relationships between memory accesses, but it can also be more expensive in terms of analysis time and memory usage.
- **anders-aa** (Andersson Alias Analysis): Andersson is another context-insensitive alias analysis that is based on the Andersson algorithm. It is similar to CFL-Steens AA but uses a different algorithm for alias analysis. It also provides more precise aliasing information at the cost of increased analysis time and memory usage compared to Basic-AA or TBAA.
- **scoped-noalias**(No-Alias Analysis): No-alias analysis is a conservative alias analysis that assumes that all memory accesses do not alias each other. It is used when the compiler cannot determine any aliasing relationships between memory accesses, and it helps the compiler to perform safe optimizations without introducing incorrect results due to aliasing.

The following table describes the details of the above mentioned alias analysis methods in terms of their ability to identify different aliasing instances.

Alias Analysis method	No-Alias	Partial Alias	May Alias	Must Alias
bacis-aa	Yes	Yes	Yes	Yes
tbaa	Yes	-	Yes	-
globals-aa	Yes	-	-	-
scev-aa	Yes	-	-	Yes
obc-arc-aa	Yes	-	Yes	-

steens-aa	Yes	-	Yes	-
anders-aa	Yes	-	Yes	Yes
scoped-noalias	Yes	-	-	-

## 2.1 Optimization levels

The main factor that affects the performance of a certain AA method on a specific bitcode is its level of optimization. The direction of overall optimization can be either time focused or code size focused where the AA results may largely vary upon the direction and the strength of the optimization. LLVM has inbuilt support for several levels of optimization which are as follows,

- O0- Disable as many optimizations as possible.
- O1- Optimize quickly without destroying debuggability.
- O2- Optimize for fast execution as much as possible without triggering significant incremental compile time or code size growth.
- O3- Optimize for fast execution as much as possible.
- Os- Similar to O2 but tries to optimize for small code size instead of fast execution without triggering significant incremental execution time slowdowns.
- Oz- A very specialized mode that will optimize for code size at any and all costs.

O3 is the extended version of O2 and Oz is the extended version of Os.

## 2.2 Benchmark

After thorough consideration, I selected the LLVM Test Suite for the experiment. It provides a comprehensive collection of test cases designed to ensure correctness and reliability within the LLVM compiler infrastructure. Composed of thousands of individual units, regression, and integration tests that cover every part of its system such as front end, back end, and intermediate representation components, LLVM was evidently the best tool for testing purposes.

The test suite is developed so as to be highly modular and extensible. Therefore, developers can easily add tests as new features are introduced to LLVM. Furthermore, its highly automated operation involves running and analyzing test results using tools such as a test harness, test driver, and various reporting utilities.

The LLVM test suite is an essential element of the development process for LLVM to ensure new features and modifications don't introduce bugs or regressions into the system. The test suite is used by both users and developers alike not only to ensure correctness in their codes, but also to diagnose and debug any potential issues. Within the benchmark 3316 applications are used for the evaluation purpose. The test suite consists of a variety of

applications such as image processing, graph algorithms and optimization algorithms and database implementations. Therefore, I considered this benchmark as a standard sample of the application domain and used it to deduce conclusions about the behavior of alias methods.

## 3 Related works

Alias analysis is one of the major types of static analysis methods and has been the subject of extensive study [1][2][3][4]. The task in hand is to choose whether two pointer variables may point to the same object during program execution. As the problem is computationally expensive [5][6], practically relevant results are acquired via approximations. One popular way to perform alias analysis is via points to analysis, where two variables may alias if their points-to sets intersect. Symbolic Points-to Graphs (SPGs), which contain information about variables, heap objects and parameter passing due to method calls are usually used to express points to analysis as a Dyck reachability problem.[7][8].The distinction between context and field sensitivity is quite significant in alias analysis. Generally, the Dyck parentheses are used in SPGs to identify two types of constraints. Context sensitivity is the condition that reachability paths must obey the calling context due to method calls and returns. Field sensitivity refers to the condition that reachability paths must respect field accesses of composite types in Java or references and dereferencing of pointers [9] in C. Considering both types of sensitivity makes the problem undecidable [10]. While the conventional approach is to consider only one type of sensitivity, approximation algorithms are also a recent solution. [11], Field sensitivity has been reported to produce better results, and be more scalable [12].

The latest trend in alias analysis research is Incremental pointer analysis which is a strategy used to analyze programs and identify how data is accessed through pointers. It works by incrementally updating the results of a pointer analysis as the program is modified or new information becomes available. This approach assists in analyzing large and complex programs, as it can help to improve the accuracy and efficiency of the analysis process. It involves handling code changes that can be formed from one or more insertions and deletions. Handling insertion is straightforward, but deletion is more complex and requires maintaining provenance information to delete all facts that are "no longer reachable" from existing statements. There are two approaches for handling deletion: reset-recompute and reachability based, but both have performance limitations. The reset-recompute algorithm involves resetting the points-to sets of all relevant variables and recomputing them, while the reachability-based algorithm requires repeated whole-graph reachability analysis. REVISER[14] is an IDE/IFDS-based algorithm that adopts a clear-and-propagate strategy to clear and recompute the relative analysis result when necessary. IncA[15] suggests a domain-specific language to incrementally alter

program analysis results based on graph pattern matching. Creating a huge performance enhancement [16] proposed the first parallel incremental pointer analysis algorithm which resulted in a large gain compared to previous methods.

Boomerang[13] is a demand-driven, flow-, field-, and context-sensitive pointer analysis for Java programs. It calculates rich outputs that include both the possible allocation sites of a given pointer and all pointers that can point to those allocation sites. For improved precision and scalability, clients can inquire Boomerang regarding particular calling contexts of interest. This work enhances the research boundaries of demand-driven alias analysis.

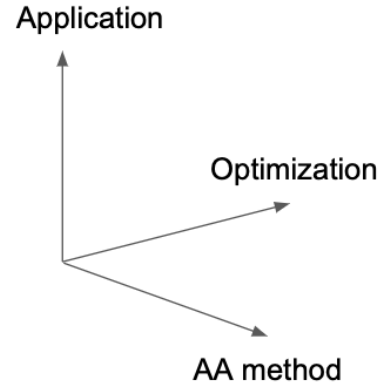
There are several recent efforts on exploiting selective context-sensitivity to accelerate the performance of object-sensitive pointer analysis (i.e., kOBJ) [17], [18], [19]. EA-GLE [17] escalates the efficiency of kOBJ while preserving its precision by conservatively reasoning about value flows via CFL reachability. ZIPPER [18], as a representative of non-precision preserving approaches [18][19], [20], trades precision for efficiency by exploiting several value flow patterns. These techniques reduce the context explosion problem of kOBJ by exploring only a subset of methods in the program context-insensitively. In contrast, CONCH[23] represents a novel mitigation approach as it can debloat contexts for all the objects in the program, allowing existing algorithms to run significantly faster at a negligible loss of precision. TURNER [21] uses object containment to predict context-independent objects. MAHJONG [22] alleviates context explosion by merging equivalent heap abstractions at the cost of precision in alias relations. Bean[24] is a usual approach for improving the precision of any k-object-sensitive analysis, by still using a k-limiting context abstraction. The novelty is to recognize allocation sites that are redundant context elements in k-obj from an Object Allocation Graph (OAG), which is built based on a pre-analysis (e.g., a context-insensitive Andersen’s analysis) performed initially on a program and then avoid them in the succeeding k-object-sensitive analysis for the program.

Lin’s work[25] had a major impact on my project. He uses the may percentage as an indicator for precision. Though he had extended the experiment on multiple benchmarks his work lacks a significant depth as it has not considered the effect of optimization levels as well as lacks the range of the alias analysis methods. Other than that I added more analytics tools to extract stronger insight into the performance of alias methods. Also, the work BEAN[24] which is focused on precision, uses “may alias pairs” as the indicator for precision which assures my decision of choosing may percentage as an indicator.

## 4 Contribution

### 4.1 3-D Evaluation

As mentioned above both the optimization stage and considered application may affect heavily upon the performance of an AA method. Due to that both of them should be variables of solid performance evaluations. With that focus I plan the evaluation as a 3-D evaluation where analysis is done in a manner such that it could observe how the performance fluctuates against both axes.



### 4.2 Technical Approach

As the first step LLVM-test suite is compiled for all the pre-mentioned optimizations. Then I developed a script to extract alias analysis query results from each file save appropriately. I executed the script against all the AA methods. Through that I could gather a substantial amount of data which was used to execute a deeper analysis.

As the second stage the focus was to experiment the effect of different alias methods on various optimizations methods including both memory optimizations and loop optimizations. For that I experimented with all combinations of AA methods and optimizations to gather the execution times. Towards the end I evaluated the effect of different AA methods against combined optimizations on different test groups. I had to develop a script to retrieve execution results.

## 4.3 Analysis

### 4.3.1 First Phase- Effect on Alias Analysis method by different optimizations

As the first phase of the study I experimented to monitor how the performance of various alias analysis methods changes with the level of optimization

I did the alias evaluation for the test suite for all combinations of alias methods and optimization levels. I compiled the test suite for different optimization levels and on each of those outputs did the alias analysis evaluation using each of alias analysis methods.

	basic-aa	tb-aa	objc-arc-aa	scoped no-alias	external-aa	scev-aa	globals-aa	steens-aa	anders-aa
O0	90.08	90.08	90.08	90.08	90.08	90.08	89.84	89.4	89.24
O1	79.49	79.49	79.49	79.49	79.49	79.32	78.38	79.32	79.2
O2	52.08	52.08	52.08	52.08	52.08	51.69	51.67	52	50.94
O3	32.87	32.87	32.87	32.87	32.87	32.66	32.71	32.74	32.16
Os	81.58	81.58	81.58	81.58	81.58	81.15	80.39	81.47	81.4
Oz	84.23	84.23	84.23	84.23	84.23	84.16	82.94	84.11	84.24

Overall may percentages

The results of AA methods type based, object arched based, external and scoped no-alias didn't outperform the basic alias analysis in none of the optimization levels. The results were the same in the file level. Due to that the rest of the evaluation would be more focused on other alias analysis methods.

According to the table it is harder to identify an AA method which has the best overall precision. Therefore it would be meaningful to identify that for each optimization level

- O0 : Anders AA, Steens AA
- O1 : Globals AA, Anders AA,
- O2 : Anders AA, Globals AA
- O3 : Anders AA, Scsev AA
- Os : Globals AA, Scsev AA
- Oz : Globals AA, Steens AA

Though Anders seems to have a precisional advantage, it has a high computational overhead as illustrated in the following table. The evaluation times are given in seconds.

	basic-aa	tb-aa	objc-arc-aa	scoped no-alias	external-aa	scev-aa	globals-aa	steens-aa	anders-aa
O0	160	161	183	163	158	271	186	177	2808
O1	114	116	120	116	112	144	127	119	889*
O2	311	314	323	316	312	360	328	319	960*
O3	844	846	854	843	837	910	863	846	2017*
Os	88	88	92	88	88	114	93	91	2173
Oz	71	72	75	72	70	92	75	75	2295

Overall analysis times

In the above example, executional time is given in seconds. It is obvious that Anders consumes a large amount of resources during the execution. One of the interesting things about the marked entries in anders-aa, namely in O1, O2, O3. During the experiment the program crashed despite all the available resources being given to that. The crash occurred while processing the largest file of the benchmark consisting of more than million alias queries. Due to that marked measurements were taken without certain application which justifies the irregularity in the times.

Scevs have approximately 20% additional overhead and both globals and steens have about 10% additional overheads.

Optimization level one reduces the analysis time as well as final execution time. However further reduction in execution time has largely increased analysis time as seen for O2, and O3. Meanwhile reducing the code size causes reduction in analysis time and overhead.

Though the above data measures the overall performance it doesn't project a real picture about the impact by these AA methods. One of the optimum ways to retrieve that is counting the number of applications against percentages of additional may reduction compared to Basic-aa.

Below table demonstrates the number of applications which are affected by at least one of the four considered AA methods (steens, anders, globals, scsev). The impact by these AA methods significantly reduces when higher levels of optimizations are applied. Seems time focused optimizations reduces the room for an impact and code focused optimizations increase that.

With better analysis, the below insight has been generated for each of the AA methods.

	0% <	5% <	10% <	20% <	40% <
O0	1374	387	207	91	20
O1	989	247	156	61	16
O2	1002	229	135	60	12
O3	995	207	114	53	12
Os	993	255	161	60	17
Oz	948	257	147	63	19

Combined May reduction Statistics as file counts

Below tables illustrate application count of the relevant percentage “may reduction”(reduction in comparative to basic-aa) done by different alias analysis methods. According the the tables of anders and steens it could be seen that though they are effective in lower optimization levels at higher optimization levels they do not perform well when it comes to precision. Also their strength of the impact is quite low as there are less applications which have a higher percentage of may reduction.

However, considering the AA methods globals and scev, their performance is different as they perform well in higher level of optimizations. In particular the strength of the Globals alias analysis is quite high as there are a high number of applications which have a higher percentage of may reduction considering globals.

	0% <	5% <	10% <	20% <	40% <
O0	1200	313	174	85	20
O1	209	42	19	8	0
O2	227	41	23	6	0
O3	217	40	23	7	0
Os	214	42	19	6	0
Oz	219	37	19	5	0

Steensgard May reduction Statistics

	0% <	5%	10%	20%	40%
--	------	----	-----	-----	-----

		<	<	<	<
O0	1260	330	186	89	20
O1	204	43	22	9	0
O2	227	41	25	7	0
O3	221	41	24	8	0
Os	215	41	22	8	0
Oz	231	43	22	9	0

Anders May reduction Statistics

	0% <	5% <	10% <	20% <	40% <
O0	345	53	19	2	0
O1	414	133	86	47	16
O2	383	122	72	46	12
O3	374	107	65	38	12
Os	400	134	90	48	17
Oz	423	138	89	49	18

Globals May reduction Statistics

	0% <	5% <	10% <	20% <	40% <
O0	41	0	0	0	0
O1	742	73	48	4	0
O2	788	65	39	6	0
O3	783	60	25	6	0
Os	755	80	49	4	0

Oz	695	79	37	5	1
----	-----	----	----	---	---

SCEVs May reduction Statistics

### Orthogonality of Alias methods

The implementations and characteristics of the alias analysis methods differ from one another. Due to that these methods may affect one different part of the applications in different manners. Also they may be having the same behavioral nature despite their implementation being different. To evaluate this precisely I implemented a matrix to measure the orthogonality in other words independence between the alias analysis methods.

$$\text{Orthogonality} = 1 - \frac{n(A \cup B) * n(A \cap B)}{n(A) * n(B)}$$

Orthogonal varies between 0 and 1 where 1 refers to completely orthogonal.

A and B stands two distinct alias analysis methods and

$n(A \cup B)$  : Number of applications where there was a may% reduction by either A or B

$n(A \cap B)$  : Number of files applications where there was a may% reduction by both A and B

$n(A \cap B)$  : Number of files applications where there was a may% reduction by A

$n(A \cap B)$  : Number of files applications where there was a may% reduction by B

And the equations were applied for different alias pairs and different optimization levels at different percentage levels. Following are the considered alias pairs. Note that anders was not considered much due to its high computational overhead.

	0% <	5% <	10% <	20% <	40% <
O0	0	0	0	0	0
O1	0	0.01	0	0.03	0
O2	0	0	0	0.05	0
O3	0	0	0	0.04	0
Os	0	0	0	0	0
Oz	0	0	0	0	0

Orthogonality of Steens and Anders

	0% <	5% <	10% <	20% <	40% <
O0	0.25	1	1	1	1
O1	0.56	0.91	1	1	1
O2	0.55	1	1	1	1
O3	0.56	1	1	1	1
Os	0.57	0.94	1	1	1
Oz	0.56	0.93	1	1	1

Orthogonality of Steens and Globals

	0% <	5% <	10% <	20% <	40% <
O0	0.24	1	1	1	1
O1	0.23	1	1	1	0
O2	0.18	0.96	0.93	1	0
O3	0.19	0.96	0.92	1	0
Os	0.22	1	1	1	0
Oz	0.21	1	1	1	1

Orthogonality of Steens and SCEV

	0% <	5% <	10% <	20% <	40% <
O0	0.68	1	1	1	0
O1	0.35	0.98	1	1	1
O2	0.33	0.98	1	1	1
O3	0.34	0.97	1	1	1
Os	0.34	1	1	1	1
Oz	0.32	1	1	1	1

Orthogonality of Globals and SCEV

Considering the relationship between anders and steens it is quite obvious that they perform similarly on the applications. Due to that it will not make much sense using those alias methods together. In comparatively other pairs of the alias methods behave orthogonally. However considering the table it can be seen that when the strength of the optimization increases the orthogonality between the alias methods slightly decreases. Nevertheless it is safe to continue to experiment with following alias analysis combinations.

- Steens- Globals
- Steens- SCEV
- Globals- SCEV

### Experimenting with Alias Analysis pairs

I	scev -aa	globals -aa	steens -aa	anders -aa	steens+ globals	steens+ scevs	globals+ scevs
O0	90.08	89.84	89.4	89.24	89.16	89.4	89.84
O1	79.32	78.38	79.32	79.2	78.21	79.15	78.2
O2	51.69	51.67	52	50.94	51.58	51.61	51.27
O3	32.66	32.71	32.74	32.16	32.58	32.53	32.5
Os	81.15	80.39	81.47	81.4	80.27	81.04	79.96
Oz	84.16	82.94	84.11	84.24	82.82	84.05	82.87

May percentages of combined alias analysis methods

Through this table it can be seen that combining optimizations have enhanced the performance of the alias analysis results. When steens+scevs and globals+scevs have performed fairly well it can be seen that using steens and globals together has made a great improvement in the may percentage reduction.

To take a better insight about these alias pairs I evaluated them on may reduction statistics.

	0% <	5% <	10% <	20% <	40% <
O0	1304	378	196	88	20
O1	551	177	108	54	16
O2	537	166	95	54	12
O3	525	153	88	45	12

Os	547	177	113	54	17
Oz	575	180	109	55	18

Steens+Globals may% reduction statistics

	0% <	5% <	10% <	20% <	40% <
O0	1208	315	175	85	20
O1	800	118	68	11	0
O2	841	110	62	15	0
O3	838	105	49	15	0
Os	811	123	69	10	0
Oz	756	117	57	10	1

Steens+SCEV may% reduction statistics

	0% <	5% <	10% <	20% <	40% <
O0	374	53	19	2	0
O1	945	206	137	51	20
O2	961	190	114	53	16
O3	956	169	91	45	17
Os	942	218	140	53	21
Oz	896	221	126	55	24

Globals+SCEV may% reduction statistics

Above tables demonstrate the application wise “may” reduction(reduction in comparative to basic-aa) of alias analysis methods when applied as a combination. We can see that their is an obvious precision gain by combining the alias analysis methods rather than applying alias methods individually. Especially when steens and globals are combined it can be seen that rather than improving a particular sector both high level and low level optimizations seems to have a precision gain.



#### 4.3.2 Second Phase- Effect on various optimizations by different alias analysis methods


When executing any kind of optimizations it is unavoidable to use the results of alias analysis results. Because alias results represent the underlying connectivity of the programme which is substantially important for efficient optimizations. Therefore when evaluating the alias analysis methods it is extremely important to study the variance of the AA implementations. Therefore as the second phase I experimented how the performance of the optimization methods are affected by alias analysis methods.

The considered set of optimizations for the experiment are

- **Interprocedural Sparse Conditional Constant Propagation(ipscpp):** identifies and propagates constant values across function calls, which helps to eliminate redundant computations and improve program performance.
- **Instruction Combine(instcombine):** merges two or more instructions into a single, more efficient instruction. This can result in a reduction of the number of instructions executed, leading to better program performance.
- **Simplify CFG(simplifycfg):** simplifies the control flow graph (CFG) of a program by removing redundant or unnecessary control flow instructions. This can result in a reduction of the program's execution time and memory usage.
- **Dead code Elimination(dce):** identifies and removes code that is never executed, which can improve program performance and reduce code size.
- **Dead Store Elimination(dse):** removes unnecessary assignments to memory locations that are never read from, which can reduce memory usage and improve program performance.
- **Loop invariant code motion(LICM):** moves code that does not change within a loop outside of the loop, which can reduce the number of times that the code is executed, leading to better program performance.
- **Loop-unroll(loop-unroll):** increases the size of a loop by duplicating loop iterations, which can reduce the number of instructions executed and improve program performance.

For all following combinations I executed the complete test-suite separately and measured the execution times.

Interestingly I noticed that rather than a single alias analysis method giving the best performance for all optimizations , different AA methods were most suitable for various optimizations. This is because these alias analysis methods have diverse implementations which are built targeting various features. However it should be noticed that though there is some fluctuation in the execution times the performance gain could be achieved through the optimal AA method is about 1-3%.

Optimization	Basic	Globals	SCEV	Steens	anders	Best performing 
ipscpp	10417.73	10466.82	10677	10599.95	10375.71	Anders
instcombine	10464.81	10423.92	10474.73	10464.64	10613.76	Globals
simplifycfg	10630.49	10399.83	10771.06	10505.89	10570.27	Globals
dce	10648.88	10587.85	10422.96	10377.54	10547.77	Steens
dse	10533.35	10434.72	10903.64	10580.2	10404.17	Anders
LICM	10746	10723.37	10629.55	10700.84	10846.45	SCEV
loop-unroll	10603.34	10462.79	10657.79	10408.53	10462.28	Steens

Execution times for individual optimizations for different AA

Application Group	Basic	Globals	SCEV	Steens	anders	steens+g lobals	steens+s cev	globals+ scev
Group 1	1780.04	1500.34	1496.87	1496.66	1495.12	1493.68	1543.37	1493.61
Group 2	1613.8	1472.11	1466.03	1469.35	1467.25	1464.71	1518.19	1465.87
Group 3	1603.95	1471.7	1465.88	1468.56	1466.16	1463.17	1499.56	1465.77
Group 4	1591.35	1470.65	1462.57	1468.12	1464.61	1463.11	1484.7	1465.72
Group 5	1577.82	1470.04	1462.43	1468.01	1461.38	1462.82	1474.11	1464.55
Group 6	304.5	317.66	295.54	295.22	293.51	296.39	294.44	295.97
Group 7	299.37	211.85	206.64	214.46	207.64	217.37	217.58	215.08
Group 8	282.57	198.35	192.48	197.58	193.82	202.54	216.74	189.25
Group 9	138.51	130.12	129.23	128.33	128	128.06	138.08	128.35
Group 10	132.29	128.42	122.82	122.13	123.69	122.25	127.68	120.27
Group 11	74.9	76.56	72.93	73.5	71.85	72.9	75.8	72.38
Group 12	57.92	49.46	49.18	49.36	49.22	49.19	52.16	49.22
Group 13	53.2	48.47	48.52	48.37	48.48	48.3	49.53	48.32
Group 14	52.92	48.45	48.13	48.35	48.36	48.19	49.18	48.23
Group 15	52.45	48.43	48.1	48.35	48.24	48.18	48.9	48.23
<b>Total time</b>	9615.59	8642.6	8567.34	8596.36	8567.32	8580.86	8790.04	8570.8
<b>Percentage increase in Exec time compared to Basic-AA</b>		10.11%	10.90%	10.59%	12.12%	10.76%	8.58%	10.86%

Execution times for combined optimizations for different AA

## 5. Future Directions

The above table represents the execution times for different AA methods followed by a combined optimization( combination of above mentioned optimizations). There are several interesting details we can observe from that table.

1. The variance of the effect by AA methods is similar against all considered application groups.
2. Though all the execution times are not similar, all advanced alias analysis methods have given almost the same performance gain. This can be proved using the percentage increases.
3. Despite the higher may percentage reduction monitored when combining alias analysis methods, there isn't a specific advantage considering the execution time.
4. Though the Anders is costly when it comes to compilation it has given the best overall execution time.

Within my project I could cover most of the expected phases for static alias analysis. But I had plans to evaluate the execution times in the resolution of individual applications where we can retrieve the statistics of performance gain at the atomic level which could not be finalized within a given period.

However the alias analysis itself is a broad field and there are plenty of directions that future researchers could focus on.

- **Scalability:** The scalability of alias analysis is a major challenge in analyzing large codebases. Future research in this area could focus on developing more effective algorithms and techniques that can handle more complex codebases. This may involve developing new data structures, pruning techniques, and heuristics to reduce the computational complexity of the analysis.

- **Precision:** The precision of alias analysis is another important area of research. Most existing techniques have limitations in handling complex pointer manipulations and indirect memory accesses, which can lead to false positives or false negatives. Future research could explore new ways to improve the precision of alias analysis by incorporating more contextual information, such as type information, control-flow, and data-flow analysis.
- **Dynamic analysis:** Dynamic alias analysis involves analyzing the program code while it is executing. This approach can provide more accurate results but is often more expensive than static analysis. Future research could focus on developing dynamic alias analysis techniques that are both efficient and accurate. This may involve developing new instrumentation techniques and runtime analysis algorithms.
- **Alias analysis for concurrency:** Alias analysis for concurrent programs is a relatively unexplored area. Concurrent programs introduce additional challenges for alias analysis, such as race conditions and data dependencies. Future research could focus on developing techniques for analyzing aliasing in concurrent programs, which can help identify data races and other concurrency-related bugs.
- **Integration with other program analyses:** Alias analysis is often used as a building block for other program analyses, such as pointer analysis, taint analysis, and security analysis. Future research could focus on developing techniques that can integrate alias analysis with other program analyses to provide more comprehensive program analysis capabilities. This may involve developing new techniques for combining different analyses, such as constraint solving or machine learning-based approaches.

Even in the present there is a large number of ongoing research related to alias analysis which extends the boundaries of the compilation domain.

## 6. Conclusion

As alias analysis is a compulsory analysis for a sturdy optimization, this project is conducted to study the scope of alias analysis. The performance of alias analysis depends on the considered application (program), level of the optimization and the implementation of AA method. Therefore, within the first stage I experimented varying all three factors where I took the “may” percentage as the indicator for precision. Through this, I could bring back many insights and most importantly I could see Steens work well for lower levels of optimizations while Globals do well at higher levels. Moreover, I could see that combining AA methods reduces the “may” percentage which suggests the

precision is increased. For the second stage I planned the experiments to observe how the performance of different optimizations change with the alias method using execution time as the indicator. Though different alias methods gave the optimal performance for different optimizations, when it comes to execution times that optimality is negligible. That is verified through the final experiment which was performed with different AA methods against a combined optimization. Though the advance optimizations gave a performance gain about 10% compared to basic alias analysis, optimal alias analysis is not needed in most cases as their variance is negligible.

## REFERENCES

- [1]Jong-Deok Choi, Michael Burke, and Paul Carini. 1993. Efficient Flow-sensitive Interprocedural Computation of Pointer-induced Aliases and Side Effects. In Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '93). ACM, 232\$245.
- [2]Michael Hind. 2001. Pointer Analysis: Haven'T We Solved This Problem Yet?. In Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '01). ACM, 54\$61.
- [3]William Landi and Barbara G. Ryder. 1992. A Safe Approximate Algorithm for Interprocedural Aliasing. In Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation (PLDI '92). ACM, 235\$248
- [4]Manu Sridharan, Satish Chandra, Julian Dolby, Stephen J. Fink, and Eran Yahav. 2013. Aliasing in Object-Oriented Programming. Chapter Alias Analysis for Object-oriented Programs, 196\$232
- [5]Susan Horwitz. 1997. Precise Flow-insensitive May-alias Analysis is NP-hard. ACM Trans. Program. Lang. Syst. 19, 1 (1997), 1\$6
- [6]G. Ramalingam. 1994. The Undecidability of Aliasing. ACM Trans. Program. Lang. Syst. 16, 5 (1994), 1467\$1471.
- [7]Guoqing Xu, Atanas Rountev, and Manu Sridharan. 2009. Scaling CFL-Reachability-Based Points-To Analysis Using ContextSensitive Must-Not-Alias Analysis. Springer Berlin Heidelberg, 98\$122
- [8]Dacong Yan, Guoqing Xu, and Atanas Rountev. 2011a. Demand-driven Context-sensitive Alias Analysis for Java. In Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA '11). ACM, 155\$165.
- [9]Xin Zheng and Radu Rugina. 2008. Demand-driven Alias Analysis for C. In Proceedings of the 35th Annual ACM SIGPLANSIGACT Symposium on Principles of Programming Languages (POPL '08). ACM, 197\$208.
- [10]Thomas Reps. 2000. Undecidability of Context-sensitive Data-dependence Analysis. ACM Trans. Program. Lang. Syst. 22, 1 (2000), 162\$186

- [11] Qirun Zhang and Zhendong Su. 2017. Context-sensitive Data-dependence Analysis via Linear Conjunctive Language Reachability. In Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL). 344–358.
- [12] Ondřej Lhoták and Laurie Hendren. 2006. Context-Sensitive Points-to Analysis: Is It Worth It?. In Proceedings of the 15th International Conference on Compiler Construction (CC). 47–64.
- [13] Späth, Johannes, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. "Boomerang: Demand-driven flow-and context-sensitive pointer analysis for java." In 30th European Conference on Object-Oriented Programming (ECOOP 2016). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.
- [14] Steven Arzt and Eric Bodden. 2014. Reviser: Efficiently updating IDE-/IFDS-based data-flow analyses in response to incremental program changes. In Proceedings of the 36th International Conference on Software Engineering. ACM, 288–298.
- [15] Tamás Szabó, Sebastian Erdweg, and Markus Voelter. 2016. IncA: A DSL for the definition of incremental program analyses. In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. ACM, 320–331.
- [16] Liu, Bozhen, Jeff Huang, and Lawrence Rauchwerger. "Rethinking incremental and parallel pointer analysis." ACM Transactions on Programming Languages and Systems (TOPLAS) 41, no. 1 (2019): 1–31.
- [17] J. Lu and J. Xue, "Precision-preserving yet fast object-sensitive pointer analysis with partial context sensitivity," Proceedings of the ACM on Programming Languages, vol. 3, no. OOPSLA, pp. 1–29, 2019.
- [18] Y. Li, T. Tan, A. Møller, and Y. Smaragdakis, "Precision-guided context sensitivity for pointer analysis," Proceedings of the ACM on Programming Languages, vol. 2, no. OOPSLA, pp. 1–29, 2018.
- [19] Y. Smaragdakis, G. Kastrinis, and G. Balatsouras, "Introspective analysis: context-sensitivity, across the board," in Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. New York, NY, USA: Association for Computing Machinery, 2014, pp. 485–495.
- [20] Y. Li, T. Tan, A. Møller, and Y. Smaragdakis, "A principled approach to selective context sensitivity for pointer analysis," ACM Transactions on Programming Languages and Systems, vol. 42, no. TOPLAS, pp. 1–40, 2020.
- [21] D. He, J. Lu, Y. Gao, and J. Xue, "Accelerating Object-Sensitive Pointer Analysis by Exploiting Object Containment and Reachability," in 35th European Conference on Object-Oriented Programming (ECOOP 2021), ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 194. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, pp. 16:1–16:31.
- [22] T. Tan, Y. Li and J. Xue, "Efficient and precise points-to analysis: modeling the heap by merging equivalent automata," in Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. New York, NY, USA: Association for Computing Machinery, 2017, pp. 278–291.
- [23] He, Dongjie, Jingbo Lu, and Jingling Xue. "Context debloating for object-sensitive pointer analysis." In 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 79–91. IEEE, 2021.
- [24] Tan, Tian, Yue Li, and Jingling Xue. "Making k-object-sensitive pointer analysis more precise with still k-limiting." In Static Analysis: 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8–10, 2016, Proceedings, pp. 489–510. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016.
- [25] Lin SH. Alias analysis in LLVM. Lehigh University; 2015