An LLVM Implementation of Shapiro and Horwitz's Points-To Analysis

Adam St. Arnaud and Soham Sinha
CMPUT 680

1 Introduction

Information about pointers is important when attempting to analyze and optimize code. Alias analysis (or points-to/pointer analysis) is a compiler technique used to determine where different pointers point to, and the information learned from alias analysis is critical for code optimization to happen both correctly and as aggressively as possible. Different strategies for alias analysis can be classified as either flow-sensitive or flow-insensitive; that is, they can either have distinct points-to information for different points in a program, or a single solution that holds for all program points [HL]. For a flow-insensitive analysis, the assumption is that instructions of a program can be executed in any order [SH]. In their paper, "Fast and Accurate Flow-Insensitive Points-To Analysis," Shapiro and Horwitz analyze and extend the work of Andersen and Steensgaard and provide two algorithms for flow-insensitive alias analysis.

For our course project, we implemented both Shapiro Horwitz algorithms using the LLVM infrastructure. Steensgaard's analysis was previously implemented in an older version of LLVM but has since been removed due to lack of use and maintenance. Andersen's analysis was never officially integrated into LLVM, but Tristan Smelcher has implemented it in his own branch of LLVM. We started from scratch in our implementation. Details of the algorithms follows in section 2, and our implementation details are in section 3. Experimental design and results are discussed in section 4.

2 Algorithms

Andersen and Steensgaard give flow-insensitive pointer analysis techniques that depend on the creation of an *alias graph*. The vertices of this graph represent variables, and directed edges indicate a may points-to relationship for any point in the program. The key distinction between the two is that Steensgaard only allows an out-degree of one for any vertex and merges vertices as needed to fit this constraint, whereas Andersen allows each vertex to have an arbitrary number of out-edges [SH]. This distinction leads to Andersen's algorithm to have higher precision in its alias analysis but Steensgaard's algorithm to have a more efficient worst-case run-time [SH]. From the completed alias graph, the points-to set can be computed, and final alias information is

obtained from this set; if pair (*p*,*q*) belongs to the points-to set, then *p* may point to *q*, and for any pair (*r*,*s*) not found in the set, *r* must not point to *s*.

Shapiro and Horwitz's first algorithm is a generalization of Andersen's and Steensgaard's algorithms, where the number of out-degree of a vertex, *k*, is an input parameter. As another input parameter, the variables of the program are assigned to one of *k* categories, and only variables that belong to the same category can be merged. Merging occurs whenever a vertex has out-edges to variables in the same category. This generalized algorithm can be tuned to produce the same results as Steensgaard's by assigning all variables to the same category (all vertices must be merged), while assigning each variable to its own category yields Andersen's (no merging will occur).

Their second algorithm utilizes algorithm one as a subroutine. It can be observed that different assignments of the variables to categories can produce different final points-to sets. The idea is to run algorithm one multiple times and to take the intersection of all points-to sets that are formed. Only those points-to pairs that occur in all sets may actually be true, since if a pair, (*p*,*q*), does not occur in even one set, then *p* must not point to *q*. For an input number of categories, *k*, and *N* number of variables, run the algorithm $R = ceiling(log_k N)$ times; this is the minimum number of runs required for every pair of variables to be in distinct categories for at least one run. To assign the variables to categories, first assign each variable a unique number in the range *N*-1. The numbers are written in base *k*, and on the *n*th run, the *n*th digit from the right it used to determine the variable's category. In this way, it is guaranteed that any variable pair is assigned distinct categories for at least one run, since they must differ in at least one digit of their unique numbers.

3 Implementation

3.1 The Alias Graph

The alias graph is the underlying structure used in Shapiro and Horwitz's algorithms. We implemented it using three C++ classes: Edge, Vertex, and Graph. Since the edges are directed, they have a source and a target. Vertices contain a list of the variables represented by them, a list of out-edges, and a list of in-edges (needed for updating other vertices' out-edges on a merge). The Graph class is the main component of our alias graph implementation. Graphs contain a list of all vertices, a map from variable names to the vertices which represent them, and a map from variable names to the categories they belong to. There are methods to create and add vertices to the graph, construct edges between vertices in the graph, handle specific LLVM-IR instructions (see Table 1), and various utility methods.

3.2 Merging:

Merging vertices can occur when an edge is added from a source to a new target. The category map is checked to see if the new target belongs to the same category of any of the source's previous targets (we will say for brevity that two vertices belong to the same category iff the variables they represent belong to the same category). If so, these vertices are merged into one. As a concrete example, say that vertex *v* has a previous target, *x*, that needs to be merged with a new target *y* (see Figure 1). This means that vertex *x* and vertex *y* belong to the same category. To accomplish the merge, *x* receives all targets and sources from *y*, ie. all in and out-edges. Also all variable labels from *y* are added to *x*'s variable label list. There is a recursive nature to merging with respect to the targets. As we create an edge from *x* to *y*'s targets, the category map needs to be checked to compare *x*'s previous targets and the new targets from *y*; if there is a match, then another merge needs to occur, and the process repeats. For example, in Figure 2, vertex *a* may need to be merged with vertex *b*. Note that no extra merging occurs due to any of *y*'s sources now having an out-edge to *x*. Assume, for example, that vertex *u* is a source vertex of *y* and that *u* has another target *z* (See Figure 2). As part of the merge, *u* will now also have an out-edge to *x*. For *z* to possibly merge with *x*, they would need to belong to the same category; however, if this is the case, then *z* would have already merged with *y* itself at a previous time, as *y* and *x* belong to the same category.

3.3 The Shapiro Horwitz Algorithms

We designed a class ShapiroHorwitzAliasAnalysis which is the part of our implementation that interacts with the core LLVM infrastructure. This class first passes once through the given program; on this pass, it identifies all pointer variables. For variables found in an LLVM-IR alloca instruction, variables found as function parameters, and global variables, we also create a unique "stack" label. For variables found in a malloc instruction, we create a unique "heap" label. These variables and the stack and heap labels, along with the input parameter *k,* are then passed to a member class called ShapiroHorwitz. This class utilizes two other helper classes, Categorize and Base, to calculate how many runs are needed, and assigns the variables (including stack and heap variables) to categories for each run. It is possible to run algorithm one by passing in another argument flag. Now that the ShapiroHorwitz class has information about run number and the category assignments, it creates a new Graph for each run and passes again through the program, this time updating the graph on alloca, malloc, bitcast, load, store, phi, and get element pointer instructions. Table 1 summarizes how the alias graph is updated on these instructions. Note that the list of LLVM-IR instructions handled by our implementation is not exhaustive. Since each graph

instance has a different category assignment, the graphs may be different. Once a graph is completed for a given run, the resulting

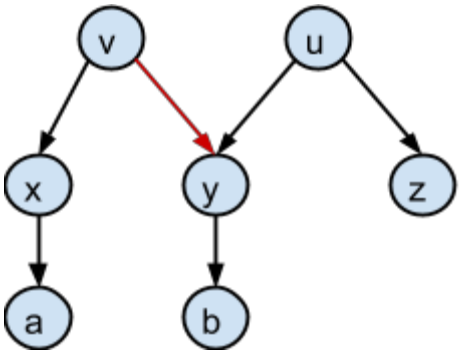

Figure 1: A new edge is created from vertex *v* to vertex *y*. Since vertices *x* and *y* belong to the same category, they must be merged.

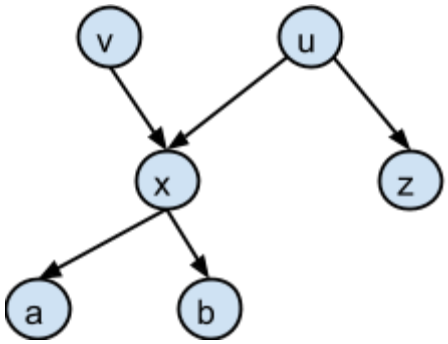

Figure 2: After the merging of *y* into *x*. It is possible that vertices *a* and *b* may need to be merged, but vertices *x* and *z* will never need to be merged.

points-to set is extracted and stored in the class. When all points-to sets have been created, the intersection is calculated. This is done by first identifying the smallest points-to set; then for every pair (*p*, *q*) in this smallest set, we determine if (*p*, *q*) is also in every other points-to set - if so, it is part of the intersection. Finally, to determine if two pointers *p* and *r* may alias, there is an *alias* method to check for the existence of a points-to pair (*p*, *s*) and (*r*, *s*).

Table 1: Effect on the alias graph for different LLVM IR Instructions

LLVM-IR Instruction	**Effect on Alias Graph**
alloca	A vertex is created for the allocated variable and for the unique stack label it corresponds to. An edge is created from

	the variable vertex to the stack vertex.
malloc	A vertex is created for the allocated variable and for the unique heap label it corresponds to. An edge is created from the variable vertex to the heap vertex.
bitcast	A vertex is created for the variable receiving the value. Edges are created from this vertex to all targets of the RHS variable vertex.
load	A vertex is created for the variable receiving the value. Edges are created from this vertex to the targets of all of the other variable's targets.
store	For a call which stores the value of variable *x* into the spot where variable *y* points to, edges are created from all targets of *y* to all targets of *x*.
phi	A vertex is created for the variable receiving the value. Edges are created from this vertex to all targets of each RHS variable vertex.
get element pointer	A vertex is created for the variable receiving the value. Edges are created from this vertex to all targets of the RHS variable vertex.

4 Experiments and Evaluation

In our first experiment, we evaluated our implementation on 5 test programs taken from the test directory of LLVM3.4. As it is a prototype of the Shapiro-Horwitz algorithm and has limited functionalities, we have been unable to successfully test it for some larger programs. For each program, we have executed: Shapiro Horwitz algorithm 2 with k = (2,3,4,5) three times and taken the average running time. We also tested our implementation by combining our analysis pass to the *basicaa* Alias Analysis pass of LLVM. We did not notice any improvement over the results of *basicaa* pass. The reason

behind this is the incomplete set of instructions we have covered in our implementation. We hope that if we extend our prototype to a full-fledged implementation, then *basicaa* combined with our implementation may result in a more precise analysis than either can provide alone. We have used the LLVM pass *aa-eval*, which queries all possible pointer pairs and returns a summarised result of the alias relationships between them. We have evaluated the running-time by using *time-passes* pass of LLVM.

Table 2 shows the variation of running-time with the number categories for different programs. It also has size of points-to set and the total number of variables in each of the programs. Figure 3 depicts the Category vs Time graph for the same experiments. The legends on the right side of Figure 3 represents the number of lines for each of those programs. The time has been recorded as a percentage of the total time-taken by the analysis as we ran the analysis along with aa-eval. From Figure 3, we can see that the running-time gets higher when we increase the category from 1 to 2 and then it levels off or sometimes decreases. For small programs such as with 31 lines of LLVM IR code, the number of categories does not affect our output significantly. When the number of categories is 1, then the algorithm practically runs Steensgaard's algorithm. When we increase the number of categories to 2, the Shapiro-Horwitz algorithm 2 runs algorithm 1 multiple times. Because of this multiple iterations, the running-time increases. However, as we continue to increase the number of categories, fewer merges occur in the graph, along with decreasingly fewer number of runs. So, the running-time also decreases. This result is in accordance with the result observed in the original [SH] paper for smaller programs. They have found that Andersen's algorithm actually runs faster than Steensgaard's algorithm for those small programs (even though the worst case run time of Andersen's is greater). So, we believe if we increase the categories to equalize with the number of variables, the running-time would come close to the running time of Steensgaard's algorithm.

One of the other important points to observe is the variation of the size of points-to set with increasing number of categories. Although we have observed significant changes to size of the points-to set with different categories, we have not been to able test this experiment with large programs because of the small instruction-set we are considering. For the program, *Analysis/BasicAA/phi-aa.ll*, our implementation generates 7 points-to relationship with 1 categories and 5 points-to relationship with 2 categories. Fewer points to relationship also improves the precision of alias queries for this program.

With 1 category, we were able to detect only 3 *may points-to* relationship, but with 2 categories, we had detected 4 *may points-to* relationship. The result with 2 categories exactly matches the result observed by running *basicaa* pass. We have also observed similar kinds of improvements for our own written programs which were precisely created for possible complexity in alias analysis (See attached files) [Files].

Table 2: Time-taken for Shapiro-Horwitz analysis for different programs with different number of lines

Lines	**Size of PointsTo Set**	**Number of Pointer-Variables**	**Time(%)**
661	84	94	42.50
			68.07
			63.07
			55.40
			54.73
699	86	97	26.63
			47.83
			43.43
			43.47
			40.73
31	7	5	98.83
			98.77
			98.33
			98.67
			98.90
535	41	82	24.70
			51.97
			43.13
			38.47
			39.50
604	96	96	51.40
			69.97
			63.37
			62.27
			62.87

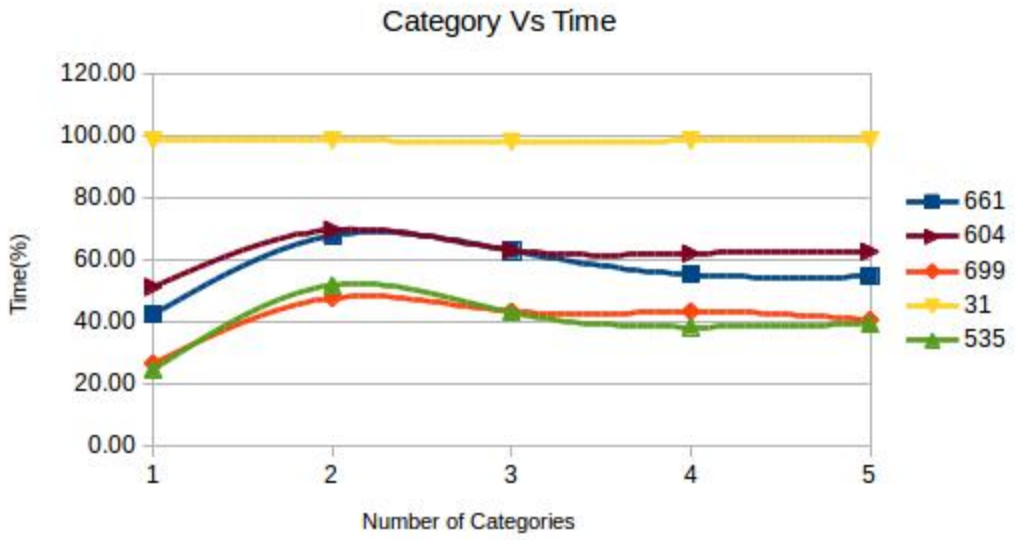

Figure 3: Time-taken for different number of categories for different programs

5 Future Work:

We have tried to cover important LLVM instructions in our implementation. Still, there are instructions which we didn't handle like calls to functions other than malloc, `addrspacecast` etc. We have planned to implement function calls following the suggestion given in [SH]. Additionally, forward-referencing is also not adequately handled. In case of phi functions, there can be a reference to a variable which is defined later in the program. This kind of forward referencing can be extended with a chain of phi-functions in the program. We handled the base-case of this situation and plan to extend it to support the chain of phi-functions in future. We think that a typical iterative algorithm for static analysis might be appropriate for this problem. There is a case where NULL values can be assigned to a pointer variable. We have not considered this specific case as well.

Handling these new cases may lead to further unforeseen complication; however, as we have built our `Graph` class from scratch, we believe we have the infrastructure in place to extend our implementation as needed.

Conclusion:

We have implemented the flow-insensitive points-to analysis algorithms as described by Shapiro and Horwitz [SH]. Our implementation should be considered as a prototype, as there are aspects of LLVM-IR which we have not considered yet; however, our implementation is sufficient to run on several on the provided LLVM test programs. During our implementation of alias graphs, we observed and proved the fact

that merging happens recursively with respect to the targets of merged vertices and that this is not true with respect to sources.

We have seen that the categories have significant effect on the size of the output sets and should be studied with care. Increasing the number of categories after a certain number has insignificant effect on the running-time. However, as we experimented with only small programs, we hope that with larger set of test programs, we will be able to deduct more meaningful conclusions.

There are cases which we have considered but due to time constraints have not incorporated into our implementation, but we believe further extensions can easily be added to our infrastructure.

References:

[HL] Hardekopf, Ben, and Calvin Lin. "Flow-sensitive pointer analysis for millions of lines of code." *Code Generation and Optimization (CGO), 2011 9th Annual IEEE/ACM International Symposium on*. IEEE, 2011.

[SH] Shapiro, Marc, and Susan Horwitz. "Fast and accurate flow-insensitive points-to analysis." *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1997.

Appendix:

[Files] We have included the generated graphs for k=2 and the test program we ran it on. Observe that different graphs are possible for different assignments to categories.

We are providing the link to our Github repo with all of our code.
https://github.com/sohamm17/ShapiroHorwitz