Chapter 10

Conclusions and Future Work

10.1 Summary and Conclusions

The optimization phase is one of the crucial phases of a compiler. In order to get effective results in this phase, one needs sophisticated flow analysis techniques to collect accurate information about the various variables and aggregate structures used by a program. In the presence of pointer type variables, the performance of most of the analyses depends on the accuracy of alias analysis.

In this thesis, we discussed an accurate and practical alias analysis algorithm that collects the points-to relationship between variables instead of explicit alias information in the form of exhaustive alias pairs. Our analysis is focused on stack-allocated data structures. We consider single and multi-level pointers, aggregate structures (arrays, structures, and pointers to arrays), type casting, and arithmetic operation on pointers. The algorithm has been implemented in the framework of the McCAT compiler, and is being used by several high and low-level analyses.

For a given program, we first build the invocation graph with a special representation for recursive function calls. Then, we perform the interprocedural points-to analysis in two phases: (i) a first pass over the program to collect all the variables relevant to our analysis (this is done to minimize the size of our bit matrix data structure), (ii) a second pass to collect the points-to information by developing and using an explicit rule for each possible statement. We handle function calls by following the actual program execution paths using our invocation graph.

The results of points-to analysis is used when there exists an indirect reference (e.g., `*x`) in the program. Our experimental results over some standard benchmarks show that the average number of variables that a pointer type variable points to, is very close to unity (1.04). In other words, an indirect reference `*x` is in almost all cases, either possibly or definitely aliased to a single variable. We also conclude that around 47% of all the points-to relationship are definitely-points-to, these relationships are very valuable for any analyses. These results are due to our accurate analysis methods.

10.2 Future Work

There is a considerable scope for further work in the following directions:

- Extension to dynamic data structures: Our analysis is designed for stack-based variables. Presently a single location is used to represent all the dynamically allocated objects in order to be able to handle benchmarks. An appropriate abstraction to handle dynamic data structures is currently being developed, based on the path-matrix model proposed in [Hen90, Ghi93a].
- Procedure cloning using points-to information: When the points-to information propagated from two different calls to a function is significantly different, the information from one of the calls can be propagated to a clone of the function. In this way, the imprecision resulting from merging significantly different information can be avoided. As the input to a function for each call is saved in our invocation graph, this feature can be easily added to our analysis.
- Precise saved information: Presently, the merged information gathered during different passes to a function is saved in each statement node of the program under analysis. This leads to some imprecision caused by the merge. In order to get the exact result, one can save the points-to information at each statement for different passes, separately along with the related invocation graph node. Since each node in the invocation graph represents a call-chain, and it is unique for each call, keeping this extra information is sufficient to differentiate points-to information of different calls. Therefore, the only cost would be the size of data. If this size is not an issue, this extension can be naturally added to our analysis.
- Extension to our array analysis: Presently, we use one location in our abstract stack for an entire array. However, we treat an array differently depending on, if the address of its first element or some other element is assigned to a pointer. We further plan to refine this scheme and use two locations for each array. One for the first element and one for the other elements. In our method a possibly-points-to relationship[1] for an array occurs due to: (i) a merge at the join points

[1] Refer to Chapter 3 for definition.

of compositional control statements, or (ii) assigning the address of an element of the array, which is not its first element, to a pointer. Currently , we can not distinguish whether a possibly-points-to relationship for an array is caused by (i) or (ii). While in the proposed method the other array element will be used for case (ii). Therefore, the distinction between this two cases would be possible.

- Array parameters: In the C-language, when an array is passed as a parameter, it is changed to a pointer variable in the callee. Therefore, one can not know if a pointer variable is a pointer to array or not. Using our analysis, we can find out if a pointer points to an array. We can also provide the range of the pointed-to array. This can be done using our invocation graph and the map information which is saved in each node of the invocation graph. Suppose a pointer variable `x` points-to a variable `y`. If `y` is in the scope of the callee, one can easily know if it is an array and also what is its range. Otherwise, it is an invisible variable[2], and the map information saved in each invocation graph node can be used to get the equivalent variable(s) for this invisible variable.

  The result of this extension would be useful in handling pointer arithmetic, since we handle arithmetic operation on pointers to arrays more precisely than on pointers to scalar variables. Further, this information is very useful for array dependence analysis.

- Union type: Our analysis can be easily extended to handle union types. We just need to generate names for all the possible access to a union type.

- Points-to information saved in invocation graph: The abstract stack representation before and after processing a function call, is saved in each node of an invocation graph. This information can be used as a data base, to reduce the passes to a function call. If an input abstract stack has already occurred and an output abstract stack is computed for it, in the second appearance of the same input, one can use the already computed output.

[2] Refer to Chapter 4 for definition.