Alias Analysis Improvement in LLVM

Yogesh Chavan (MTech, IIT Hyderabad)
(cs13m1012@iith.ac.in)

Objective

To provide a most efficient alias analysis algorithm for LLVM.

Criteria of Success

At the time of writing, Alias Analysis in LLVM isn't in an ideal state. The standard LLVM build comes with a few simplistic alias analysis implementations, but nothing incredibly advanced. This is because traditional vanilla alias analysis methods either take a large amount of time (Anderson Analysis at cubic time and large memory requirements) or are cheap and somewhat imprecise (Steensgard Analysis at linear time). More recently, demand-driven, context-insensitive CFL-based methods have been researched, which prove more accurate than Steensgard's, while significantly faster than Anderson's. Thus, we have selected a variant of one of the CFL-based approaches for implementation in LLVM.

The algorithm selected is an optimized version of what is outlined in [1]. In short, the algorithm in [1] models each value (variable/array/struct/memory location/…) as its own node in a graph, and MayAlias analysis is done by finding a path from one node to another. Each edge in the graph is labelled; if a path is found wherein the language formed by concatenating the labels of the edges traversed meets a specified grammar, then the two variables considered to potentially alias. In testing, the authors found that, with a search budget of 500 nodes, they could conclusively answer anywhere from 80-100% of mayAlias queries conclusively. On average, this search of up to 500 nodes took around 520us. This is unacceptable because alias analysis can potentially be queried *over one million times* for a single .cpp file. Assuming 520us is taken to answer each query, compiling that file will take quite a long time. So, a better approach is needed.

The optimized version of this algorithm adds a time overhead at initialization of O(n + m log m) (n == number of nodes, m == number of edges) and space overhead of O(n + m). Experimental results show that the number of edges is usually within the same order of magnitude as the number of nodes and, if anything, there are *less* edges than nodes [2]. This extra time is spent constructing what is effectively a list of sets. Each set contains values. The only way that two values may alias is if they reside in the same set. Now, instead of a graph search, we have a maximum of two set queries to perform for each call to alias(). As an added benefit, in practice, the graph representation also requires more sustained working set than the set representation. So, AA can be done with less memory using this method than with the CFL-based approach.

Ultimately, the optimized CFL algorithm was chosen because it shows promise of giving more accurate results than something like Steensgard's within similar algorithmic time bounds. At initialization, it seems that we do observe an algorithmic penalty (O(m log m + n) build time, as opposed to O(n)). However, if we can overcome this, it could prove to generate faster code with a minimal impact on compile time.

Background

With the selected algorithm, a few key decisions need to be made. These include:

- How to handle struct member access
- How to handle array access
- How to handle indirect function calls (virtual, or through function pointers)
- How to handle NULL pointers/uninitialized pointer values
- How to handle opaque function calls (i.e. functions we don't have the body for)

For struct access, there are three approaches generally taken: treating entire structs as nodes, treating individual struct members as nodes (not structs), and treating structs and each individual member as a node.The first option, treating each struct instance as a node, has been selected due to simplicity. If time permits, we will also explore treating each member as its own node.

On the topic of arrays, three approaches are favored by researchers: treating an entire array as a node, treating each element of the array as a node, and treating array[0] and array[1:] as two distinct nodes. To start with, we will take the approach of treating an array as a node. However, it may be worth exploring the impact of treating array[0] and array[1:] as two separate nodes.

In the case of indirect function calls, we will try to determine the set of functions that can be called. If this is possible, we will query each of these about the arguments we have passed in, globals that may be changed, etc. Otherwise, we have to treat them as opaque, because we cannot determine what could possibly be called.

Opaque function calls can basically only be treated in one way. We have to assume that all things that are reasonably accessible by the opaque function call may alias any of the results of the function. That is, any pointers to locals passed in, anything derived directly from the function's return value, and any arguments in scope may alias anything else that was not passed in/that we do not control.

Finally, LLVM treats each store/load from NULL or uninitialized values as not aliasing. This allows us to potentially be more precise in our algorithm, so each instance of NULL/uninitialized values will be noted in the graph as its own (otherwise inaccessible) node.

Timeline and Testing Methodology

Roughly 3 months required to finish the initial analysis and finalize the algorithm/methodology to improve the Alias Analysis in LLVM.

References

[1] - www.cs.cornell.edu/~rugina/papers/popl08.pdf
[2] - http://www.cse.cuhk.edu.hk/lyu/_media/paper/pldi2013.pdf