Google Summer of Code 2016 Proposal

Better Alias Analysis By Default

Background

Alias analysis is one of the most important enabling techniques in an optimizing compiler: without knowing that two pointers cannot alias, performance-critical compiler optimizations like loop invariant code motion, load/store code motion, and dead code elimination could not be applied.

Although LLVM uses an IR of partial-SSA form to trivialize the alias analysis of non-address-taken variables, it still relies on dedicated alias analysis passes to obtain aliasing information for non-promotable memory locations [1]. There are five alias analysis passes in the current LLVM codebase (as of March 2016):

- Basic-aa, which applies various local heuristics to catch many common aliasing/non-aliasing patterns.
- Globals-aa, which is a cheap bottom-up context-sensitive analysis that obtains mod/ref information for globals and call instructions.
- Tbaa, which relies on type annotations inserted by the front end to detect aliases
- Cfl-aa, which formulates alias analysis as a context-free language reachability problem and solves it in a demand-driven fashion.
- Scev-aa, which relies on ScalarEvolution pass to get more precise aliasing result in loops.

However, only the first three are actually enabled by default. Useful as they are, those enabled alias analysis passes also have their weaknesses:

- Basic-aa is mostly an intraprocedural analysis. It cannot reason about pointers that may escape the current function. In addition, it recursively walks through use-def chains for each alias query. This is slow and requires a multi-layer caching scheme to speed it up.
- Globals-aa is interprocedural, but its scope is rather limited: it only handles non-address-taken globals and simple facts about function invocations.
- Tbaa is effective when the source language has a strong type system. For weakly-typed language like C, the usefulness heavily depends on the front-end’s ability to emit good type-related metadata.

The cfl-aa pass implemented by Gerorge Burgess IV back in GSoC 2014 [2] overcomes most of the listed weaknesses: it is fast [3], precise [4], interprocedural, works on pointers of all kinds, and does not depend on type annotations. It is also said to be easily extensible to add field-, flow-, and context- sensitivity. Nevertheless, the pass is checked in but not enabled in today's LLVM build due to (1) various self-hosting miscompilation bugs [3], and (2) not sufficiently tuned for existing optimization passes that uses it.

The goal of this GSoC project is to bring cfl-aa to a usable state and make it a good complement, if not a replacement, of the existing alias analysis pipeline. If time permits, I could also try to do the same for scev-aa.

Objectives

- Fix existing miscompilation bugs and performance regressions of cfl-aa
- Resolve all issues that prevent cfl-aa from being turned on by default
- Tune various alias analysis clients to better capitalize on cfl-aa
- Enhance cfl-aa so that clients get more precise information from it
- (Optional) Improve the performance of scev-aa and also turn it on by default

Implementation Details

The deliverables of this project should be a set of patches. Source codes under the Analysis library should be my primary playground. Most of the time it is going to be CFLAliasAnalysis.h and CFLAliasAnalysis.cpp, although to fine-tune the alias analysis framework I would also need to touch AliasSetTracker, MemoryDependencyAnalysis, etc.

To fully unleash the potentials of cfl-aa, studies on how its clients may benefit from it also needs to be conducted to get a better understanding of the best precision-performance tradeoff. This task may require some comparisons between existing aas and cfl-aa, and also some investigations on important cases where cfl-aa fails.

Alias analysis in general is notoriously difficult to debug. Fortunately, tools are available to help making the task easier. Neongoby [5] is a tool that instruments the program, analyzes the traces, and verifies that an alias analysis does not contain soundness bugs for a particular execution. I have used it in the past and found it to be very helpful in bug detection. I believe it is also going to come in handy in this GSoC project.

Biography

I am Jia Chen, a Ph.D. student in Computer Science department at University of Texas at Austin.

I do research in pointer analysis and I use LLVM as the target language of my analysis. I am confident that my familiarity of pointer analysis as well as my familiarity of LLVM qualifies me as the right person for this project.

As a proof of my expertise, here are two of my projects that I wrote in the past:

- Andersen-style pointer analysis on LLVM. There used to be an anders-aa pass in LLVM trunk but it got removed later due to lack of maintenance. This is an alternative and a complete implementation of Andersen's algorithm with all optimizations proposed in Ben Hardekopf's publications turned on. For many years It was the only working Andersen's analysis for LLVM one can find on github.
- Fully-sparse field-sensitive flow-sensitive and context-sensitive taint analysis on LLVM. The taint tracker relies on a very efficient flow- and context- sensitive pointer analysis that propagates points-to information in a fully sparse fashion.

I also consider myself a very proficient C++ programmer. I am the current instructor of C++ programming class taught here at UT Austin CS department.

Reference

[1]http://llvm.org/docs/AliasAnalysis.html

[2]
https://docs.google.com/document/d/1nGFKMmr-HbdEiag9G1GeWurgOV0CweSUjLXFr3LAwgg/edit

[3] http://lists.llvm.org/pipermail/llvm-dev/2016-March/097189.html

[4] www.cs.cornell.edu/~rugina/papers/popl08.pdf

[5] https://github.com/wujingyue/neongoby