Code Optimization and Transformation Course Project on:

Program Dependence Graph

The Code Optimization and Transformation course exam is composed by two parts. One is an oral test, the other is an homework, to be terminated before course last call. To pass the whole exam, you must get a pass grade from both the test and the homework. The homework must be taken in pairs.

During the lab classes, the LLVM [6] compiler has been introduced. The homework must use the 3.0 release version of LLVM. A sample project – COT passes [7] – is available on GitHub [5]. It must be used as a starting point for the homework. LLVM testing framework [4] must be used to validate the implementation.

Sources must versioned using Git [8]. A good tutorial can be found here [3]. Sources must be published on GitHub [5].

Assignment

You are required to build analysis passes computing the following graphs over functions expressed using LLVM Intermediate Representation:

Data-dependence Graph: encodes *data-dependencies* between nodes in the Control-flow Graph. The definition given in [1] labels edges according to hardware constraints. Since the DDG must be build over LLVM IR, do not consider hardware details – e.g. instruction latencies – and thus do not label graph edges.

Control-dependence Graph: encodes *control-dependencies* between nodes in the Control-flow Graph. Refer to [2] for the algorithm computing the CDG.

Program-dependence Graph: encodes both *data-dependencies* and *control-dependencies* between nodes in the Control-flow Graph. Edges are categorized based on the kind of dependency they represent. It can be built starting from the PDG and the CDG.

Each graph must expose accessors used to determine whether a given couple of nodes are dependent – e.g. `DataDependenceGraph::depends(const llvm::Value *, const llvm::Value *)`.

The output of each analysis can also be inspected using the console – i.e. calling `opt` with the `-analyze` switch, and graphically – e.g. like the `view-cfg` pass.

Advices

Inside LLVM there is a standard procedure for building this kind of passes:

1. define a class for your graph – e.g. implement the DataDependendeGraphBase class
2. define the class implementing the analysis by sub-classing both llvm::FunctionPass and your graph class – e.g. DataDependenceGraph. Implement graph construction in the analysis runOnFunction member function. Implement text-based visualization of analysis results in the print member function
3. provide graphical rendering of analysis by implementing a pass sub-classing and specializing llvm::DOTGraphTraitsViewer
4. provide DOT-based output of analysis by implementing a pass sub-classing and specializing llvm::DOTGraphTraitsPrinter

In order to exploit LLVM infrastructure for viewing/printing the graph, you have to specialize LLVM graph traits – llvm::GraphTraits – and LLVM DOT graph traits – llvm::DOTGraphTraits – for your graph.

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers – Principles, Techniques, and Tools*, chapter Instruction Level Parallelism. Pearson.

[2] Andrew W. Appel. *Modern Compiler Implementation in Java*, chapter Static Single-Assignment Form. Cambridge University Press.

[3] Scott Chacon. Pro Git. URL `http://git-scm.com/book`.

[4] John T. Criswell, Daniel Dunbar, Reid Spencer, and Tanya Lattner. LLVM Testing Infrastructure Guide. URL `http://llvm.org/releases/3.0/docs/TestingGuide.html`.

[5] GitHub Inc. GitHub. URL `http://github.com`.

[6] University of Illinois at Urbana-Champaign. Low Level Virtual Machine. URL `http://www.llvm.org`.

[7] Ettore Speziale. Compiler Optimization and Transformation Passes. URL `https://github.com/speziale-ettore/COTPasses`.

[8] Linus Torvalds. Git. URL `http://git-scm.com`.