12 points

Course Project Deliverable 5

CS 4620
assigned: 11/18/2022
due: 12/2/2022 at 5pm

The course project provides a context for you to study concepts related to compilation more deeply. This is the final of the 5 project deliverables:

Deliverable 1 Extend the parser to support SIP

Deliverable 2 Extend the front end to support SIP

Deliverable 3 Extend the semantic analyses to support SIP

Deliverable 4 Extend the code generator to support SIP

Deliverable 5 Extend the optimizer for SIP

The project is solved in teams of two, unless you want to do it by yourself.

Final Deliverable : Problem Statement

In this deliverable, you are to extend the optimizer to improve the quality of LLVM bitcode generated. The existing optimizer includes 5 passes:

- `PromoteMemoryToRegister` : moves values from memory to register wherever possible and builds SSA form
- `InstructionCombining` : collection of optimizations that combine nearby multiple instructions into one (aka "peephole" optimization)
- `ReassociatePass` : applies associative rule to expose opportunities for other optimizations
- `GVNPass` : applies global value numbering to detect common subexpressions
- `CFGSimplifcationPass` : collection of optimizations that restructure CFG, e.g., merging blocks, removing blocks, etc.

The optimizer takes a pretty simple approach:

- it runs optimizations on each function independently (using a `FunctionPassManager`)
- it runs optimizations in a fixed order
- it runs each optimization once (it ignores the return value of the `run` method of the function pass manager)

In this deliverable, you are to develop a more sophisticated approach to optimization for the SIP language. You are to do this by:

- adding additional optimization passes for each function
- adding interprocedural optimization passes for the entire program module
- running optimizations multiple times until no further improvement is possible

Unlike prior deliverables a significant portion of the work on this deliverable will involve:

- Understanding the available LLVM optimization passes and how they might improve SIP programs
- Selecting LLVM function and IPO passes to add to the optimizer
- Assessing the benefits of selected passes

There are many ways that you can measure benefit in optimization. For this project you may use the number LLVM bitcode instructions and declarations. Of course not all bitcodes are created equal, e.g., a `load` or `store` is much more expensive than other bitcodes. Note also that there are optimizations that can expand the size of the program and improve runtime performance and we encourage you to explore those as well. You are permitted to define your own performance metric to accomodate your evaluation, just be explicit in the rationale for your metric.

For each pass, including the 5 included in the base optimizer, you must justify its inclusion with a small example program that demonstrates the benefit of the optimization *in combination* with the other optimizations. This is sometimes referred to as an *ablation* study in which you remove a single optimization and show reduced performance.

The project is open-ended in the sense that you can add as many optimizations as you like, but you must justify each optimization that you add. You must also explicitly discuss your selection process and why you stop exploring additional optimizations; an inadequate justification for not incorporating optimization passes will lead to a reduced score.

Final Deliverable : Accessing the Project, Solution Expectations and Grading

Work on this deliverable is a continuation of your prior deliverable. Just keep working in the same repository that your team has been using.

You are expected to extend and modify, as necessary, the implementation of the `tipc` optimizer, e.g., `tipc/src/optimizer/Optimizer.cpp`. To facilitate performing your ablation study you may want to add command-line options to disable individual optimizations, e.g., `-do=gyving`, and other aspects of the optimization approach, e.g., disabling multiple runs of a pass. The description of your work should in a file `SOLUTION-FINAL.md` that you

add to the root directory of your project. You are to describe your process for selecting optimizations, describe each optimization that is included, and describe the evidence that supports its unique benefit in optimizing SIP programs. In addition, you are to include a "big bang" optimizer test that is a substantial SIP program that really shows off the power of your optimizer.

We will access the latest project commit prior to 5pm Eastern Time on Dec. 2, 2022 to assess your solution. Grading will be based on the following rubric:

Quality of Description (6pts) The completeness and clarity of the description of your solution.

Quality of Solution (4pts) The quality of your optimization pass in terms of its ability to improve LLVM bitcode.

Big Bang Test (2pt) The extent to which your big bang test integrates all of the different optimization passes and shows benefit.

Final Deliverable : Scope, Strategy, and Suggestions

Unlike other deliverables the work for this deliverable is not depicted very effectively by measuring the lines of code of the implementation. Our implementation adds less than 70 lines of code including new commandline argument processing.

What will take time and effort is consideration of the different optimization passes available in LLVM's `Scalar.h` and `IPO.h` in light of their potential for improving bitcode generated by your compiler. In looking for passes, you can look in `Utils.h`, which is where the `PromoteMemoryToRegister` pass is defined, or in subdirectories of `llvm/include/llvm/Transforms` as well. For example, the `createGVNPass` is actually in `Scalar/GVN.h` file not the `Scalar.h` file. The legacy pass constructor methods are named `create*Pass` and they return either `FunctionPass` or `ModulePass` for intra and inter-procedural passes, respectively.

LLVM also supports vectorization in that a number of bitcodes that can operate on vectors of values, rather than individual values. The passes in `Vector.h` seek to transform programs into this form and this is another area that you can explore for optimization of SIP.

There are many passes defined in LLVM and you will find many that will have little or no impact on `tipc` generated bitcode; you can read the in-code documentation of the passes to get a sense as to whether they offer potential.

The current `Optimizer.cpp` file uses `legacy::FunctionPassManager`, but you can add a `legacy:PassManager` to support interprocedural optimizations.

An interesting problem in optimization is referred to as the *phase ordering problem*; though it should probably be called the *pass* ordering problem. When you add passes to the function pass manager and then call `run` the passes are applied in the order you add

them. Different orders can result in different results; experiment with this to see for yourself. It is hard to determine an optimal order, since it will depend on the program structure. Fortunately, `run` returns a Boolean that indicates whether any optimization has yielded a change in the function. You can use this to trigger another round of optimization. In doing this, you should convince yourself that this process will terminate.

Note: we have run into trouble with the `run` method for `legacy:PassManager` returning `true` when really there is no change to the module produced by running the configured optimizations. This is annoying because you cannot simply use the return value to continue to apply optimizations until you reach a fixpoint. A reasonable solution is to pass a commandline option that lets you control the number of times you choose to iterate the application of optimizations.

The function pass manager and the IPO pass manager are distinct. Here again you can consider the possibility of phase ordering and running multiple rounds of optimization with each manager – convincing yourself of termination again.