CS5130 Project 2: Value Numbering

Due: Monday, Oct. 10 @ 5:00 PM

We will continue basing our project on the Low Level Virtual Machine (LLVM) compiler infrastructure. Please make sure you are working on LLVM 14.0.6 as we did in Project 1. In this project you will implement and compare three value numbering techniques, local value numbering (LVN), super local value numbering (SVN) and dominator-tree-based global value numbering (GVN).

1 Project Summary

GVN has already been implemented in LLVM (see `$(llvm)/lib/Transforms/Scalar/GVN.cpp`). Note that GVN subsumes SVN, which, in turn, subsumes LVN. You need to update the GVN code so that it can be downgraded to perform LVN or SVN only. To implement super local value numbering, you are asked to implement an analysis pass that identifies extended basic blocks. For simplicity, you can embed your EBB pass inside GVN.cpp.

2 Implementation

You will need to introduce two new command line options for `opt` which suggest the optimizer to perform LVN or SVN instead of GVN. I will use command line option `-gvn` to trigger global value numbering. You should add the following two options to `opt`:

1. `-lvn`: perform local value numbering only.
2. `-svn`: perform super local value numbering only and output EBBs for each function if this option is enabled.

I will use "opt -gvn -lvn" to test your implementation of local value numbering. Similarly, "opt -gvn -svn" is for super local value numbering. Most of your work is to understand GVN.cpp and make changes to integrate local and super local value numbering into it. The command line option "-stats" can be used to check the statistics of "opt" including the number of instructions deleted by "gvn". Note that for "-stats" to be effective, you will need to create a "Debug" build or a "Release" build with "ENABLE_LLVM_ASSERTIONS" turned on, i.e., add -DENABLE_LLVM_ASSERTIONS=1 as a cmake option.

2.1 Testing

1. GVN, SVN and LVN Testing

To focus on testing the effects of GVN, SVN and LVN, you are recommended to disable partial redundancy elimination in GVN.cpp.

We will use `test-suite/SingleSource/Benchmarks/Stanford` as the testing benchmark suite. Your optimization should pass all the benchmarks in this suite. You need to collect the number of instructions deleted by value numbering. Note that the LLVM command `llc` can convert bitcode to assembly code. You can use `gcc` to compile the generated assembly to executable. You should use `gcc` to compile the original source code and compare the output with the one by your optimized code to ensure your optimization is correct.

I have attached a hand-coded test case, vn.ll, for your convenience. You should expect different numbers of instructions deleted for this test case.

2. Extended Basic Block Testing
   Dump all the extended basic blocks in each function led by the function name followed by a colon. Each EBB should be output as a pre-order sequence of the basic blocks in it enclosed by a pair of curly brackets and each basic block can be represented by its name or the label of the lead instruction. The sample output of EBBs for the bitcode in Project 1 would be as following:

```
erk:
{%2}
{%9, %13, %21, %24}
```

3 Submission

Tar all the source files you have changed including their directories into a tar ball, project2.tar, and submit it through Canvas. Write a report that summarizes your implementation and analyze your experimental results for the `Stanford` benchmark suite.

Please also attach your EBB outputs and name them after the original input file names with .ebb extension. For example, the EBB output for Bubblesort.c should be stored in file Bubblesort.ebb.