CS 5130
Project 1
Robbie Watling

About Project 1

This project performs an LLVM pass to perform basic block analysis through the generation of histograms. This is done by running Clang on a given C program. We pass an optimization level and instruct Clang to produce bit code that will be used by LLVM in the next step. We then run the LLVM disassembled which translates the byte code into that is a human readable LLVM assembly file which is an intermediate representation of our original C program. We then perform a pass this representation and generate histograms of the number of lines in each basic block for each function. Additionally we generate histograms of the number of lines that contain a load instruction in each basic block for each function. Following this we display the average number of lines in a basic block for all of the instructions as well as the average number of lines in a basic block for the load instructions.

Implementation

The LLVM pass is performed in a C++ program called `BasicBlockHist`. This program borrows much of its code from `Hello` program as instructed by the project description. The Hello program utilizes the `runOnFunction` procedure and prints a message to standard error when it encounters a new function. This implementation separated the `runOnFunction` procedure definition and implementation for readability. This function generates the information requested by the project description.

The information that is printed was obtained by using the iterators supplied by the LLVM source tree. In particular, the LLVM Function Class' Basic Block iterator and the LLVM Basic Block Class' instruction iterator. This allows us to create a nested for loop to iterate over all of the instructions. In this for loop we also check for loads by utilizing the LLVM Instruction Class' `getOpcode()` function to obtain the instruction type. We compare this instruction type to the load instruction class which performs the comparison as expected due to the clever design of the LLVM source code.

The `BasicBlockHist` version of `runOnFunction` utilizes integer arrays to keep track of the counts of each size of basic block encountered in a function. We do this by using the size of the basic block as the index to the array which is observed by `allCount` for all instructions and `loadCount` for load instruction. These arrays are of size 1024 as defined by a constant at the top of the program `BB_MAX_INSTS`. This definition could be problematic if the number of lines in a basic block is greater than 1024 as we would access an index that is out of bounds. Originally, when we had the constant defined as 64 we encountered this problem on the Stanford test benchmark `Oscar`. For this project, our constant definition of 1024 is sufficient, but in a more high stakes scenario the use of vectors found in C++ might be better. However it should be noted that the arrays used in this project have significantly less overhead.

Additionally, in `runOnFunction`, the sum of the number of lines is calculated for all instructions and load instructions through `allSum` and `loadSum` respectively. Then the

average block sizes can be calculated by dividing the sum by the total. We are then able to print the histograms and averages.

Analysis of Clang Options and Basic Block Size

Below is a summary of the effects of the optimization level on the basic block size for the Stanford benchmarks from the LLVM test suite.

Bubblesort

- Optimization level 3 output often has smaller blocks for all types of instructions and loads.
- The functions `bInitArr` and `Bubble` have a block of size 21 when the optimization level is 0, but the largest block size for those functions at optimization level 3 is 13. It also appears the the number of loads are significantly reduced so some array optimization likely took place.
- Main has larger block sizes at optimization level 3. This seems counterintuitive but assuming a higher level of optimization is better, the hypothesis is that the calling structure of the program better accounted for at the higher optimization level. This may be because main is often called once and `Bubblesort` could or is called several times.

Initrand

- Initializations are sometimes larger at optimization level three but the operating functions often exhibit less loads. As with the previous benchmark. This may be because the higher optimization level understands the relationship between function calls and their respective memory accesses.

IntMM

- In `Initmatrix` the number of instructions in blocks at optimization level 3 is larger than that of optimization level 0. However, it has only one load compared to optimization level 0. Optimization level 0 has significantly more load blocks.
- The operating functions (functions assumed to be more computational) have significantly smaller blocks.

Oscar

- Many functions at optimization level 0 have smaller blocks.
- It is difficult to say whether higher optimization level improves code's performance for this benchmark as the block sizes are smaller for optimization level 0 in some cases and smaller for optimization level 3 in others. The better performing code likely has smaller block size on frequently performed functions.

Perm

- The `Main` function has much larger block size which is strange for optimization level 3. This may be due to some sort of unrolling optimization.
- Many of the functions have smaller block sizes for optimization level 3.

Puzzle

- Many of the called functions are more optimal for optimization level 3, especially memory intensive functions (many load instructions) from a glance.
- Similar to the main function in Perm, Puzzle's puzzle function and main have some large blocks at optimization level 3

Queens

- Overall optimization level 3 significantly reduced all types of instructions.

Quicksort

- Optimization level 3 generally reduced loads.

RealMM

- The inner product function appears to be more optimal at optimization level 3.
- The `Mm` function is better at optimization level 0.
- Experimental analysis would be necessary to determine which optimization would be more desirable.

Towers

- The number of loads in each block is significantly reduced for optimization level 3.

Treesort

- The number of loads in each block is significantly reduced for optimization level 3.

Summary fo Clang Options and Basic Block Size

Generally optimization level 3 appears to be more optimal. Several basic optimizations were observed at optimization level 3. These optimizations likely included function inling, loop unrolling, and multiplication substitution. The most certain and obvious optimization was the reduction of load instructions. The load reductions make much sense because memory accesses often take several more cycles than regular instructions. There are some cases in which optimization level 0 reduced the block sizes compared to optimization level 3, but it is difficult to say whether this is more optimal. In the future, it would be worthwhile to compare the execution times of the programs at the optimization levels to more completely determine which optimization level is the most optimal for the Clang compiler.