CS 5130
Project 2
Robbie Watling

About Project 2

This project performs different value numbering optimizations with the LLVM compiler infrastructure. Value numbering is an optimization that can reduce the number of instructions by eliminating unnecessary expressions. Expression operands are enumerated as values and entered into a table with their operation and assignment. If an expression is already in the table upon insertion, it is redundant. Global value numbering considers the expressions found anywhere in the program. Local value numbering considers the expressions only in a single basic block. Super-local value numbering analyzes the control flow graph of basic blocks to create extended basic blocks which then determine the context of expressions. Extended basic blocks are a finite set of basic blocks B_1, B_2, …, B_n such that B_1 may have multiple predecessors but all other blocks in the set have only one predecessor, and there is a path from B_1 to B_k only going through other nodes in the extended basic block. Global value numbering is expected to be the best optimization strategy as more expressions are available. Super-local has the second most expressions available to it so we expect those expressions to be available. Local value numbering has the least expressions available therefore fewer redundancies can be found.

Implementation

The implementation of these value numbering schemes are modifications to `GVN.cpp` in the LLVM infrastructure. `GVN.cpp` is LLVM's implementation of global value numbering so no modifications were necessary. For SVN and LVN clang options were added in the same style as the other options already present. Additionally the options had associated global variables for convenience. The options were considered at `iterateOnFunction` and `processBlock` in `GVN.cpp` as the value table needed to be adjusted at those locations to accommodate a specified value numbering scheme.

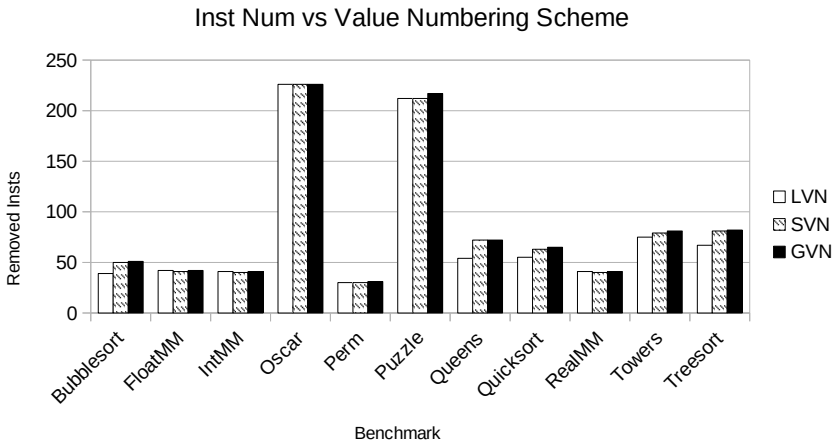
For local value numbering, each block cleared the table using a provided function `cleanupGlobalSets.` When the `processBlock` function was called, cleanupGlobalSets removed information in the value table structure and therefore allowed each block to only consider expressions locally as desired.

For super-local value numbering, each time a function was called, the value table was cleared using `cleanupGlobalSets`. Then the basic blocks of the function were iterated on to obtain the extended basic blocks. The extended basic blocks were determined by using the predecessor size function to satisfy the extended basic block criteria. For simplicity, the blocks were placed in a two dimensional array of a large size with the rows corresponding to a set of extended basic blocks. An attempt at putting these basic blocks into a vector of vectors found in the standard C++ library was made. However there were issues resolving pointers and stack size for this approach which was problematic. In the implemented two dimensional array, each basic block in the extended basic block was passed into the `processBlock` function. In practice, the dominating relationship between individual blocks and their expressions would lead to the popping of instructions in the value table. However, the value table lookup determines if expressions are reachable as it has an associated dominator tree. Therefore this popping of instructions was unnecessary other than between function calls

(same as global value numbering). However, if these assumptions are incorrect, there appear to be utility functions in the value table. These utilities could remove instructions identified in a non-dominating relationship for an expression between blocks in an extended basic block.

Results

Benchmark	LVN	SVN	GVN
Bubblesort	39	50	51
FloatMM	42	41	42
IntMM	41	40	41
Oscar	226	226	226
Perm	30	30	31
Puzzle	212	212	217
Queens	54	72	72
Quicksort	55	63	65
RealMM	41	40	41
Towers	75	79	81
Treesort	67	81	82



The value numbering schemes matched the hypothesis that global value numbering would eliminate the most instructions. However, it was only slightly better than local value numbering and super-local value numbering.

In most cases, local value numbering eliminated the fewest instructions. However, in a few cases it was better than super-local or tied global value numbering (RealMM and IntMM). It may be the case that our method of clearing the value table for the super-local value numbering may have had unintended side effects. It is also likely that some of the other options in `GVN.cpp`. Given that this was the case for two similar benchmarks, RealMM and IntMM this could be a result of the benchmark's representation.

The super-local value numbering tended to rival the global value numbering. Thus extended basic blocks must catch a large amount of between basic block redundancies at a similar rate to global value numbering. Since super-local value numbering is comparable to global value numbering, if a different compiler was engineered, and it was the case that super-local value numbering was easier to implement, it would potentially be worth implementing the super-local value numbering and foregoing the global value numbering.

Overall global value numbering always eliminated the most instructions. This follows our hypothesis that more redundancies are eliminated because global value numbering has more expressions available to it.

Appendix

Experimental analysis was performed in `run_proj2.sh`, the clang options were necessary for output to be written using the release build.