Ally Dinhofer, Adam Stewart, Emily Howell **Project**
Email: abd54@case.edu ID: 3379570
Email: axs1477@case.edu ID: 3386782
Email: eth35@case.edu ID: 3401809
Course: CSDS 337 - Compiler Design Term: Spring 2021
Instructor: Dr. Vipin Chaudhary Due Date: 9th May, 2021

Number of hours delay for this Project: 0
Cumulative number of hours delay so far for each member: 0, 25, 72

Project Team Members: Ally Dinhofer, Adam Stewart, Emily Howell

---

Problem 1
Write a C or C++ program to showcase some of the features of Clang and LLVM. You want to write a C program that will best showcase the features of various optimizations. Use time function inside the code to measure program time for performance measurements. This is open ended project but minimum expected effort includes:

1. For a given architecture, compare the time it takes for different types of compiler optimization. Also use compiler option to minimize code size. Compare the codes (target assembly code) for all cases to isolate the types of optimizations implemented.
2. Show at least three different optimizations in your code that are affected by compiler optimizations.

Choice of the C/C++ code will have an impact on the grades. Indicate where such code could be used or use code for known algorithms (sorting, searching, numerical computation, etc.).

1. Bonus (upto 33% based on quality and quantity): Use as many LLVM Analysis, Transform (most important), and Utility Passes (https://llvm.org/docs/Passes.html) that are applicable to your example with complete makefile (or command line script) for the passes and appropriate documentation of results. You may also want to refer to clang.llvm.org

---

Deliverables: A zip file containing

- **File with your code**
- **README text file with directions to run the various programs**
- **Report showing original code and optimized code snippets and the particular optimization used; results (table of performance). Similarily for the bonus portion.**
  - **Full names and Case IDs**
  - **(not required) any special notes about your implementation the grader should be aware of**

Tip: An app such as Notepad++ can be used to create .t files which you can use to test your program in the test folder.

Out C++ Program

We put together a C++ program that takes an input of integers and sorts them using bubble sort. We created three versions of the program that each take the input of integers in different ways. The first used basic standard input std::cin, the second used command inputs, and the last one read the input integers from a file. We choose to use the file that reads the input integers from the file for this project because it could be tested easily using a Makefile, unlike the standard input option, and had a slightly more complicated code-base because it parses the integers apart, unlike the command line option.

For the timing aspect of this project, we used the high resolution clock from chrono to get accurate time stamps before and after the parsing ans sorting run.

The above options can be seen and run from the Makefile included in this submission. The standard input option can be called using *make toy_input.o*, the command line option using *make toy_command.o*, and the file input option that was used for the testing of this project can be run using the *make toy_file.o* command.

Optimization Passes:
We experimented around with several different optimization options. On its own, converting the code to the Assembly code had a negative impact on the run time. Similarly, the intermediate representation also had a negative impact on the outcome of the program. However, with a little research we found some additional optimization tags that we could use to improve the complexity of the assembly code and the final run time. We tried a number of tags such as -mem2reg and -reg2mem which had negligible improvements on the number of lines in the assembly file or on the run time of the program. The tags that we discovered had the most positive impact were the -01 through -0z tags related to code optimization at the clang level. The lowest level of optimization, -O1, reduced the number of lines in the assembly file from 3742 to 3724. The higher level of optimization, -Oz, was much more aggressive and reduced the number of lines in the assembly file from 3742 to just 1037. In addition to the reduction of number of lines of code in the resulting assembly file, there were also improvements to the runtimes. We tested the runtimes for this project on two separate machines and for each machine ran the programs multiple times and averaged the runtimes between each test for the bellow results.

On the first machine we used for this project, a PC running Windows 10 Home and with a Intel 10th Gen core i7 1065G7 @ 1.30 Ghz processor, we got the following average times:

1. Base Time: 185ms not using assembly code
2. Assembly Code time: 207ms
3. Using Intermediate Representation: 221ms
4. Opt 1 - with the -O1 tag: 49ms
5. Opt 2 - with the -Oz tag: 34ms
6. -mem2reg: 201ms
7. -reg2mem: 204ms

On the second machine we used for this project, a PC with a virtual machine running the provided virtual box image running Ubuntu 20.04.1 LTS and with a Intel 10th Gen core i7 8550U @ 1.80 Ghz processor, we got the following average times:

1. Base Time: 285ms not using assembly code
2. Assembly Code time: 329ms
3. Using Intermediate Representation: 382ms
4. Opt 1 - with the -O1 tag: 319ms
5. Opt 2 - with the -Oz tag: 72ms

6. -mem2reg: 319ms

7. -reg2mem: 321ms

The above options can be seen and run from the Makefile included in this submission. The base time option is the *make toy_file.o* option previously mentioned. Then the assembly option is *make code* and the intermediate representation is *make ir*. Finally, the first optimization with the -O1 tag can be called using *make opt*1 and the second optimization with the -Oz tag can be called using the *make opt*2 command.

These optimizations in our code are clearly affected by compiler optimizations since they change the time taken to run the program, which in our case, is sorting the integers. Further, all the differences in the files can be seen in the *diff* directory output when you run *make diff*. In our submission we have included the diff files that were generated when we ran the Makefile ourselves.

EC - Analysis Passes:
Analysis passes generate outputs, either inline or file generation, that provide information that can be used to analyze the performance of the program. We tested all of the analysis pass options listed on (https://llvm.org/docs/Passes.html) and found that including the tag for a given analysis pass resultied in three different outcomes:

1. The pass resulted in a data output either inline or in a file

2. The pass ran but did not generate any information

3. When executed, the computer flagged that the tag was not applicable and was ignored

Below we have listed out and explained the tags that fell into the first two of the categories. We also created an easy way to run all of the optimizations that fall into category 1 all at once using *make analy*.

Passes that produce an output:

- -dot-callgraph
  Prints the call graph into a .dot graph. We tried to use the "dot" tool to convert it a .png but it said the file was damaged.

- -dot-cfg
  Prints a control flow graph into a .dot file. We tried to use the "dot" tool to convert it a .png but it said the file was damaged.

- -dot-cfg-only
  Prints a control flow graph, without the function bodies, into a .dot file. We tried to use the "dot" tool to convert it a .png but it said the file was damaged.

- -dot-dom
  Prints the dominator tree into a .dot graph. We tried to use the "dot" tool to convert it a .png but it said the file was damaged.

- -postdomtree
  A post-dominator construction algorithm for finding post-dominators.

- -print-alias-sets
  Prints the alias sets of the code.

- -print-callgraph-sccs
  Prints the SCCs of the call graph to standard error.

- -print-callgraph
  Prints the call graph to standard error.

- -print-cfg-sccs
  Prints the SCCs of each function CFG to standard error.

- -print-dom-info
  Prints the dominator information.

- -print-function
  Prints the functions of the module as they are processed.

- -aa-eval
  For each function in the program, it simply queries to see how the alias analysis implementation answers alias queries between each pair of pointers in the function.

Run but didn't output anything:

- -da
  Detects dependencies in memory access.

- -domfrontier
  A dominator construction algorithm for finding forward dominator frontiers.

- -domtree
  A dominator construction algorithm for finding forward dominators.

- -intervals
  Represents the interval partition of a function or any preexisting interval partitions which can be used to reduce the flow graph down to its degenerate single node interval partition.

- -lazy-value-info
  Interface for lazy computation of value constraint information.

- -lint
  Checks for common constructs that are easily-identifiable which which produce undefined and unintended behavior in LLVM IR. This pass is not comprehensive, it is just identifying the common ones. Additionally, it can only check the common constructs that can be checked statically.

- -memdep
  Determines what preceding memory operations a given memory operation is dependent on.

- -module-debuginfo
  Decodes and prints the debug info metadata.

- -regions
  A region is any sub graph that is connected to the remaining graph in only two spots. This pass detects the single entry/single exit regions and builds a hierarchical region tree of them.

- -scalar-evolution
  Analyzes and categorizes the scalar expressions in loops.
- -scev-aa
  Tests for the dependencies within a single iteration of a loop rather than between different iterations.

- -stack-safety
  Determines is variables are at risk for memeory access bugs.

- -instcount
  Collects the count of all instructions and reports them.

- -intervals
  Calculates and represents the interval partition of a function, or a preexisting interval partition.

- -iv-users
  Bookkeeping for "interesting" users of expressions computed from induction variables.

EC - Transform Passes:
Transform passes impact the resulting assembly code and are by far the most important of the three types of passes listed in this extra credit section when it comes to improving the run time of a program. We tested all of the transform pass options listed on (https://llvm.org/docs/Passes.html) and found that including the tag for a given transfor pass resultied in three different outcomes:

1. The pass added further optimization and somehow altered the generated assembly code.
2. The pass ran but did not alter the generated assmbly code at all and therefore caused not optimizations.
3. When executed, the computer flagged that the tag was not applicable and was ignored

Below we have listed out and explained the tags that fell into the first two of the categories. We also created an easy way to run all of the optimizations that fall into category 1 all at once using *make transf*.

Adds further optimization:

- -strip-nondebug
  Implements code stripping, specifically deleting names for virtual registers symbols for internal globals, and functions debug information. This transform pass did not add or remove any lines of code but did cause some minor changes to the file.

- -tailcallelim
  Transforms calls of the current function (self recursion) followed by a return instruction with a branch to the entry of the function, creating a loop. This transform pass decreased the number of lines in the output assembly code from 3742 lines to 3680 lines.

- -sink
  Moves instructions into successor blocks so they aren't executed if not needed for the result of the current path. This transform pass increased the number of lines in the output assembly code from 3742 lines to 3744 lines.

- -strip
  Deletes names for virtual registers, symbols for internal global variables and functions, and debug information to reduce the code size. This transform pass did not add or remove any lines of code but did cause some minor changes to the file.

- -lowerinvoke
  Designed for use by code generators which do not yet support stack unwinding. Converts invoke instructions to call instructions, so that any exception-handling landingpad blocks become dead code. This transform pass decreased the number of lines in the output assembly code from 3742 lines to 3423 lines.

- -sroa
  Breaks up alloca instructions of aggregate type into individual alloca instructions for each member. Then it transforms the individual alloca instructions into scalar SSA form. This transform pass decreased the number of lines in the output assembly code from 3742 lines to 3570 lines.

- -licm
  Attempts to remove as much code from the body of a loop as possible. It does this by either hoisting code into the preheader block, or by sinking code to the exit blocks if it is safe. This transform pass increased the number of lines in the output assembly code from 3742 lines to 3744 lines.

- -loop-rotate
  Performs a simple loop rotation. This transform pass increased the number of lines in the output assembly code from 3742 lines to 3761 lines.

Makes no difference:
The following transform passes were tested and, while they did run, they were found to make no impact on the resulting code.

- -loop-unroll-and-jam
  Unrolls the outer loop and "jams" (fuses) the inner loops into one.

- -sccp
  SCCP stands for sparse conditional constant propagation which means that values are assumed and proven to be constant, assumes Basic Blocks are dead unless proven otherwise, and proves that specific conditional branches are unconditional. In simpler terms, this is removing unneeded code and unused paths.

- -memcpyopt
  Performs various transformations related to eliminating memcpy calls or transforming sets of stores into memsets.

- -loweratomic
  Lowers atomic intrinsics to non-atomic form for use in a known non-preemptible environment.

- -strip-dead-prototypes
  Loops over all of the functions in the input module, looking for dead declarations and removes them.

- -mergereturn
  Ensure that functions have at most one ret instruction in them. It also keeps track of which node is the new exit node of the CFG.

- -jump-threading
  Tries to find distinct threads of control flow running through a basic block.

- -loop-reduce
  Performs a strength reduction on array references inside loops that have the loop increment variable as one or more of their components.

- -loop-simplify
  Performs several transformations to transform natural loops into a simpler form, which makes subsequent analyses and transformations simpler and more effective.

- -loop-unroll
  Implements a simple loop unroller.

- -loop-unswitch
  Transforms loops that contain branches on loop-invariant conditions to have multiple loops.

EC - Utility Passes:
We attempted to use the Utility Passes but found that non of them would work because the GraphViz tool was nonfunctional on our machine. If these were to work they would output control flow graphs, dominator trees, or post dominate trees using the GraphViz tool. Because these option all simply add the functionality of displaying a visual, none of them would not reduce the code size of the compiled code and therefore do not seem applicable to this projects goal.

EC - End Notes:
We also created a Makefile option that runs all of the category 1 analysis tags and all of the category 1 transform tags. This is the *make all* option. One thing we noticed was that there are fewere .dot files generated by the analysis tags when the code has been optimized. Below is the changes in runtime we got from each section of the extra credit project.

On the first machine we used for this project, a PC running Windows 10 Home and with a Intel 10th Gen core i7 1065G7 @ 1.30 Ghz processor, we got the following average times:

1. Base Time: 185ms not using assembly code

2. Assembly Code time: 207ms
3. Analysis Passes: 256ms
4. Transform Passes: 292ms
5. Utility Passes: 247ms
6. All Analysis, Transform, and Utility Passes: 51ms

On the second machine we used for this project, a PC with a virtual machine running the provided virtual box image running Ubuntu 20.04.1 LTS and with a Intel 10th Gen core i7 8550U @ 1.80 Ghz processor, we got the following average times:

1. Base Time: 285ms not using assembly code
2. Assembly Code time: 329ms
3. Analysis Passes: 413ms
4. Transform Passes: 493ms
5. Utility Passes: 405ms
6. All Analysis, Transform, and Utility Passes: 82ms

Obviously, none of the additional tags improve the runtime of the resulting code. While they do provide additional information and can make the generated assembly code base smaller, the cost is runtime.