Final Project Documentation

Shane Skikne Yuzhong Huang Ziyi Lan

What did you do?

In this project we gained an indepth understanding of the compilation process and implemented part of a local optimizer in python.

We spent the first week learning the main components of a compiler and going through the compiling process to see how C codes get translated to machine code. Then, each member of the team member did in-depth research on a specific part of compiler he finds interesting and shared shared with the team. In the end we chose to implement the local optimizer to further understand how optimization in a compiler works.

For the local optimizer we mainly focused on five parts: single assignment form, algebraic simplification, constant folding, common subexpression elimination, copy propagation and dead code elimination with each part serving a function. We then compared the running time of the code before and after the optimization to see how much does our script improve the efficiency of the program.

Why did you do it?

Researching in compiler construction and optimization and building a local optimizer gave us the chance to learn how higher level programming languages get translated into low level machine code. Understanding what is inside the black box not only helps us to write better code, but also allows us to make connections between computer architecture and high level coding.

How did you do it?

Research

For the first week of the project time, we did a research about compilers and learned how they convert code to assembly language. We followed Coursera's course: Compiler to familiarize ourselves with core phases in the compiling process: Lexical Analysis, Parsing, Semantic Analysis, Optimization and Code generation. Our team gained a general understanding about compiler afterwards. Then, each of us chose a topic that was most interesting to us in the process and did in-depth research about it. We explored type checking, local optimizer and

peephole optimizer during that time. Discussing our interests, we realized local optimizer not only interested us all, but it fit in the scope of our project as well. As a result, we decided to build a local optimizer.

Local Optimizer

For the second half of the project, we wrote a local optimizer for C. The optimizer is consisted of six functions, Single Assignment Form, Algebraic Simplification, Constant Folding, Common Subexpression Elimination, Copy Propagation and Dead Code Elimination. Each of them is described below:

- **Single Assignment Form**
  Before optimizing a basic block, the compiler will first go through this function to make sure each register is at most assigned once. As a result, each variable will have one constant value throughout the basic block.

- **Algebraic Simplification**
  In this stage, the compiler simplifies some basic statements, such as “a * 1”, “a * 0”, “a + 0” and etc. Also, while going through the script, the compiler also transforms ‘expensive’ operations’ to less time consuming operation according to the rule power > multiplication > shift > constant/ plus/ subtraction.

- **Constant Folding**
  Operations on constant are computed in this function. By doing so, we can propagate some code with constants and therefore make it easier to discover unused statements. Constant folding itself doesn’t improve the code in a large scale, but provide a good tool for dead code elimination.

- **Common Subexpression Elimination**
  This function deals with multiple statements that have the same right-hand side and replace them with the first variable assigned for this value.

- **Copy Propagation**
  If assignment like “y = x” is appeared in the code, and y is later used in other assignments, y can simply be substituted by x and delete the y from the block. Copy propagation do not improve the code itself. It usually works with constant folding and dead code elimination.

- **Dead Code Elimination**
  As its name suggests, this function eliminates the lines of code that are not relating to the output of the code block. Dead code is usually generated during the constant folding and copy propagation phases, but it may also exist in the original source code(situations

like debugging code or library using). Removing dead code will save the processor from executing this useless instructions and improve caching effect by improving spatial locality.

Result Analysis

We tested our improved code against the raw file by writing a wrapper file in C to measure the code execution time of each single block.

Specifically, we added the raw file to the wrapper to generate an executable named “raw”; copied optimized code to the wrapper to generate an executable named “opti”. We then run the two executables. Each files printed the average time it takes to run.

Below is a chart of data we collected from 5 showcases we’ve created to show the improvement.

	Raw Time (s)	Optimize Time (s)	% Improvement
Multiply and Divide	0.000096	0.000088	91.6%
Diff of Fourth	0.000025	0.000021	84.0%
Heron	0.00004	0.000033	82.5%
Law of Cosines	0.000038	0.000032	84.2%
Three Squares	0.000022	0.000019	86.3%
Average	0.0000442	0.0000386	85.7%

As we can see, generally, it takes about 85.7 percents of the execution time of the raw file for the optimized code to execute, while returning the same result.

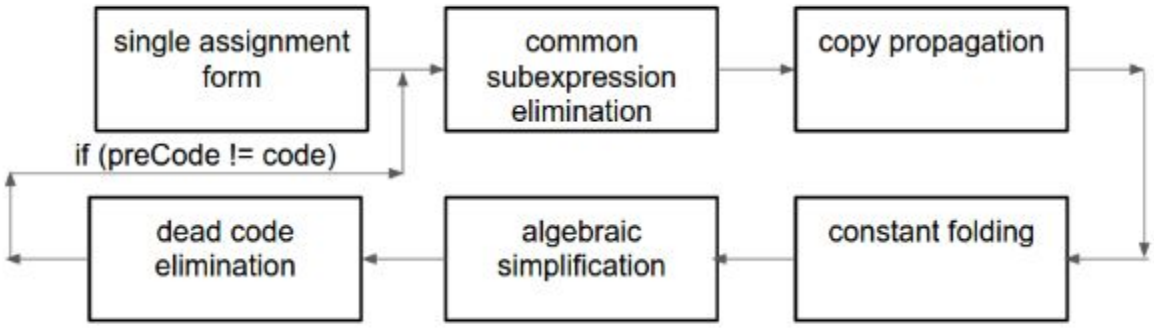
How can someone else build on it?

Code

All of our code can be found in https://github.com/Skinc/C_Compiler
The important files to know are:
local_optimizer.py - This file includes our local_optimizer class. To optimize a file, see build instructions below. Edit this file to change our code is optimized.
tree.py - This file includes the tree class, which creates binary trees. This is used for our syntax tree.
wrapper_open.txt and **wrapper_close.txt** - These files are used to write the beginning and end of our optimized C files. Edit this file to change code in all the optimized C files.
rawcode Folder - This is the folder where the optimizer looks for rawcode to optimize. Put code you’d like to optimize here.

optimize Folder - This is the folder where optimized code is.
expected Folder - This is the folder where the expected version of output code is. When test() is run, the the compiler looks for the expected output here.

Schematics



Build instructions

- **To get optimized file**
    1. Put the code block (.txt format) to be optimised into the /rawcode folder.
    2. Change the file name in “Main: ” in local_optimizer.py.
    3. Find the result in folder /optimized.

- **To see how much the local optimizer has improved**
    1. Get an executable for the original file (can use the wrapper file if you don’t)
    2. Run the executable and record the printed execution time 1
    3. Run in the command line “gcc -O0 -o opti ./optimize/<-filename->”, where filename is the corresponding improved file in “./optimize/”
    4. Run “opti” and record the printed execution time 2
    5. Compare execution time 2 with execution time 1 to see how much is the original code improved

A list of difficulties and ‘gotchas’ while doing this project

The one major challenge we faced was defining our intermediate language. At first, we all had different understandings of what code our optimiser would take in. With different understandings, we all wrote code that could handle different input. While the team had

discussed the intermediate language and thought we were on the same page, it became very clear that we weren't. Actually defining this language was quite simple; it was simply a matter of realizing we had to. Someone continuing this project would need to understand our intermediate language and decide to either continue using it or change our code to handle a different language.
Besides that, most of the difficulties were avoided by planning well at the beginning and having the optimizer broken into appropriately bite-sized chunks. We are sure our optimizer could be much faster, but not without a loss of simplicity.

Work Plan reflection

Throughout the project, we followed our plan well and evenly distributed the workload on the two week span. We anticipated potential challenges and risks at the start of the project and in the end managed to tackle many of the tasks in a reasonable amount of time.

Possible Extensions to the Project

First, we could further improve the algebraic simplification function so that it can deal with some more complicated calculation, such as power or multiplying a number that is not 2's exponential.

Second, we could add Peephole Optimization, a process that happens after code generation. It has a lookup table function which maps certain statement to certain optimized equivalent statement. For example, "addiu $a $b 0" will be mapped to "move $a $b".

Finally, another possible direction to take this project in would be to have the local optimiser optimize a more complex intermediate language. While the intermediate language we designed could be used as C, it was very simple and limited. Expanding that language would allow the optimiser to optimize more effectively.