Summary

The goal of this project is to create a calculator compiler frontend, backend, data dependency analyser and an optimisation. The issue I had focused on primarily in this project was Common Subexpression Elimination (CSE) to improve code efficiency. As done in the previous lab, I have made use of 3-nested if/else statements. The data dependency analysis can accurately identify flow, anti and write dependence between lines and complements the if-else nested loops. I have noticed that the optimisations achieve a speedup of around 34% on average. It is done so by eliminating unnecessary variables, redundant mathematical operations and dead variables.

Implementation

As done in the previous lab, once the TAC line is generated, the data is sent to the data dependency analyser where the three types of dependencies are detected and printed out. The data is then sent over to the optimisation module for CSE, where the initial unoptimised TAC code is fed as the input. The backend produces the C versions of the unoptimised/optimised TAC. The flex and bison rules used in other labs have been altered to have added calls for new modules which is required by this project.

As we know, the frontend's responsibility is to convert the input program into TAC lines, the flex rules must tokenise the program and the bison rules apply to the grammar used and checks for any errors. As for the data dependency module, it checks dependencies for each variable in TAC by looking backward into the previously recorded lines. If the variable is written to or from TAC, subsequently checks for the three types of dependencies. Flow or write dependencies are found outside the nested loops. Once all the dependencies are found, the control is returned to the compiler frontend to produce the next TAC line. For accurately determining the dependencies, the algorithm looks backward into the processed code.

A pivotal aspect of this analysis is that it can find dependencies in the if/else statement path flows. It does so by going backward from the current line. The tracking mechanism that has been used to keep track of the if/else depth each variable is set in, helping with determining paths of execution and blocking dependencies. The data structure management used is quite simple and is easy to workaround.

Compiler Subexpression Elimination (CSE)

The aim of CSE is to remove redundant operations and reduce the number of variables that required to be in the registers. CSE is carried out once the frontend TAC has been generated and the dependencies have been found. Its goal is to improve performance of the initial program while retaining its functionality. This is achieved by computing a CS and storing it in a new temporary variable `(_c# = a op b;)`. It then substitutes every future use of the subexpression with this variable until the original values are written to. They are recognised in the form of b op c, where b and c can be the combination of any constants.

This can help us save the cost of recomputing operations while increasing performance. However, the program is not quite intuitive when it comes to deciding to eliminate which subexpression. The CSE optimisation reads in a TAC line, looks for the subexpression and if the subexpression hasn't been encountered before, looks ahead and makes sure if there any future uses before it is completely invalidated. If there is one more use of it prior to the invalidation, it is recorded in the subexpression table. Reusing its own temporary variables is to prevent stacking of subexpression assignments which would result in better performance and efficiency. If any variable is written to after the expression is saved to the temporary variable, the expression becomes invalid.

Optimisation: Heuristic

As mentioned in the project description, CSE is not always beneficial. That is, in some cases, it helps performance, while in others, it may actually hurt the performance. This is the very reason why we have to carry out a heuristic optimisation. It must help optimisations run and achieve a positive performance. The primary objective of a CSE is to prevent unnecessary overhead of adding a temporary variable that can only be used once.

The heuristic allows CSE to reduce the number of variables used within several parts of the program, subsequently reducing the register pressure and increasing performance since there is a lower chance of spilling. The tradeoff here is that CSE will increase the code size with the addition of temporary variables. The CSE will return the number of changes it made back to the main loop. When no changes can be made, it will return to the value of 0.

Conclusion

Over the course of this project, I have learnt a great deal about perfecting the first three labs to make the program more efficient by taking up less computational power. I have noticed a considerable amount of speedup of which the results I will be attaching with the code. I've learnt that the other benefit of optimisation is the reduction of the code length. Since the first lab, one of the shortcoming of the project is the smaller buffer sizes which I plan on addressing in the future as I work more on this. I do believe that the compiler operates accurately with no errors. Also, the CSE algorithm I have put in is not based off on the strength of the operations and only considers the re-usage of the subexpressions. Of all the labs, this project was the most time consuming as it took me weeks to essentially implement it.