Compiler Middle-End Optimizer Design

Yunjie Zhang Minh Ho Kevin Lou

September 17, 2025

Optimizer Design

High-Level

In order to solve our middle end optimizer problem, we implemented a dead code elimination algorithm dependent on reaching definitions for a def-use chain. Importantly, we converted the IR into a graph based representation, a CFG. The overall algorithm is as follows:

The algorithm first parses the IR, and then the optimizer iterates through every function in the program to build a Control Flow Graph. Building the CFG is necessary for doing any form of data flow analysis. In the process, the data flow analysis algorithm iterates through each block's GEN and KILL sets and updates the IN and OUT sets accordingly. It loops through this process until it reaches a point where none of the OUT sets have changed after one complete pass. Then the optimization pass occurs, where a mark-sweep algorithm that utilizes reaching definitions marks all critical instructions and iteratively marks any define instructions that the critical instructions depend on until the worklist set is empty. Finally, the code is reconstructed, leaving out all of the dead code.

Low-Level

We decided to choose C++ as our language because it provides high-performance and low-level control. Also, our team members had more experience and preferred C++ over Java, and C++ is the industry standard for building high performance compilers for clang and gcc. C++ offers an extensive standard library of pointer logic, data structures, and the ability to implement OOP based strategies.

For example, we applied C++ standard library's associative array/hashmap to map the final IN and OUT sets to a specific block in the CFG. The CFG is built through an iterative algorithm, where basic blocks are identified with the 'buildCFG' function, and the edges are built with 'buildEdges'. In addition, C++ std library's unordered set for additional data flow analyses was utilized to identify GEN and KILL sets for data flow analysis. The C++ std library queue data structure was used in the mark and sweep algorithm to traverse the graph to find 'live' code.

Software Challenges

When creating the optimizer, we ran into many issues. These issues popped up in version control, the source code, bash scripting, and even the make and compilation process. Our issues with version control stemmed from the fact that our team had relatively limited experience with working on a collaborative project in C++. As such, all of us made the crucial mistake of committing and pushing our build/ directory to the remote, which caused issues when each one of us pulled the cache files from the remote, causing the build process to fail. We resolved this by following good software engineering practices of pushing build files to GitHub.

We also had challenges with the dead code elimination, one of which was same block overwriting. We initially did not consider same block overwriting, and once we considered it, we realized that there was a lot of dead code that could be eliminated as a result of that.

For the bash scripts that we wrote, we initially wrote it in a way that hardcoded everything. This caused replicability issues, and so to fix it we wrote it such that the scripts were far more portable. This was an issue in a similar vein to the make and compilation process, as we did not make it portable and replicable initially. The same process to solve the bash scripting issue was applied here.

Unresolved Bugs/Deficiencies

At the moment, we are satisfied with our current project, and there do not seem to be any bugs/deficiencies that would otherwise be noticeable from out current suite of test cases.

Build/Usage Instructions

Before building, ensure that you are currently in the top level directory, otherwise you will encounter path/file not found issues. To build the optimizer, please run the following command:

```
./build.sh
```

This is under the assumption that a Unix-like machine will be used, if on Windows use WSL. After running the script, a build/ directory should be generated in your current working directory.

To run the optimizer, run the following command:

```
./run.sh <path to ir file> <output ir name>
```

Remember to replace

```
<path to ir file>
```

with the IR that you wish to run the optimizer on and

```
<output ir name>
```

with the name of the output IR file.

Results

We passed all of the public test cases. We took the optimized IR from our middle end compiler and interpreted it by utilizing the given Java interpreter script. The output from the interpretation was benchmarked against the CSV files for expected outputs, and the number of dynamic instructions between the two were equal for the given test cases.