Machine Learning Guided Ordering of Compiler Optimization Passes

Tarindu Jayatilaka
tarindujayatilaka@gmail.com

University of Moratuwa
Sri Lanka

April 10, 2021

1 Introduction

Developers generally use standard optimization pipelines like -O2 and -O3 to optimize their code. Manually crafted heuristics are used to determine which optimization passes to select and how to order the execution of those passes. However, this process is not tailored for a particular application, or kind of application, as it is designed to perform "reasonably well" for any input.

We want to improve the existing heuristics or replace the heuristics with machine learning-based models so that the LLVM compiler can provide a superior order of the passes customized per application.

2 Motivation

If the dependencies between the existing Passes and Code Structure can be analyzed using LLVM's internal statistics, the findings can be used to improve the existing heuristics or come up with new machine learning models to optimize the LLVM compiler. This will allow the compiler to order the transform passes in a superior manner tailored to individual applications (or groups of applications) and code structures. A wide range of applications and application developers will benefit from this with improved performance.

3 Project Objectives

- Study the LLVM's internal statistics and derive insights about the dependencies between existing passes.
- Create new standard pass-pipelines that will result in high-performance code depending on the program type. The desired pass-pipeline can be selected by the user in the form of a flag. For instance: -O3a, -O3b, ...
- Improve existing LLVM heuristics to select the best ordering of optimization passes depending on the code structure.
- Build a machine learning model that can identify patterns in the code structure, and select the best ordering of optimization passes based on the recognized pattern.

4 Current Progress

We track application code features and how they transform during the execution of the optimization pipeline in order to categorize applications with regards to the impact passes have on them. Once the dependencies between the existing passes and code features are identified for the different "kinds" of programs, the findings can be used to improve the existing heuristics or design new heuristics and models to optimize the LLVM pass pipeline. This will allow the compiler to order the transform passes in a superior manner, tailored to different code structures and features.

We look at the following code features on a per-function basis.

- total number of basic blocks in the function
- total number of basic blocks reached from conditional instructions in the function
- total number of uses of the function

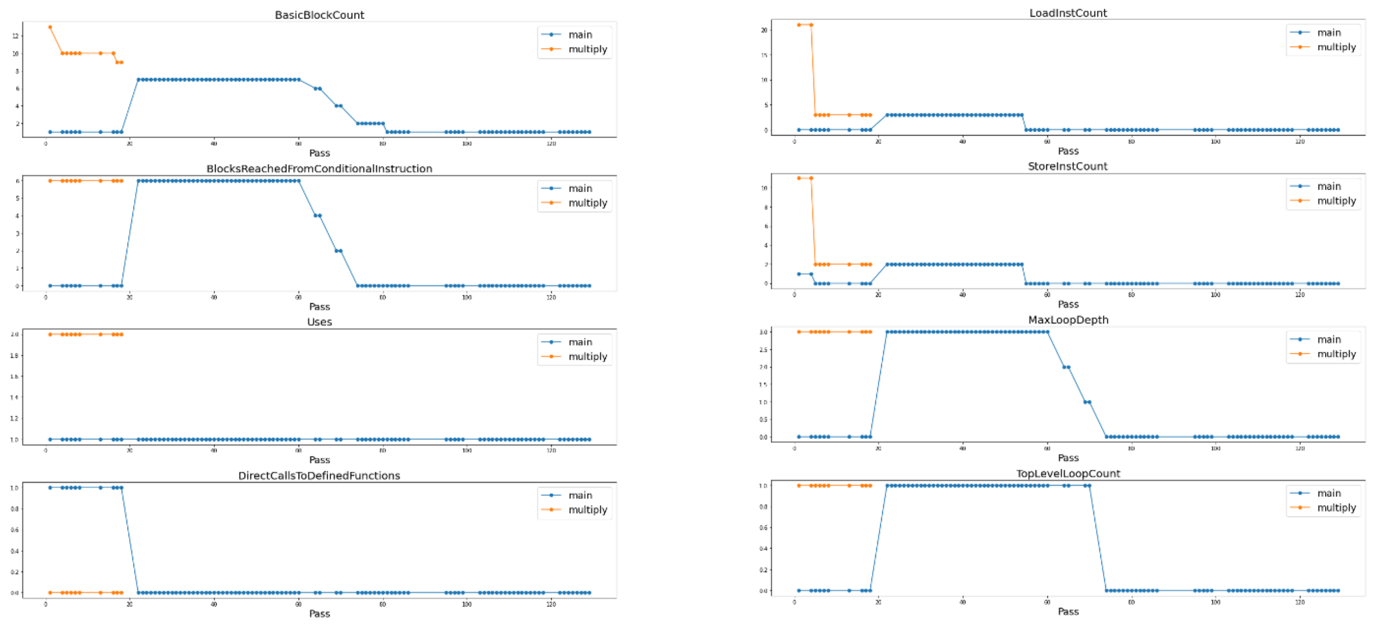


Figure 1: Function code features after each pass in the -O3 optimization pipeline is applied to a simple matrix multiplication program. The horizontal axis is the pass pipeline, while the vertical axis is the code feature value. Each line represents a function.

```
#define N 2

void multiply(int mat1[][N], int mat2[][N], int res[][N]);

int main() {
    int mat1[N][N], mat2[N][N], res[N][N];
    int i, j;
    multiply(mat1, mat2, res);
    return 0;
}

void multiply(int mat1[][N], int mat2[][N], int res[][N]) {
    int i, j, k;
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            res[i][j] = 0;
            for (k = 0; k < N; k++)
              res[i][j] += mat1[i][k]*mat2[k][j];
        }
    }
}
```

Figure 2: Source code for the matrix multiplication program

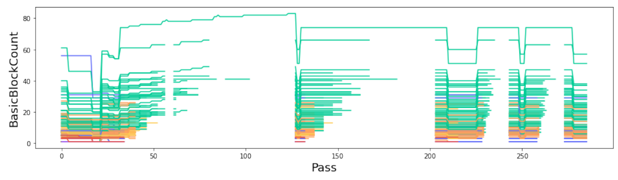
- total number of direct calls to other defined functions from the function
- total number of load instructions in the function
- total number of store instructions in the function
- maximum loop depth of the function
- total number of top-level loops in the function

4.1 Global Decisions

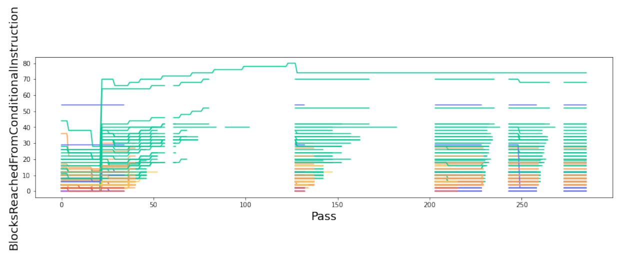
We record code features after every pass in the pass pipeline to determine how the pass affects them. Figure 1 illustrates how the code features change for the simple matrix multiplication program shown in Figure 2 with just two functions. You can see how inlining (and dead function elimination) affects the block count in the first plot and how canonicalizations later eliminate a loop in the program and replace it with a closed-form expression.

We determine different function "kinds" using the initial code features and the effect passes have on the function. We cluster the functions using the K-Means algorithm. Figure 3 shows the clusters for one module from the LLVM Test Suite. Each color in the graph represents a cluster.

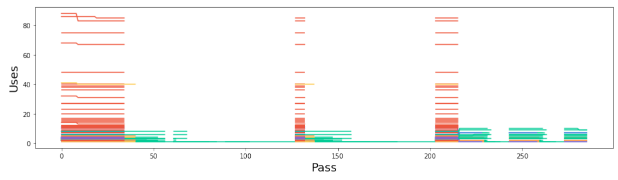
Each cluster requires different passes on their pipeline. If we can categorize functions into different kinds with high enough accuracy, we can predict if a particular pass may change (optimize) the code or if it can be skipped to reduce the compilation time or determine what pass-order to apply for maximizing the resulting optimization.



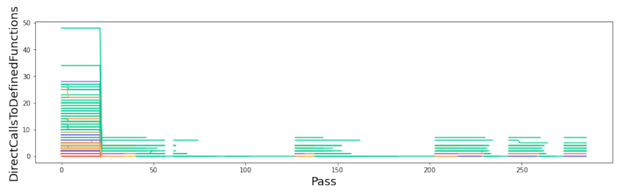
(a) total number of basic blocks in the function



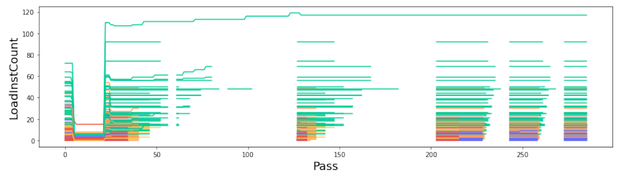
(b) total number of basic blocks reached from conditional instructions in the function



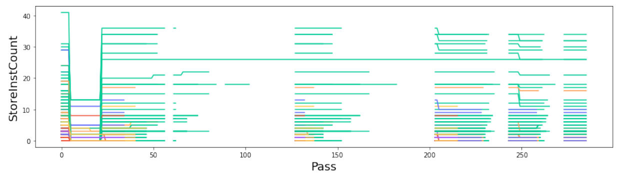
(c) total number of uses of the function



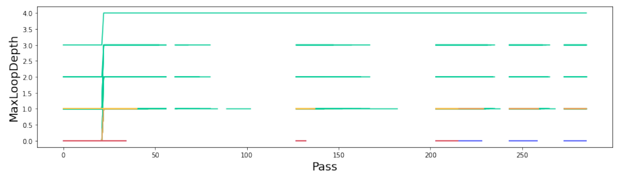
(d) total number of direct calls to other defined functions from the function



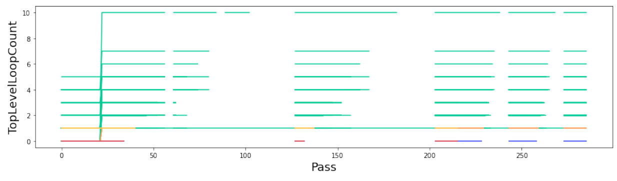
(e) total number of load instructions in the function



(f) total number of store instructions in the function



(g) maximum loop depth of the function



(h) total number of top-level loops in the function

Figure 3: Function code features after each pass in the -O3 optimization pipeline is applied to /MultiSource/Applications/SPASS/clause.c module from the LLVM Test Suite. The horizontal axis is the pass pipeline, while the vertical axis is the code feature value. Each line represents a function. Each color represents a cluster.

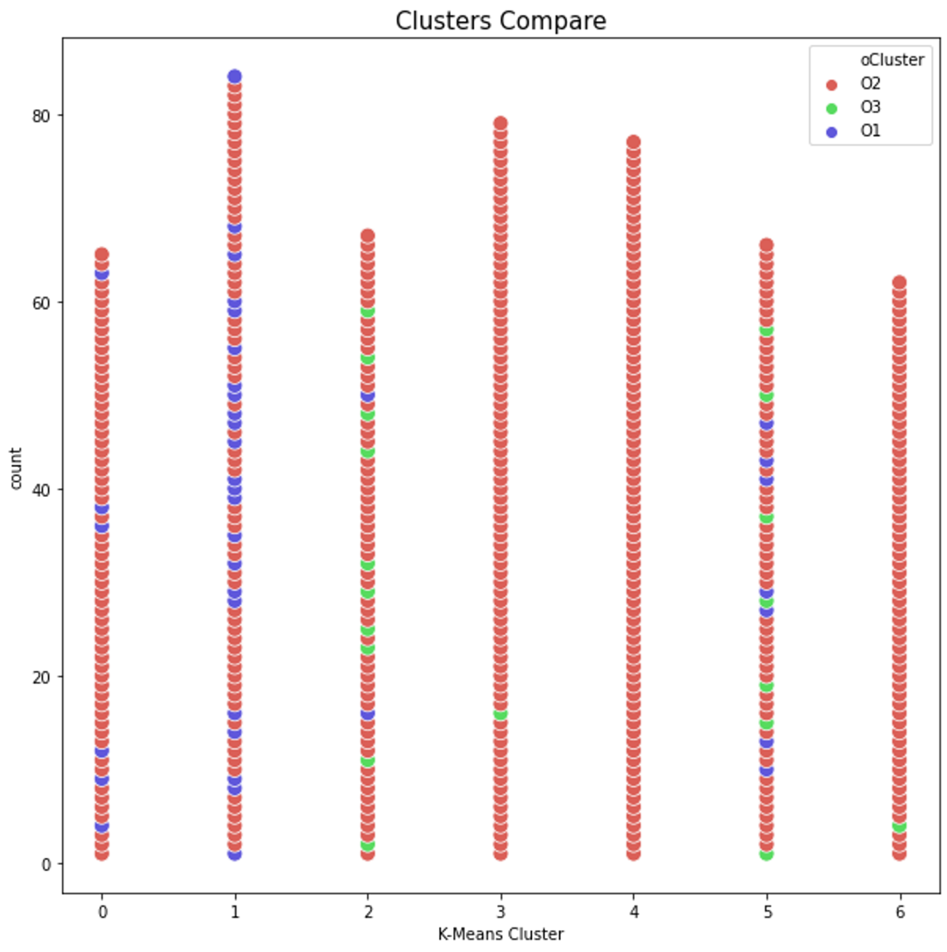


Figure 4: Optimization level required for each cluster. The x- axis represents the clusters obtained through K-Means clustering. Green represents functions that require O3, while red and blue represent functions that require O2 and O1 respectively.

We built an initial predictive model, and got an 85% accuracy to predict the cluster after just 30 passes, of the 300 passes in the pipeline.

4.1.1 Usage

To use these clusters practically, we compare the assembly files of each function after applying standard LLVM optimizations: O1, O2, and O3. If O1 produces same assembly file as O2 and O3, then applying O1 is enough for that function. If O2 produces same assembly file as O3, then applying O2 is enough for that function. Otherwise you need to apply O3. Figure 4 represents different optimization levels required for each function in /MultiSource/Applications/SPASS/clause.c module from the LLVM Test Suite. We can observe that the clusters 3,4, and 6 does fairly well with just O2, while the other clusters require some fine-tuning.

4.2 Local Decisions

LLVM has a lot of optimization passes, but current optimization passes do not change IR most of the time, as seen in the Figure 5. We are investing time in optimization passes which do not succeed, so we want to improve this situation by skipping such passes.

We create an oracle that can predict the effect of a pass on a function beforehand, we can skip the passes and save compile time. We use a machine learning model to build this oracle. Code features and most recent pass results are used as inputs to the model. The same static counters above are used as code features. The model then predicts which passes to apply from the next set of passes. The prediction model is depicted in Figure 6.

We embed this model into specific points in the optimization pipeline, by hacking into the buildFunctionSimplificationPipeline pass. First we run a set of passes and reach a checkpoint. At this point, previous pass results and code feature are passed to the oracle. We skip passes based on the oracle predictions. This is repeated this until the end of the pass pipeline. Figure 7 shows how the predictor is embedded into the pipeline.

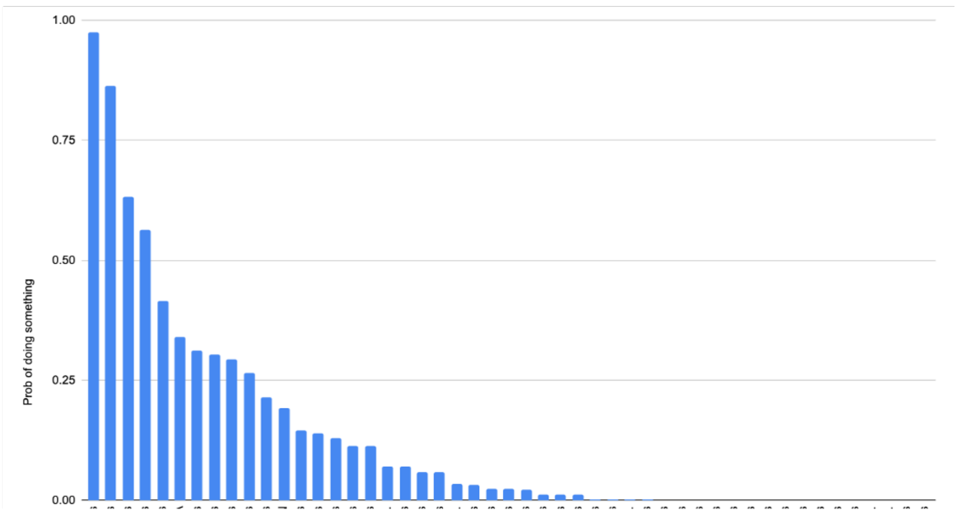


Figure 5: Probability an optimization pass is likely to change IR. The X-axis indicates kinds of passes and the Y-axis indicates the likeliness to change it.

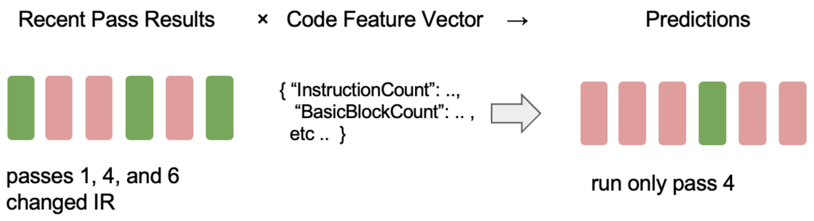


Figure 6: The prediction model takes the 'n' most recent pass results and code features after the most recent pass. Model returns the predictions for the next set of passes. This is a binary classification which determines whether to apply or skip a given pass.

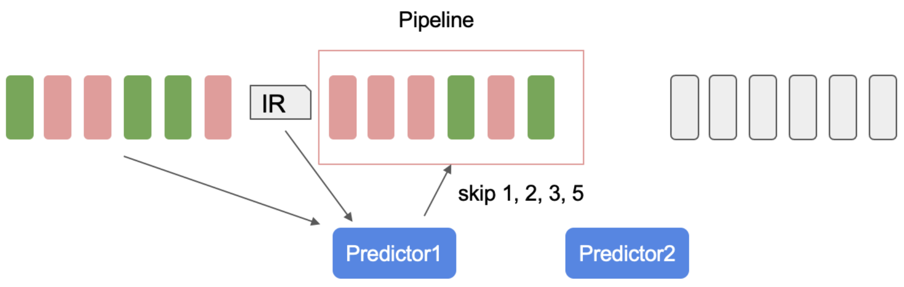
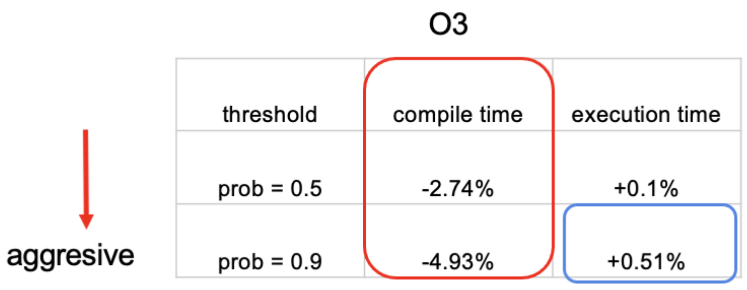


Figure 7: The predictor is embedded into the optimization pipeline at 4 points in buildFunctionSimplificationPipeline pass.


O3

threshold	compile time	execution time
prob = 0.5	-2.74%	+0.1%
prob = 0.9	-4.93%	+0.51%

Figure 8: Relative changes to the baseline in compile time and execution time after the predictor is embedded to the O3 optimization pipeline.

There is no perfect predictor as,

- False-negative (predictor says "no change" but actually "will change") → execution time regression
- False-positive (predictor says "will change" but actually "no change") → compile time regression

We have used relatively simpler models since complex machine learning models take longer to make a prediction. We train the model using LLVM's test-suite/MultiSource + SingleSource without CTMark, and run tests on CTMark. Figure 8 shows the current result, confirming that we can indeed reduce the compile time.

We have a hyper-parameter in our model which defines the threshold over which the predictor decides to apply a pass. The predictor only applies the pass, if the pass is likely to change the code with a probability above the threshold. This threshold defines aggressiveness of the algorithm.

As can be seen in Figure 8, the execution time slightly increases which might be due to too many passes being skipped.

5 Proposed Methodology

- Building a deployable model, that can decide between O1, O2, and O3 given a function and apply that, based on our current result.
- Introduce more code features and identify the most relevant features.
- Extend this to a larger dataset like SPEC. However, there is a problem in encoding a larger dataset. The hyper pass pipeline gets longer, thus takes more memory and computation time.
- Go beyond tabular code features by encoding the CFG or Calling Graph in a graph data structure and use graph neural networks for predictions.
- Extending the local predictions for more passes beyond function passes.
- Come up with a few different pipelines for beyond the current LLVM optimization pipelines. One approach to come up with these new pipelines would be to use RL to search through all combination pass pipelines, like Huang et al. [1] have suggested. Search space could be reduced by identifying dependencies between passes and grouping them together.

References

[1] Qijing Huang, Ameer Haj-Ali, William Moses, John Xiang, Ion Stoica, Krste Asanovic, and John Wawrzynek. Autophase: Juggling hls phase orderings in random forests with deep reinforcement learning, 2020.