Towards Compile-Time-Reducing Compiler Optimization Selection via Machine Learning

Tarindu Jayatilaka
tarindu.16@cse.mrt.ac.lk
University of Moratuwa
Sri Lanka

Hideto Ueno
hide05058945@g.ecc.u-tokyo.ac.jp
University of Tokyo
Japan

Johannes Doerfert
jdoerfert@anl.gov
Argonne National Laboratory
USA

Giorgis Georgakoudis
georgakoudis1@llnl.gov
Lawrence Livermore National Laboratory
USA

EunJung Park
ejpark@lanl.gov
Los Alamos National Laboratory
USA

ABSTRACT

Compilers come with a multitude of optimizations to choose from, and the chosen optimizations significantly affect the performance of the code being optimized. Selecting the optimal set of optimizations to apply and determining the order to run them is non-trivial. All of these optimizations closely interact with each other. An optimization's ability to improve the code heavily depends on how the previous optimizations modified it. The current approach to solve this is to use a one-size-fits-all optimization sequence, that is designed to perform reasonably well for any given source code. In other words, they are not designed to optimize depending on the nature of the code which usually results in non-optimal performance. In this paper, we present preliminary work tackling the problem from the perspective of compile-time by adapting the optimization sequence to cater to different program types. We start by analyzing how the source code interacts with the passes, as well as how the passes interact with each other. We use this information and propose two potential methods driven by machine learning to run customized optimization sequences on the source code. First, we look at how we can use a neural network to predict and skip passes that do not optimize the code to improve compilation time. Second, we look at how we can use clustering and predictive models to select custom pass pipelines. We believe that our approach has the potential to replace the current one-size-fits-all approach, with better optimization sequences that are tailored to perform better depending on the code. At the same time, it will allow testing the potential pipelines thoroughly, a practical requirement to gain confidence in the correctness of the compiler.

CCS CONCEPTS

• Software and its engineering → Compilers; **• Computing methodologies → Machine learning**.

Woodstock '18, June 03–05, 2018, Woodstock, NY
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06...$15.00
https://doi.org/10.1145/1122445.1122456

KEYWORDS

compiler optimization, phase-ordering, machine learning

ACM Reference Format:
Tarindu Jayatilaka, Hideto Ueno, Johannes Doerfert, Giorgis Georgakoudis, and EunJung Park. 2021. Towards Compile-Time-Reducing Compiler Optimization Selection via Machine Learning. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Compilers use transformation passes to produce an optimized version of the application that is expected perform better than the original source code. Since manually selecting the best set of optimizations to run on a given application is tedious and impossible to thoroughly test, developers generally use standard optimization sequences or pass pipelines like O1, O2, or O3 to optimize their code. Production compilers, like LLVM [13], still rely on manually curated pass pipelines to determine which optimization passes are run and in which order. While, this process is not tailored for a particular application, or kind of application, as it is designed to perform reasonably well for any input source code. Similarly important, it allows the compiler developers to test their pipelines and detect unforeseen interactions between passes during development.

Compilers ship with a plethora of optimization passes that can be incorporated into the pass pipeline. New passes are added to the optimization pipeline in order to achieve more aggressive optimization. Traditionally, developers simply look at the compile-time and the performance of the final optimized code to guide the pass pipeline design. In our work, we extend the optimization pipeline design to other factors which are usually overlooked; how often the passes actually optimize (or modify) code, how passes interact with one another, and their inter-dependencies.

We start by investigating the likelihood of a pass to modify the source code, and how passes interact with one another within the LLVM compiler infrastructure. Furthermore, we track and analyze how code features of the application change through the length of the pass pipeline. Once the dependencies between the existing passes and code features are identified for programs with different mixtures of instructions, e.g., pointer heavy, call heavy, etc., we use the findings to build models that optimize the pass pipeline. These models allow the compiler to devise improved orderings of the transformation passes, tailored to different code structures and features.

We make the following main contributions in this paper.

- Study LLVM's internal statistics and derive insights about how passes affect (modify or optimize) different functions. We accomplish this by tracking how code features change after each pass.
- Derive insights about the dependencies between existing passes. Specifically, we analyze the likelihood of a pass to modify code, and the conditional probabilities of the subsequent pass to modify code.
- Build an artificial neural network that can predict and skip passes that do not modify/optimize code, effectively saving compile time.
- Build a machine learning model that can identify patterns in the code structure, and select the best optimization pass pipeline based on the recognized pattern.

2 METHODOLOGY

In the first half of our methodology, we analyze the impact of different passes on LLVM IR and its code features. In the second half, we propose methods to optimize the LLVM pass pipeline in two folds: local level optimizations and global level optimizations.

2.1 Code Feature Analysis

We record static code features, such as counters of each instruction opcode, basic block counts (i.e. small/medium/large size basic blocks, single successor/predecessor basic blocks, etc.), floating-point instruction counts, after every pass in the pass pipeline to observe how the pass affects the source code. These features are tracked separately for each individual function in the program. Figure 2 illustrates how the code features change for the simple matrix multiplication program shown in Figure 1 with just two functions. Only the following code features are depicted in Figure 2:

- total number of basic blocks in the function
- total number of switch instructions in the function
- total number of direct calls to function definitions
- total number of load instructions in the function
- total number of store instructions in the function
- maximum loop depth of the function

We note that not all the passes in an optimization level are applied to all the functions. For instance, *Loop Deletion* pass and *Loop Full Unroll* pass are skipped for the main function, while they are applied to the multiply function. Therefore, to observe how function features change throughout the pass pipeline we need to align all the passes that were applied to each function. We use a dynamic programming based algorithm inspired by the Needleman–Wunsch algorithm [20] to align the pass sequences from each function.

In Figure 2 we can observe how inlining affects the main function across all the plots. Further, canonicalization passes later eliminate one of the loops and replace it with a closed-form expression. Another important observation is that most of the passes do not have an impact on the code feature. We go into more details about this in the next section.

```
#define N 2

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

int main() {
    int mat1[N][N], mat2[N][N], res[N][N];
    multiply(mat1, mat2, res);
}
```

Figure 1: Source code for the matrix multiplication program

2.2 Pass Analysis

We investigate the probability of a pass actually modifying, or optimizing, code. We apply the default O3 optimization to the LLVM test suite [16] and record whether each pass in the pass pipeline reported to have performed modifications. Note that we do not analyze the IR but trust the passes to report changes accurately. Figure 3 shows that only 5 passes out of the ones we observe are likely to change the code over 25% of the time. All the other passes have an even lower probability of modification. Moreover, there are a significant number of passes that have little to no effect on the LLVM test suite. This indicates that most of the passes waste resources most of the time without doing any actual optimizations, indicating room for large improvements in compile time.

As a next step we determine inter-dependencies between passes by running pairs of passes in isolation. A dependency exists if the rate with witch the second pass performs a modification is impacted by the first pass. Figure 4 shows the strengths of dependencies between the passes we look at. Simplified, we can avoid pass combinations in the pipeline in which the second pass has a low probability of change.

2.3 Local Optimizations

In local optimizations, we try to improve the LLVM pass pipeline by predicting if a pass will optimize (or modify) code in order to skip the ones that will not. Precisely, we create a neural network based predictor that can predict the effect of a pass on a function beforehand, and skip the passes with no impact to save compile time. Static code features mentioned in Section 2.1 and the 'n' most recent pass results (whether they modified the IR) are used as inputs to the neural network. To minimize the overhead we perform the prediction on a set of 'n' passes at once. The prediction model is depicted in Figure 5, with $n = 6$. In other words, the neural network makes predictions for subsets of 6 passes at a time.

To embed this model into LLVM's pass pipeline, we use the following approach. We insert predictors at four specific points in the pass pipeline, a different predictor for each point. We start by running a subset of passes and reaching the first checkpoint. At this point, the 'n' most recent pass results and code feature are fed to the predictor as inputs. The model predicts which passes from the next 'n' passes will optimize code based on the inputs. We skip passes based on these predictions. This process is repeated

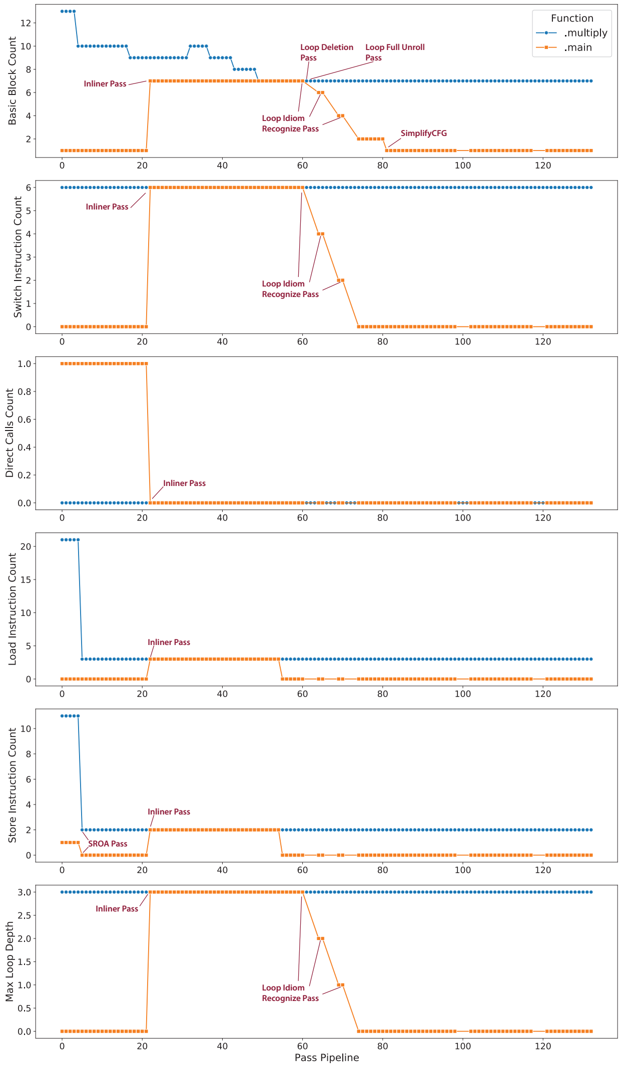


Figure 2: Function code features after each pass in the -O1 optimization pipeline is applied to the matrix multiplication in Figure 1. The horizontal axis represents the passes in the pipeline, the vertical axis the code feature count. Each line represents a function, specific passes are annotated to explain code feature changes.

until the end of the pass pipeline. Figure 6 provides an overview of how the predictor is embedded into the optimization pipeline. Note that our preliminary setup is only able to deal with function passes which substantially limits the effectiveness and is the reason we are limited to predetermined points in the pipeline.

The predictor is built using a fully connected artificial neural network with two hidden layers with ReLU and Sigmoid activation functions as depicted in Figure 7. Furthermore, we use Adam optimization with binary cross-entropy as the loss function. We use a relatively simpler neural network architecture as models with complex architectures take longer to make a prediction, effectively reducing any potential compile time gains. We use TensorFlow ahead of time (AOT) to compile the inference graph to an executable code that we call it from within the LLVM pass pipeline.

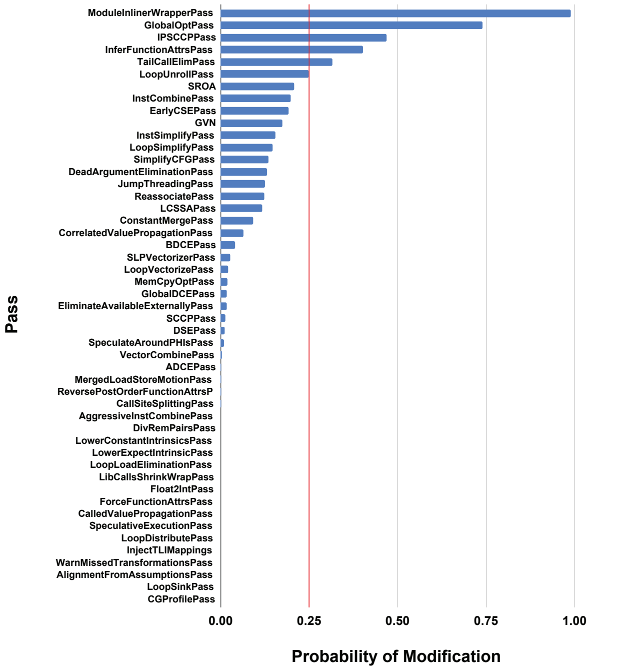


Figure 3: Probabilities of the passes in -O3 pipeline to modify the input code as reported by the passes themselves.

2.4 Global Optimizations

Our goal in global optimization is to categorize different input programs based on their code features and use the best suited pass pipeline. First, we cluster the functions using the code features

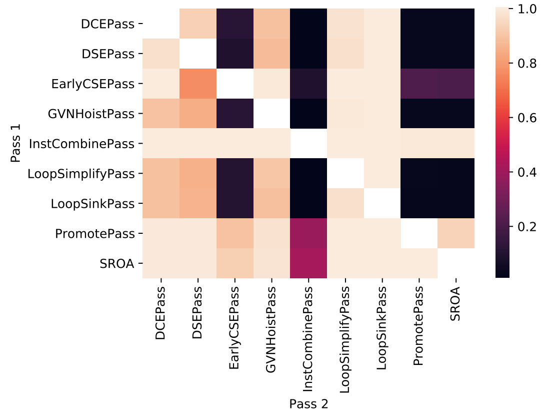


Figure 4: Heatmap showing the conditional probabilities of pass 2 modifying the IR, given pass 1 was executed immediately before that.

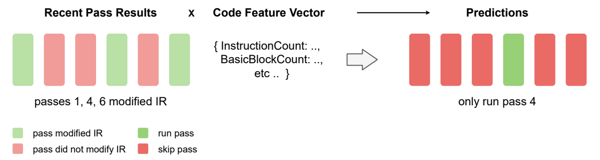


Figure 5: The prediction model takes the 'n' most recent pass results and the code features after as inputs. It returns the predictions for the next set of 'n' passes. The prediction for a single pass is a binary classification which determines whether to apply or skip the given pass.

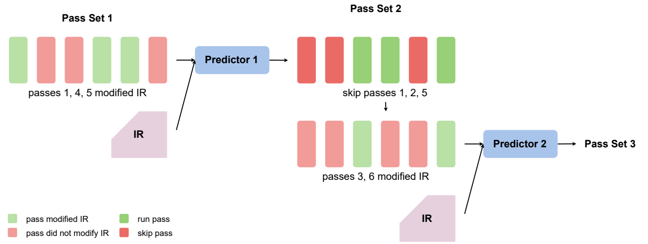


Figure 6: 4 predictors are embedded into the optimization pipeline at 4 points. This figure how the first predictor makes the predictions for $n = 6$ passes.

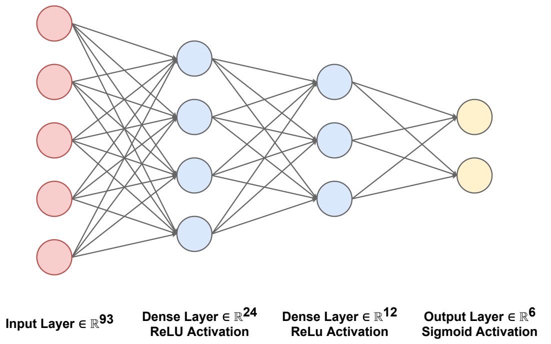

Input Layer ∈ $\mathbb{R}^{93}$
Dense Layer ∈ $\mathbb{R}^{24}$
ReLU Activation
Dense Layer ∈ $\mathbb{R}^{12}$
ReLu Activation
Output Layer ∈ $\mathbb{R}^{6}$
Sigmoid Activation

Figure 7: Neural network architecture.

introduced in Section 2.1. By using the same data set with intermediate code features, we are able to factor in the effect each pass has on the code to our clustering algorithm. We use the K-Means clustering algorithm [23] to identify 7 function clusters. Figure 8 shows the clusters obtained from a single module from the LLVM Test Suite, each color in the graph represents a cluster, the features are the same as in Figure 2.

We take the standard LLVM optimization levels–O1, O2, O3– as the potential pass sequences because they are well tested and provide different compile time investments. To assign the clusters to pipelines we compare the assembly files of each function after applying standard optimizations: O1, O2, and O3. If O1 produces the same output as O2 and O3, then applying O1 is enough for that function, similarly for O2. If the vast majority of functions within the cluster agree with the same optimization level, we allocate that pass sequence to that cluster. This allows us to directly reduce compile time.

We formulated our clusters by holistically looking at how different passes affected functions throughout the pass pipeline. But, now we need a way to know which cluster a function belongs to early on in the pass pipeline, in order to apply the custom pass sequence to it. We build a predictive model using Gradient Boosted Trees to accomplish this task.

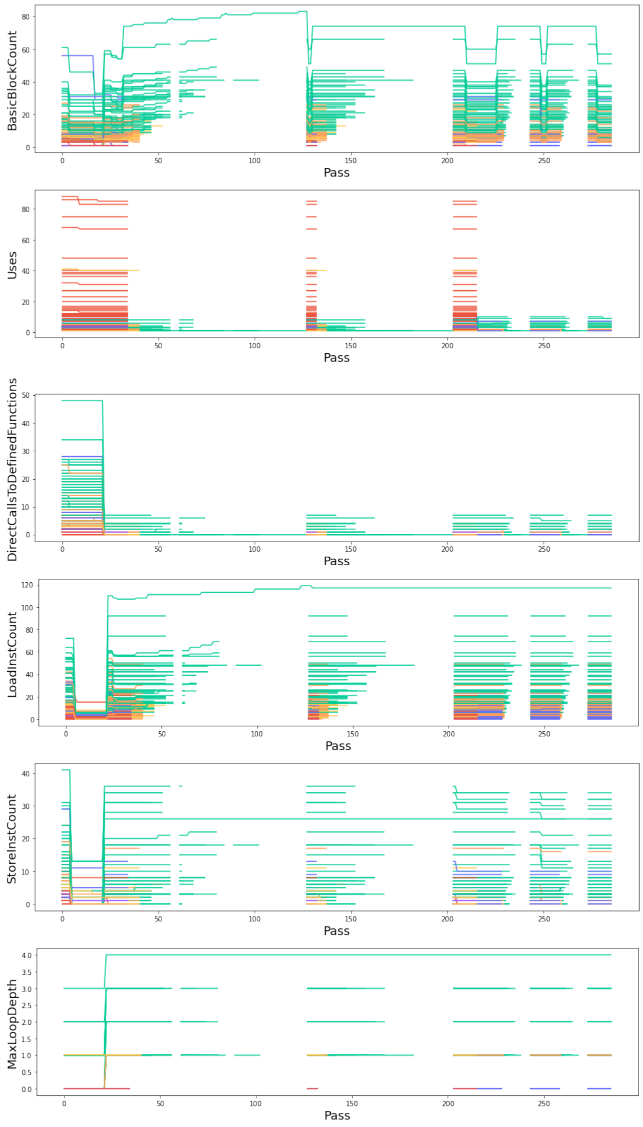


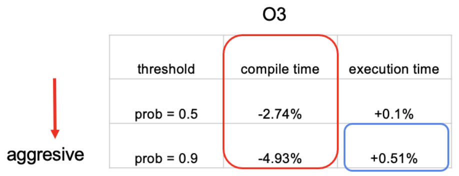
Figure 8: Function code features after each pass in the -O3 optimization pipeline is applied to MultiSource/Applications/SPASS/clause.c module from the LLVM Test Suite. The horizontal axis is the pass pipeline, while the vertical axis is the code feature value. Each line represents a function. Each color represents a cluster.

3 EXPERIMENTAL RESULTS

We evaluate our proposed methodology on Intel's Coffee Lake microarchitecture. We primarily worked with LLVM 12.0.0. For building the machine learning models we use TensorFlow 2.2.0.

3.1 Local Optimizations

We use the LLVM test suite's single and multi source benchmarks without CTMark as the train data set, and CTMark as the test data set. Figure 9 shows the current results, confirming that we can indeed reduce the compile time significantly while having only minor impact on the execution time.



O3

aggresive

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

Figure 9: Relative changes to the baseline in compile time and execution time after the pass skipping predictor is embedded into the O3 optimization pipeline for the CTMark benchmarks.

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

The predictions made by the predictor affect the compile and execution time of the program in the following ways.

- *true-positive*: predictor says pass will modify code and the pass will truly modify code → no change
- *false-positive*: predictor says pass will modify code but the pass will not truly modify code → missed compile time reduction
- *true-negative*: predictor says pass will not modify code and the pass will not truly modify code → compile time reduces
- *false-negative*: predictor says pass will not modify code but the pass will truly modify code → execution time changes

It should be noted that when we increase the aggressiveness of the algorithm, the number of false negatives increases. However, lowering the aggressiveness increases the number of false positives. Due to this trade-off, the ideal predictor is mostly dependent on the users priorities.

3.2 Global Optimizations

In this experiment we assign an optimization level from O1, O2, and O3 to each function based on their cluster. Figure 10 shows the different clusters and required optimization levels required for each function in MultiSource/Applications/SPASS/clause.c module from the LLVM Test Suite. We can observe that the majority of the functions in clusters 3, 4, and 6 only require an optimization level of O2; implying that O3 is wasteful for these functions. Similarly, clusters 0 and 1 are well served with O2, leaving O3 to be applied to clusters 2 and 5 only. While the clustering scheme is not perfect yet, we think our results show promise that this approach can indeed be implemented practically within the compiler through either better cluster analysis and/or refined custom pass sequences beyond O1, O2, and O3.

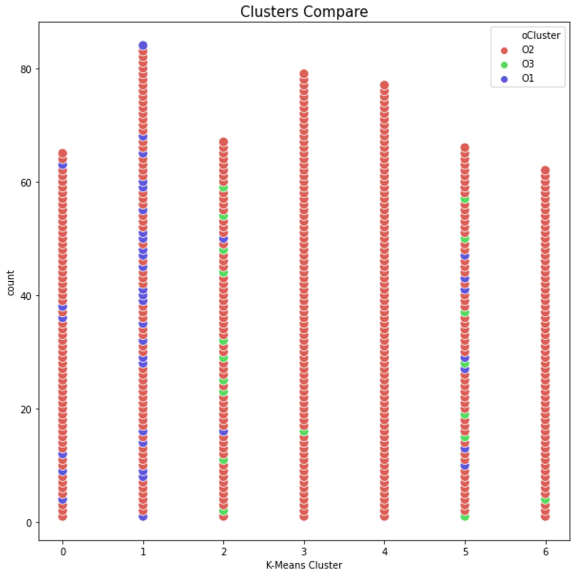


Figure 10: Clustering and optimization level required for each function in SPASS/clause.c. The x- axis represents the clusters obtained through K-Means clustering. Green represents functions that require O3, while red and blue represent functions that require O2 and O1, respectively.

The Gradient Boosted Tree based predictive model gives us an accuracy of 85% accuracy to predict the cluster of function. Our current approach requires the compiler to execute the first 30 passes (out of the 300+ passes long pass sequence resulting from the pass alignment) in the O3 pass pipeline and record the code features after the thirtieth pass. With these code features as input the model predicts the cluster for each function. We are working on the infrastructure to allow the compiler to then switch to O2 or O1 pass sequences to avoid executing futile passes. Other cutoff points and selection criteria need to be explored in the future.

4 RELATED WORK

Compiler developers face two major problems when writing optimizations due to the inter-dependencies between optimization phases: the optimization pass selection problem and the phase-ordering problem [5]. Earlier research tries to solve said problems using iterative compilation, while more recent work tends to use machine learning [15].

Applicability of iterative search techniques for compiler optimizations is explored in [6] and [10]. They show that iteratively searching over the transformation pass space can significantly improve the performance of code. Even though iterative compilation can improve performance, it was practically impossible to implement as it required hundreds of compilations and executions. There has been various attempt to address this limitation by; using compiler writer's knowledge encoded in heuristics to select a subset of optimizations [25], or algorithmically reducing the search space of optimization sequences [1, 21]. The OpenTuner framework [3] addresses these issues by using an ensemble of search techniques to select the optimization passes. Genetic algorithms have also been widely adopted to solve the search problem efficiently in heuristic tuning [24], as well as the phase ordering problem [11] .

However, due to the prohibitive overheads of repeated compilations, researchers began to look for better alternatives to iterative compilation. One such popular approach is to use machine learning techniques to learn compiler heuristics [19, 24]. Decision trees are used in [18] to generate target machine-specific heuristics for loop unrolling. Milepost GCC [8] used static analysis of code features to predict the set of passes to apply to a given program. MiCOMP [4] uses clustering and supervised learning techniques to tackle the phase ordering problem and outperforms LLVM's O3 optimization level. There is work that modifies that models the phase-ordering problem as a Markov process, and uses only the current state of the program to predict the next optimization [12]. Taking the idea of Markov models further, there has been few attempts to solve the phase-ordering problem using reinforcement learning models trained on static code features [9, 17]. However, all the said machine learning approaches rely on human-crafted code features. DeepTune [7] is the first attempt that uses a deep neural network to extract code features from the raw source code.

In contrast to most existing work we do not (yet) try to fully customize the optimization pipeline but instead select the parts of existing pipelines that are required. There are multiple benefits, e.g., a much smaller search space, though the main motivation is testability. Fully custom optimization pipelines are not testable in a reasonable way. Shipping a compiler without testing the optimization pipeline thoroughly is simply impractical as unforeseen interactions between passes that could lead to miscompilations are still way too common.

5 CONCLUSION AND FUTURE WORK

We presented analysis results of how code features change along the optimization pass pipeline. We also looked at how different passes affect code differently and interact with each other. We were able to identify passes that have strong inter-dependence. We also presented our preliminary work on how to utilize these findings within the compiler itself to reduce compile time, and in the future, execution time as well. Our current models are able to give significant improvements over the compile time. We believe that our current approach for global optimizations can be readily adapted to the compiler with the addition of few custom pass sequences.

We have a few shortcomings in our current work, which we plan to address in future work. We currently only look at function-specific static code features categorize programs. We need to investigate more and pick out more relevant code features. Furthermore, we could look beyond static code analysis, and use a machine learning based [2, 14] or a graph based [22] approach to generate the code features from the program. However, this has to be cautiously done since complex feature generation techniques can significantly impact the compile time. Additionally, we plan to extend this to larger benchmark sets to build our models, beyond the current LLVM test suite.

We also plan to come up with a few different pass pipelines that can cater to each function cluster we identify. One approach to come up with these new pipelines would be to use reinforcement learning to search through all combination pass pipelines, similar to the approach [9] has followed. The search space could be reduced by using the dependencies between passes to group them together.

ACKNOWLEDGMENTS

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering, and early testbed platforms, in support of the nation's exascale computing imperative.

REFERENCES

[1] L. Almagor, Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven W. Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. 2004. Finding effective compilation sequences. *ACM SIGPLAN Notices* 39, 7 (2004), 231–239. https://doi.org/10.1145/998300.997196

[2] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages* (2019). https://doi.org/10.1145/3290353 arXiv:1803.09473

[3] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una May O'Reilly, and Saman Amarasinghe. 2014. OpenTuner: An extensible framework for program autotuning. *Parallel Architectures and Compilation Techniques - Conference Proceedings, PACT* (2014), 303–315. https://doi.org/10.1145/2628071.2628092

[4] Amir H. Ashouri, Andrea Bignoli, Gianluca Palermo, Cristina Silvano, Sameer Kulkarni, and John Cavazos. 2017. MiCOMP: Mitigating the compiler phase-ordering problem using optimization sub-sequences and machine learning. *ACM Transactions on Architecture and Code Optimization* 14, 3 (2017). https://doi.org/10.1145/3124452

[5] Amir H. Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. 2018. A Survey on Compiler Autotuning Using Machine Learning. *ACM Comput. Surv.* 51, 5, Article 96 (Sept. 2018), 42 pages. https://doi.org/10.1145/3197978

[6] François Bodin, Toru Kisuki, Peter Knijnenburg, Mike O' Boyle, and Erven Rohou. 1998. Iterative compilation in a non-linear optimisation space. In *Workshop on Profile and Feedback-Directed Compilation*. Paris, France. https://hal.inria.fr/inria-00475919

[7] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. 2017. End-to-End Deep Learning of Optimization Heuristics. *Parallel Architectures and Compilation Techniques - Conference Proceedings, PACT* 2017-September (2017), 219–232. https://doi.org/10.1109/PACT.2017.24

[8] Grigori Fursin, Yuriy Kashnikov, Abdul Wahid Memon, Zbigniew Chamski, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Bilha Mendelson, Ayal Zaks, Eric Courtois, Francois Bodin, Phil Barnard, Elton Ashton, Edwin Bonilla, John Thomson, Christopher K.I. Williams, and Michael O'Boyle. 2011. Milepost GCC: Machine learning enabled self-tuning compiler. *International Journal of Parallel Programming* 39, 3 (2011), 296–327. https://doi.org/10.1007/s10766-010-0161-2

[9] Qijing Huang, Ameer Haj-Ali, William Moses, John Xiang, Krste Asanovic, John Wawrzynek, and Ion Stoica. 2020. AutoPhase: Juggling HLS Phase Orderings in Random Forests with Deep Reinforcement Learning. In *Proceedings of Machine Learning and Systems*. arXiv:2003.00671 https://proceedings.mlsys.org/paper/2020/file/4e732ced3463d06de0ca9a15b6153677-Paper.pdf

[10] T. Kisuki, P. M.W. Knijnenburg, and M. F.P. O'boyle. 2000. Combined selection of tile sizes and unroll factors using iterative compilation. *Parallel Architectures and Compilation Techniques - Conference Proceedings, PACT* 24, 1 (2000), 237–246. https://doi.org/10.1109/PACT.2000.888348

[11] Prasad A. Kulkarni, Stephen R. Hines, David B. Whalley, Jason D. Hiser, Jack W. Davidson, and Douglas L. Jones. 2005. Fast and Efficient Searches for Effective Optimization-Phase Sequences. *ACM Transactions on Architecture and Code Optimization* 2, 2 (2005), 165–198. https://doi.org/10.1145/1071604.1071607

[12] Sameer Kulkarni and John Cavazos. 2012. Mitigating the compiler optimization phase-ordering problem using machine learning. *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA* (2012), 147–162. https://doi.org/10.1145/2384616.2384628

[13] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis &amp; transformation. *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* c (2004). https://doi.org/10.5555/977395.977673

[14] Hugh Leather, Edwin Bonilla, and Michael O'Boyle. 2009. Automatic feature generation for machine learning based optimizing compilation. *Proceedings of the 2009 CGO - 7th International Symposium on Code Generation and Optimization* (2009), 81–91. https://doi.org/10.1109/CGO.2009.21

[15] Hugh Leather and Chris Cummins. 2020. Machine Learning in Compilers: Past, Present and Future. *Forum on Specification and Design Languages* 2020-September (2020). https://doi.org/10.1109/FDL50818.2020.9232934

[16] LLVM. 2021. LLVM Test Suite. https://llvm.org/docs/TestSuiteGuide.html.

[17] Rahim Mammadli, Ali Jannesari, and Felix Wolf. 2020. Static Neural Compiler Optimization via Deep Reinforcement Learning. *Proceedings of LLVM-HPC2020 and HiPar 2020 - 6th Workshop on the LLVM Compiler Infrastructure in HPC and Workshop on Hierarchical Parallelism for Exascale Computing, Held in conjunction with SC 2020: The International Conference for High Performance Computing, Networking, Storage and Analysis* (2020), 1–11. https://doi.org/10.1109/LLVMHPCHiPar51896.2020.00006 arXiv:2008.08951

[18] Antoine Monsifrot, Franccois Bodin, and Reneé Quiniou. 2002. A machine learning approach to automatic production of compiler heuristics. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 2443 (2002), 41–50. https://doi.org/10.1007/3-540-46148-5_5

[19] Eliot Moss, Paul Utgoff, John Cavazos Doina Precup, Darko Stefanovic, Carla Brodley, and David Scheeff. 1998. Learning to schedule straight-line code. *Advances in Neural Information Processing Systems* (1998), 929–935.

[20] Saul B. Needleman and Christian D. Wunsch. 1970. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology* 48, 3 (1970), 443–453. https://doi.org/10.1016/0022-2836(70)90057-4

[21] Zhelong Pan and Rudolf Eigenmann. 2006. Fast and effective orchestration of compiler optimizations for automatic performance tuning. *Proceedings of the CGO 2006 - The 4th International Symposium on Code Generation and Optimization* ii (2006), 12–23. https://doi.org/10.1109/CGO.2006.38

[22] Eunjung Park, John Cavazos, and Marco A. Alvarez. 2012. Using graph-based program characterization for predictive modeling. *Proceedings - International Symposium on Code Generation and Optimization, CGO 2012* (2012), 196–206. https://doi.org/10.1145/2259016.2259042

[23] Douglas Steinley. 2006. K-means clustering: A half-century synthesis. *Brit. J. Math. Statist. Psych.* 59, 1 (2006), 1–34. https://doi.org/10.1348/000711005X48266

[24] Mark Stephenson, Saman Amarasinghe, Martin Martin, and Una May O'Reilly. 2003. Meta optimization: Improving compiler heuristics with machine learning. *ACM SIGPLAN Notices* 38, 5 (2003), 77–90. https://doi.org/10.1145/780822.781141

[25] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. I. August. 2003. Compiler optimization-space exploration. *International Symposium on Code Generation and Optimization, CGO 2003* (2003), 204–215. https://doi.org/10.1109/CGO.2003.1191546