A Holistic View of Code Layout Transformations in the LLVM Opt Pipeline

Problem: Transformation passes that change code layout are an essential component of compiler optimization pipelines. A blind spot in the LLVM optimization pipeline (further referred to as 'opt pipeline') is that contributions to these pipelines are overwhelmingly local. LLVM's code layout passes are modular by design, but this obscures interactions between passes preceding and following them in the pipeline. While there are conceptual advantages to this (e.g. developer freedom to reorder passes), in practice, passes often depend on the code transformations from prior passes to perform effective optimizations. A sensitivity analysis done by Torres et al. indicates that when considered alone passes minimally impact execution time, but alongside other passes the impact of a single pass jumps immensely [1], demonstrating the importance of a holistic perspective. Although the importance of pass dependencies is understood, there is little work done in holistically mapping code layout transformations in the pipeline. To address this gap, we will review and map out the opt pipeline's code layout passes alongside their theoretical background.

Related Work: The most influential code transformation paper is by Pettis and Hansen on profile guided code positioning [2]. Petti and Hansen introduce code rearranging heuristics that are foundational to modern layout optimization strategies. Recent work builds upon this foundation, for example, Ottoni and Maher [3] extend the original heuristics by introducing a call-chain clustering directed graph algorithm that is now implemented in BOLT [4], a post-link layout optimizer that works with LLVM. These works reveal that layout transformations can happen on a broad time spectrum, from early IR to post-linking optimizations. Understanding this spectrum emphasizes the need to study LLVM's layout pipeline as a cohesive, interconnected system. However, most of the works dealing with the LLVM pass pipeline in its entirety are related to the phase-ordering problem, which attempts to find a better ordering for LLVM passes (typically using automated methods, such as in [5] and [6]). However, these treat the pipeline as a black box that can be optimized via machine learning , and do not address understanding and documenting code movement. Rather than treating the pipeline as a black box, our work aims to explore how passes interact to produce a cumulative code structure.

Method: We will investigate the LLVM 20 source for each code-layout pass, recording its actions, compiler heuristics, links to theoretical references and papers, and a short description of its role in the pipeline with respect to other passes. This investigation will include profile-guided optimizations, and optimizations at the IR and MIR levels. We will classify this information in a taxonomy by transformation type and IR unit, hosted on a publicly-available GitHub Wiki with cross links to relevant passes and theoretical references. The collected data will be synthesized into a dependency graph/diagram that briefly summarizes the code layout transformations and where they occur. In parallel to this investigation, we will explore and document the theoretical foundations and modern research that is relevant to LLVM's code transformation optimizations. Finally, we will produce a report summarizing the most relevant findings of our investigation. Our project will be evaluated to be complete once we have covered all code layout passes that modify code ordering above an instruction level. A timeline of our strategy is as follows: from now to October 25th, we will complete a high-level search and collection of relevant passes and literature. From October 25th to November 25th, we will collect data on GitHub Wiki and write up our report. From November 25th to December 8th we will finalize our report and data.

Significance: We aim to produce a publicly available comprehensive overview of LLVM code transformation passes and their theoretical backgrounds to empower further research on pass development and ordering. This will contribute to a global understanding of the pipeline providing avenues for further improvements in code layout pass design. While previous works have investigated the performance impact on pass orderings, none (to our knowledge) systematically document cumulative layout transformations in the LLVM opt pipeline; our research aims to fill that gap.

Bibliography

[1] J. C. de la Torre, P. Ruiz, B. Dorronsoro, and P. L. Galindo, "Analyzing the Influence of LLVM Code Optimization Passes on Software Performance," in *Information Processing and Management of Uncertainty in Knowledge-Based Systems. Applications*, J. Medina, M. Ojeda-Aciego, J. L. Verdegay, I. Perfilieva, B. Bouchon-Meunier, and R. R. Yager, Eds., Cham: Springer International Publishing, 2018, pp. 272–283.

[2] K. Pettis and R. C. Hansen, "Profile guided code positioning," in *Programming Language Design and Implementation (ACM)*, 1990.

[3] G. Ottoni and B. Maher, "Optimizing function placement for large-scale data-center applications," in *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2017, pp. 233–244.

[4] M. Panchenko, R. Auler, B. Nell, and G. Ottoni, "Bolt: a practical binary optimizer for data centers and beyond," in *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2019, pp. 2–14.

[5] A. H. Ashouri, A. Bignoli, G. Palermo, C. Silvano, S. Kulkarni, and J. Cavazos, "MiCOMP: Mitigating the Compiler Phase-Ordering Problem Using Optimization Sub-Sequences and Machine Learning," *ACM Trans. Archit. Code Optim.*, vol. 14, no. 3, Sep. 2017, doi: 10.1145/3124452.

[6] A. Haj-Ali *et al.*, "Autophase: Juggling hls phase orderings in random forests with deep reinforcement learning," *Proceedings of machine learning and systems*, vol. 2, pp. 70–81, 2020.