ELE 548: Project Proposal

Calvin Higgins

November 1, 2025

1 Abstract

Attaining optimal program performance on modern heterogeneous hardware requires specializing compiler optimizations to individual architectures. To ease the engineering burden, researchers have proposed automatically learning optimal orderings of compiler optimization passes. In this work, we minimize (1) binary size and (2) mean execution time by ordering LLVM optimization passes with EfficientZero, a sample-efficient reinforcement learning method. We attain an X% reduction in binary size compared to `-Oz` and an X% speedup compared to `-O3` when optimizing the MiBench embedded systems benchmark suite.

2 Introduction

The goal of a **compiler** is simple: find an optimal translation of high-level code into machine code. Compilers begin with a relatively naive translation, and iteratively refine it with transformations known as **optimization passes**. Different optimization pass orders can yield binaries with drastically different sizes and runtime performance. The **phase ordering problem** involves determining the best order to apply a fixed set of optimization passes. Historically, expert compiler engineers designed new optimization pass orders for every architecture. However, as hardware rapidly evolves and becomes increasingly heterogeneous, maintaining optimization pass orders imposes a growing engineering burden. Automation can alleviate this burden and allow compiler experts to focus on more critical tasks [17].

By formulating the phase ordering problem as a **Markov Decision Process (MDP)**, standard reinforcement learning techniques apply:

1. **State space.** S is the set of all legal programs.
2. **Action space.** A is the set of all optimization passes.
3. **Transition function.** $T(s, a)$ is the result of applying pass a to program s.

4. **Reward function.** $R(s, a) = C(s) - C(T(s, a))$ where $C(s)$ is a cost function such as binary size, instruction count, or execution time.

Unfortunately, the phase ordering problem is challenging: the action space A contains hundreds of passes, leading to a combinatorial explosion of possible optimization sequences. Moreover, rewards are very sparse as effective sequences are rare. Even evaluating the transition and reward functions is slow as they require compiler invocations.

Accordingly, any learning-based approach to the phase-ordering problem ought to be sample-efficient. CompilerDream [3], a state-of-the-art phase-ordering policy, adapted DreamerV3 [6], a general purpose reinforcement learning method, to compiler optimization. However, although DreamerV3 dominates nearly all reinforcement learning methods across a diverse set of domains, it is notably overshadowed by EfficientZero [18] in low sample regimes [6]. Motivated by its sample efficiency, I will apply EfficientZero to the phase ordering problem.

3 Related Work

Autophase [7], the first reinforcement learning method for the phase ordering problem, obtained a 1.28× speedup compared to `-O3` on high-level circuit synthesis with proximal policy optimization (PPO). CORL [11] obtained a 1.32× speedup compared to `-O3` on the LLVM test suite with deep Q-learning. POSET-RL[8] also used deep Q-learning to optimize execution time and binary size, achieving a 1.019× x86 binary size improvement relative to `-Oz` on embedded systems benchmarks. More recently, DeCOS [1] bootstrapped standard reinforcement learning methods with large language models, and achieved speedups on SPEC 2017 benchmarks. CompilerDream [3] pretrained DreamerV3 for code optimization, and obtained state-of-the-art results on IR instruction count reduction.

4 Design

EfficientZero [18] is a reinforcement learning method that continues the AlphaGo [16], AlphaZero [15], MuZero [12], and MuZero Reanalyze [13] linage. Like MuZero and MuZero Reanalyze, EfficientZero contains three networks:

1. **Representation function.** Maps an observation (program) to a latent state (program representation).
2. **Dynamics function.** Maps a latent state (program representation) and an action (optimization pass) to a new latent state (program representation).
3. **Prediction function.** Maps a latent state (program representation) to a policy (probability distribution over optimization passes) and a reward (change in binary size).

By mapping observations into a latent space with the representation function, EfficientZero can use the dynamics and prediction functions to explore the state space without executing actions on the real state space. In the context of phase ordering, this minimizing compiler invocations. EfficientZero also includes several other training tricks on top of MuZero Reanalyze to accelerate convergence.

Initially, I will pre-train EfficientZero on AnghaBench [14], a suite of 1,041,333 compile-only C/C++ functions scraped from GitHub. AnghaBench is large-scale, diverse and human-generated, but many programs are too simple to afford significiant optimization opportunities [3]. Time permitting, I will pre-train on CodeContests [10] instead, as recommended by the authors of CompilerDream [3].

5 Evaluation

CompilerGym [2] is a set of environments for compiler optimization tasks such as ordering optimization passes to reduce binary size and instruction counts. Since there is limited support for optimizing execution time, I will minimize x86-64 binary size by ordering optimization passes from the LLVM [9] compiler infrastructure. In particular, I will evaluate my method on binary size improvement factor relative to `-Oz`, the human-designed aggressive binary size optimization sequence, on the MiBench embedded systems benchmark suite[5]. Time permitting, I will explore other datasets like cBench [4], stronger baselines like random search, other objectives like minimizing mean execution time and other platforms like ARM64.

References

[1] Tianming Cui et al. "DeCOS: Data-Efficient Reinforcement Learning for Compiler Optimization Selection Ignited by LLM". In: *International Conference on Supercomputing (ICS)*. 2025. DOI: 10.1145/3721145.3725765.

[2] Chris Cummins et al. "CompilerGym: robust, performant compiler optimization environments for AI research". In: *International Symposium on Code Generation and Optimization (CGO)*. 2022. DOI: 10.1109/CGO53902.2022.9741258.

[3] Chaoyi Deng et al. "CompilerDream: Learning a Compiler World Model for General Code Optimization". In: *Conference on Knowledge Discovery and Data Mining (KDD)*. 2025. DOI: 10.1145/3711896.3736887.

[4] Grigori Fursin. "Collective Tuning Initiative". In: *GCC Developers Summit*. 2009.

[5] M.R. Guthaus et al. "MiBench: A free, commercially representative embedded benchmark suite". In: *IEEE International Workshop on Workload Characterization (WWC)*. 2001. DOI: 10.1109/WWC.2001.990739.

[6] Danijar Hafner et al. “Mastering diverse control tasks through world models”. In: *Nature* (2025). DOI: 10.1038/s41586-025-08744-2.

[7] Ameer Haj-Ali et al. “AutoPhase: Juggling HLS Phase Orderings in Random Forests with Deep Reinforcement Learning”. In: *Machine Learning and Systems (MLSys)*. 2020.

[8] Shalini Jain et al. “POSET-RL: Phase ordering for Optimizing Size and Execution Time using Reinforcement Learning”. In: *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2022. DOI: 10.1109/ISPASS55109.2022.00012.

[9] C. Lattner and V. Adve. “LLVM: a compilation framework for lifelong program analysis and transformation”. In: *International Symposium on Code Generation and Optimization (CGO)*. 2004. DOI: 10.1109/CGO.2004.1281665.

[10] Yujia Li et al. “Competition-level code generation with AlphaCode”. In: *Science* (2022). DOI: 10.1126/science.abq1158.

[11] Rahim Mammadli, Ali Jannesari, and Felix Wolf. “Static Neural Compiler Optimization via Deep Reinforcement Learning”. In: *Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC) and Workshop on Hierarchical Parallelism for Exascale Computing (HiPar)*. 2020. DOI: 10.1109/LLVMHPCHiPar51896.2020.00006.

[12] Julian Schrittwieser et al. “Mastering Atari, Go, chess and shogi by planning with a learned model”. In: *Nature* (2020). DOI: 10.1038/s41586-020-03051-4.

[13] Julian Schrittwieser et al. “Online and offline reinforcement learning by planning with a learned model”. In: *International Conference on Neural Information Processing Systems (NeurIPS)*. 2021. ISBN: 9781713845393.

[14] Anderson Faustino da Silva et al. “AnghaBench: a suite with one million compilable C benchmarks for code-size reduction”. In: *International Symposium on Code Generation and Optimization (CGO)*. 2021. DOI: 10.1109/CGO51591.2021.9370322.

[15] David Silver et al. “A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play”. In: *Science* (2018). DOI: 10.1126/science.aar6404.

[16] David Silver et al. “Mastering the game of Go with deep neural networks and tree search”. In: *Nature* (2016). DOI: 10.1038/nature16961.

[17] Zheng Wang and Michael O’Boyle. “Machine Learning in Compiler Optimization”. In: *IEEE* (2018). DOI: 10.1109/JPROC.2018.2817118.

[18] Weirui Ye et al. “Mastering Atari Games with Limited Data”. In: *International Conference on Neural Information Processing Systems (NeurIPS)*. 2021.