# Ordering Optimization Passes with Reinforcement Learning for Instruction Count Reduction

Calvin Higgins
*Department of Computer Science and Statistics*
*University of Rhode Island*
Kingston, United States
calvin_higgins2@uri.edu

***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 IR instruction count by ordering LLVM optimization passes with deep Q-learning, a standard reinforcement learning method. We attain an 52% reduction in instruction count compared to `-O0` (11% regression from `-Oz`) across six representative benchmark suites.**

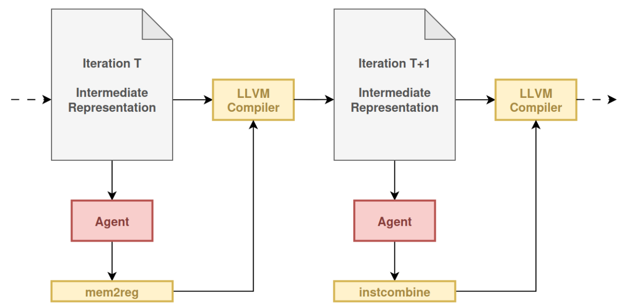


Fig. 1. Reinforcement learning for the phase ordering problem. An agent (red) inspects the intermediate representation of a program (gray), selects and applies an optimization pass (yellow), and repeats to obtain an optimized program. Adapted from [9].

## I. Introduction

The goal of a compiler is simple: find an optimal translation of high-level code into machine code. Many modern compilers rely on the LLVM compiler infrastructure which provides a unified intermediate representation (think architecture-independent assembly language) as well as associated optimization and code-generation tools [17]. These compilers typically begin with a relatively naive translation from high-level code into LLVM intermediate representation (IR), and iteratively refine it with transformations known as optimization passes. Different optimization pass orders can yield binary 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 have 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, allowing compiler experts to focus on more critical tasks [29]. Unfortunately, as LLVM exposes hundreds of distinct passes, and typical orders apply hundreds of passes, automatically finding good orders is extremely challenging. Deep reinforcement learning has successfully explored other difficult search spaces, such as video games, demonstrating promise for compiler optimization [13].

By formulating the phase-ordering problem as a Markov decision process, standard reinforcement learning techniques apply. A Markov decision process is a four tuple $(S, A, T, R)$ where $S$ is a set of states, $A$ is a set of actions, $T(s, a)$ is a transition function mapping from a state $s$ and an action $a$ to a new state, and $R(s, a)$ is the change in state cost upon applying action $a$ to state $s$. In the phase ordering problem, the set of states $S$ is an equivalence class of programs, the action space $A$ is the set of all optimization passes, the transition function $T(s, a)$ outputs the program $s$ after applying optimization pass $a$, and the 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, not only does the large action space $A$ lead to a combinatorial explosion of possible optimization sequences, but rewards are also sparse as effective sequences are rare, and evaluating the transition and reward functions is slow as they require compiler invocations.

Accordingly, any approach to the phase-ordering problem ought to be sample efficient. CompilerDream [9], a state-of-the-art phase-ordering policy, adapted DreamerV2 [12], a general purpose reinforcement learning method to compiler optimization. However, although the Dreamer methods dominate nearly all reinforcement learning methods across a diverse set of domains, they are notably overshadowed by EfficientZero [30] in low sample regimes [13]. Motivated by its sample efficiency, we aim to apply EfficientZero to the phase-ordering problem. In this work, we conduct preliminary experiments with deep Q-learning [20], hoping to gain experience and insights before implementing EfficientZero.

## II. Related Work

Over the past five years, program optimization via learning LLVM pass orders has been an active field of research. An extensive list of methods is provided in Table I. Sample efficiency has been of particular concern: recent approaches have tried better learning algorithms [9], coreset decompositions [18, 23], and integrating LLMs [5, 22]. Comparatively, little work has been done on program representation as AutoPhase features, handcrafted vectors of 56 features statically extracted from IR, remain popular despite broader trends towards end-to-end deep learning [9, 23, 22]. Only GEAN [18] has used a reasonably modern IR representation, ProGraML [8]. Evaluations of recent, powerful model-based RL methods like DreamerV3 [13] and EfficientZero [30] are also notably absent. With the introduction of CompilerGym [7], a reinforcement learning for compiler optimization framework, experiments have largely centralized around well-supported objectives (i.e. IR instruction count), program representations (i.e. AutoPhase features, ProGraML) and benchmarks.

## III. Methods

### A. Program Representation

We based our program representation on AutoPhase features [14]. AutoPhase features are a collection of 56 features statically extracted from LLVM IR including basic block counts, constant counts, branch counts, and instruction counts. As feature scales vary, we experimented with three different approaches to feature standardization: (1) no standardization, (2) Welford's online standardization algorithm, and (3) an autoencoder. In (2), we estimate feature mean and variance online, and use these estimates for standardization. In (3), we embed AutoPhase features with a simple feed-forward autoencoder, and use it as the program representation. The autoencoder is learned online using a linear combination of the agent loss and the autoencoder loss. We also attempted to evaluate inst2vec [4] embeddings but a performance bug in the CompilerGym framework made it infeasible.

### B. Deep Q-Learning

We aim to minimize IR instruction count. That is, the cost function $C(s)$ is the number of IR instructions. We consider two reward functions: (1) the change in IR instruction count $C(s_t) - C(s_{t-1})$ and (2) the signed change in IR instruction count $\text{Sign}(C(s_t) - C(s_{t-1}))$. For (1), we also considered online standardization with Welford's algorithm.

We train a neural network, known as a deep Q-network (DQN), to approximate the Q-function

$$Q(s,a) = R(s,a) + \gamma \max_{a'} Q(T(s,a), a')$$

which is the maximum discounted possible reward from taking action $a$ in state $s$. By inspecting the value of the Q-function for each action, we can select the best optimization pass to apply at each timestep. Training consists of two alternating phases: rollout and replay.

Figure III-B illustrates the rollout phase. Given a batch of programs, we follow an epsilon-greedy policy: either we greedily sample optimization passes with the DQN, or we randomly select optimization passes. The program representation before and after each optimization pass is saved to a

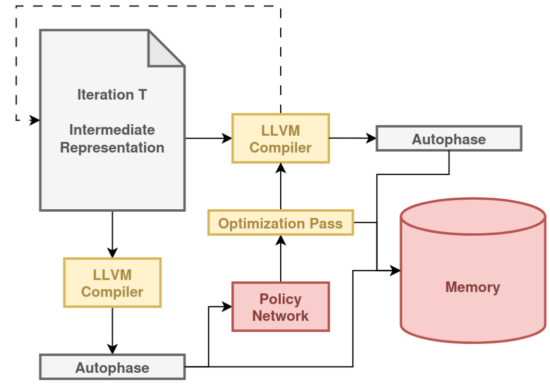


Fig. 2. Rollout phase of deep Q-learning. New training examples are generated by greedily sampling from the DQN.

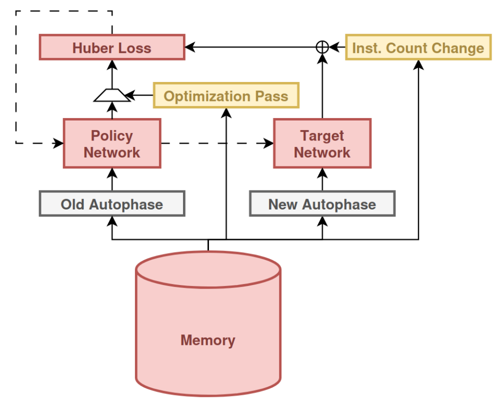


Fig. 3. Replay phase of deep Q-learning. The predicted value of taking a given action in a given state is updated to align with the actual value of taking the action and the predicted remaining value.

FIFO memory alongside the optimization pass and change in IR instruction count.

Figure III-B illustrates the replay phase. We randomly sample a batch of new and old program representations, selected optimization pass and change in IR instruction count from the memory. We estimate the expected remaining IR instruction count reduction from the new program representation using the DQN, and then train it to predict the sum of the change in IR instruction and the expected remaining IR instruction count reduction from the old program representation and selected optimization pass. We perform soft updates with a policy and target network for stability.

## IV. Experiments

### A. In-Sample

We evaluated whether the DQN could overfit small datasets and outperform `-Oz`. For these experiments, we used the change in instruction count as the reward, Autophase features for the program representation, and applied online standardization to rewards and features. As shown in Figure IV-A and Figure IV-A, the DQN was able to outperform `-Oz` in-sample.

TABLE I
SUMMARY OF PRIOR WORK

| Method | Objectives | Benchmarks | Models | Representation | Rewards | Results |
|---|---|---|---|---|---|---|
| AutoPhase [14] | Cycle count | High-Level Synthesis | PPO, A3C, Random Forest | AutoPhase features, action history | $\log \frac{C(s_{t-1})}{C(s_t)}$ | 1.28x speedup vs. `-O3` |
| CORL [19] | Runtime | LLVM test suite | Deep Q-learning | inst2vec, action history | $\log \frac{C(s_{t-1})}{C(s_t)}$ | Slowdown vs. `-O3` |
| POSET-RL [16] | Throughput, binary size | SPEC-CPU 2006, SPEC-CPU 2017, MiBench | Deep Q-Learning | IR2Vec | Linear combination of $\frac{C(s_t)-C(s_{t-1})}{C(s_{\text{-O0}})}$ for throughput, binary size | 1.04x throughput, 1.03x compression vs. `-Oz` |
| AutoPhase V2 [2] | IR instruction count | cBench, CHStone, Csmith | PPO | AutoPhase features, action history | $\frac{C(s_t)-C(s_{t-1})}{C(s_{\text{-O0}})-C(s_{\text{-Oz}})}$ | 1.00x compression vs. `-Oz` |
| GEAN [18] | IR instruction count | AnghaBench, BLAS, GitHub, Linux, OpenCV, POJ-104, TensorFlow, CLGen, Csmith, LLVM-Stress, cBench, CHStone, MiBench, NPB | Coreset, GEAN (GAT-like GNN) | ProGraML | Minimum observed cost | 1.05x compression vs. `-Oz` |
| DeCOS [5] | Cycle count | Splash-3, Parsec-3, SPEC-CPU 2017 | Unspecified RL, LLM | IR2Vec, dynamic features, action history | $C(s_t) - C(s_{t-1})$, unspecified reward for final result | 1.21x speedup vs. `OpenTuner` |
| CompilerDream [9] | IR instruction count | CodeContests, BLAS, cBench, CHStone, Linux, MiBench, NPB, OpenCV, TensorFlow | DreamerV2, guided search | AutoPhase features, action history | $\frac{C(s_t)-C(s_{t-1})}{C(s_{\text{-O0}})-C(s_{\text{-Oz}})}$ | 1.07x compression vs. `-Oz` |
| GRACE [23] | IR instruction count | BLAS, cBench, CHStone, MiBench, NPB, OpenCV, TensorFlow | Coreset, contrastive encoder, guided search | AutoPhase features | Minimum observed cost | 1.10x compression vs. `-Oz` |
| Compiler-R1 [22] | IR instruction count | BLAS, cBench, CHStone, MiBench, NPB, OpenCV, TensorFlow | LLM, GRPO, PPO, RPP | AutoPhase features | $\frac{C(s_{\text{-O0}})-C(s_t)}{C(s_{\text{-O0}})}$, format reward | 1.08x compression vs. `-Oz` |

### *B. Hyperparameter Sweep*

We conducted a hyperparameter sweep to determine the best combination of rewards and feature representation. See Table IV-B for combinations. All combinations used an initial epsilon-greedy probability of 90% linearly decayed to 1% over 32,768 steps, were trained on the AnghaBench dataset, used 64 steps per episode for 1,024 episodes with a batch size of 128 and 32 batches per episode. The memory capacity was set at 8,192 entries and filled to 50% before training began. Detailed results are provided in Figure IV-B. Notably, signed rewards stabilized the training procedure while the autoencoder did not.

TABLE II
HYPERPARAMETER SWEEP

| Hyperparameter | Value(s) |
|---|---|
| Signed Reward | Yes, No |
| Autoencoder | Yes, No |
| DQN Hidden Size | 256, 512 |
| Learning Rate | 0.0005, 0.00005 |

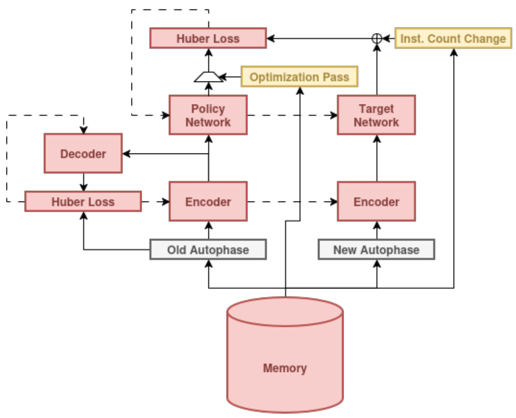


Fig. 4. Replay phase of deep Q-learning with an autoencoder. The corresponding rollout phase is similar.

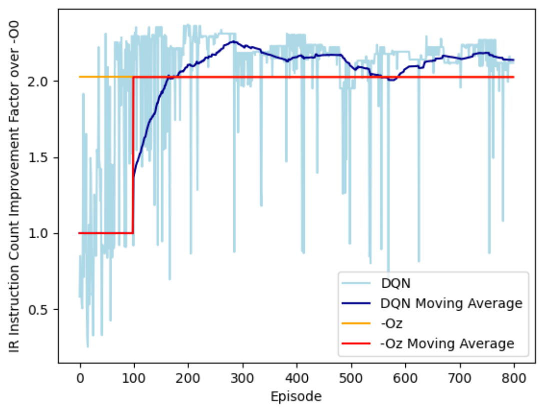


Fig. 5. In-sample performance on a quicksort function.

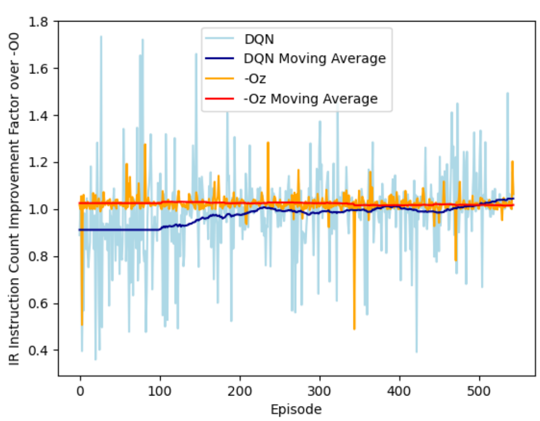


Fig. 6. In-sample performance on Tensorflow [1] objects.

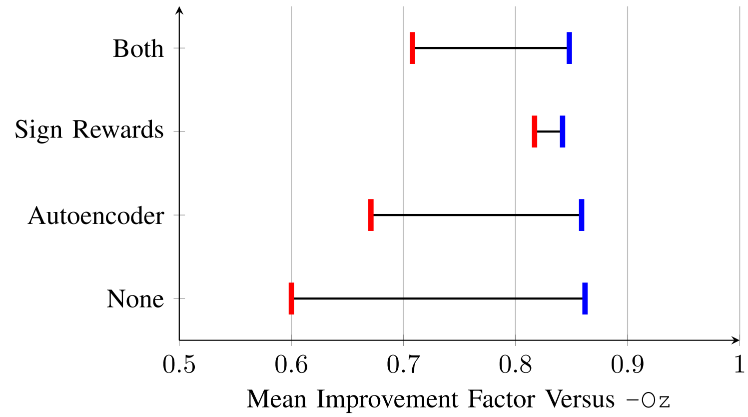


Fig. 7. Geometric mean improvement factor versus `-Oz` across six benchmark suites. Blue (red) line is maximum (minimum) improvement factor in hyperparameter sweep.

TABLE III
TRAINING AND EVALUATION DATASETS

| **Dataset** | **Samples** | **Description** |
|---|---|---|
| AnghaBench [26] | 1,041,333 | C/C++ functions extracted from GitHub |
| BLAS [3] | 300 | Linear algebra kernels |
| cBench [10] | 23 | C benchmarks |
| CHStone [15] | 12 | C-based high-level synthesis benchmarks |
| MiBench [11] | 40 | Embedded C benchmarks |
| NPB | 122 | NASA parallel benchmarks |
| OpenCV [6] | 442 | C++ objects from OpenCV |

### C. Final Model

We selected the hyperparameter combination with maximum speedup on the training set and trained it on AnghaBench for one day with 64 steps per episode. This combination used an autoencoder with the IR instruction count change reward. Due to time constraints, we could not wait for test metrics to become available. We also attempted curriculum learning, training the final model for another day with 256 steps per episode. We evaluated both the final model and the final model with curriculum learning at 64 and 256 steps for episode. Only the final model with 64 step per episode had reasonable performance, attaining 89% of the IR instruction count reduction obtained by `-Oz`. The full results are available in Figure IV-C.

## V. Future Work

### A. CompilerGym

We encountered numerous problems with the CompilerGym framework that should be fixed to support future research. In particular, the library version should be bumped, race conditions caused by multiprocessing should be eliminated, compilation should be parallelized, and the performance of built-in program representations such as inst2vec should be improved,

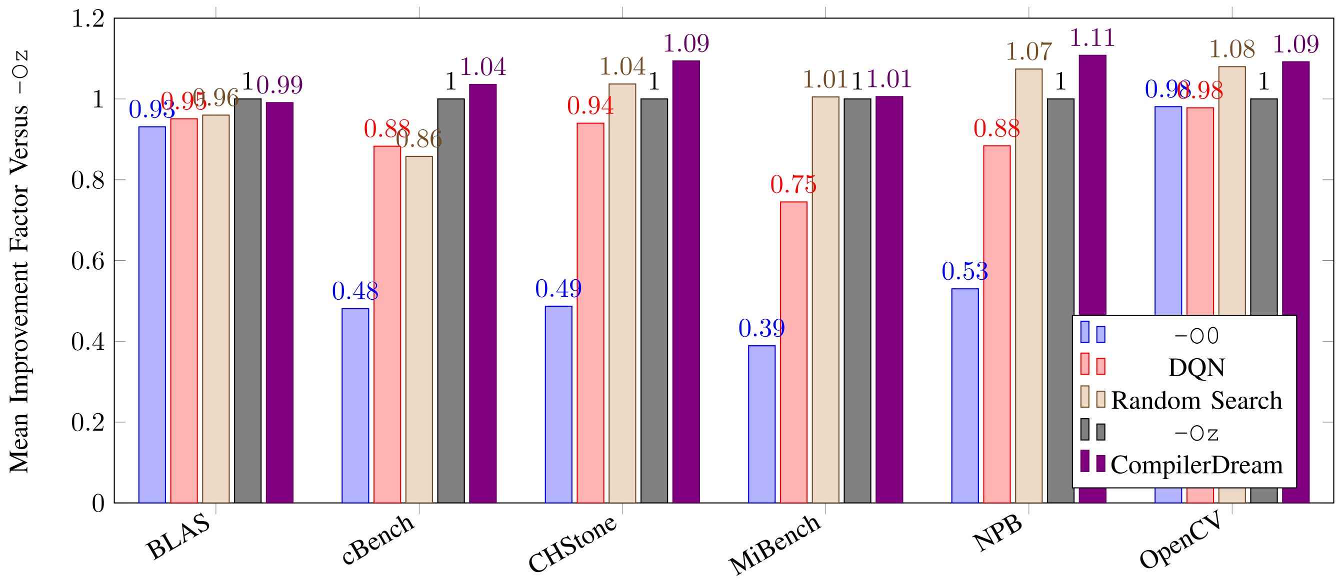


Fig. 8. DQN geometric mean IR instruction count reduction over `-Oz` on six representative benchmark suites. See Table III for additional details.

### *B. Program Representation*

End-to-end deep learning has yield incredibly successful IR-based program representations beyond handcrafted features like Autophase. Future work should evaluate modern representations such as FAIR [21] in the context of phase ordering.

### *C. Model*

Instead of deep Q-learning, future work should investigate methods in the AlphaGo [28] lineage such as AlphaZero [27], MuZero [24], MuZero Reanalyze [25] and EfficientZero [30] which offer superior sample efficiency.

### *D. Objective*

IR instruction count minimization is not a particularly useful objective. It serves primarily as a mechanism to disambiguate between different methods for ordering optimization passes. However, the task might be saturated: most methods differ by less than a percentage point improvement over `-Oz`. Future work should incorporate more relevant and underexplored objectives such as runtime and energy consumption.

## References


[1] Martín Abadi et al. "TensorFlow: a system for large-scale machine learning". In: *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2016. ISBN: 9781931971331.

[2] Mohammed Almakki et al. "Autophase V2: Towards Function Level Phase Ordering Optimization". In: *Machine Learning for Computer Architecture and Systems (MLArchSys)*. 2022.

[3] "An updated set of basic linear algebra subprograms (BLAS)". In: *ACM Transactions on Mathematical Software* (2002). DOI: 10.1145/567806.567807.

[4] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. "Neural code comprehension: a learnable representation of code semantics". In: *International Conference on Neural Information Processing Systems (NeurIPS)*. 2018.

[5] 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.

[6] Ivan Culjak et al. "A brief introduction to OpenCV". In: *International Convention MIPRO*. 2012.

[7] 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.

[8] Chris Cummins et al. "ProGraML: A Graph-based Program Representation for Data Flow Analysis and Compiler Optimizations". In: *International Conference on Machine Learning (ICML)*. 2021.

[9] 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.

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

[11] 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.

[12] Danijar Hafner et al. "Mastering Atari with Discrete World Models". In: *International Conference on Learning Representations (ICLR)*. 2021.

[13] Danijar Hafner et al. “Mastering diverse control tasks through world models”. In: *Nature* (2025). DOI: 10.1038/s41586-025-08744-2.
[14] Ameer Haj-Ali et al. “AutoPhase: Juggling HLS Phase Orderings in Random Forests with Deep Reinforcement Learning”. In: *Machine Learning and Systems (MLSys)*. 2020.
[15] Yuko Hara et al. “CHStone: A benchmark program suite for practical C-based high-level synthesis”. In: *International Symposium on Circuits and Systems (ISCAS)*. 2008. DOI: 10.1109/ISCAS.2008.4541637.
[16] 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.
[17] 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.
[18] Youwei Liang et al. “Learning compiler pass orders using coreset and normalized value prediction”. In: *International Conference on Machine Learning (ICML)*. 2023.
[19] 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.
[20] Volodymyr Mnih et al. “Human-level control through deep reinforcement learning”. In: *Nature* (2015). ISSN: 00280836.
[21] Changan Niu et al. “FAIR: Flow Type-Aware Pre-Training of Compiler Intermediate Representations”. In: *International Conference on Software Engineering (ICSE)*. 2024. DOI: 10.1145/3597503.3608136.
[22] Haolin Pan et al. *Compiler-R1: Towards Agentic Compiler Auto-tuning with Reinforcement Learning*. 2025. arXiv: 2506.15701.
[23] Haolin Pan et al. *GRACE: Globally-Seeded Representation-Aware Cluster-Specific Evolution for Compiler Auto-Tuning*. 2025. arXiv: 2510.13176.
[24] 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.
[25] 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.
[26] 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.
[27] 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.
[28] David Silver et al. “Mastering the game of Go with deep neural networks and tree search”. In: *Nature* (2016). DOI: 10.1038/nature16961.
[29] Zheng Wang and Michael O’Boyle. “Machine Learning in Compiler Optimization”. In: *IEEE* (2018). DOI: 10.1109/JPROC.2018.2817118.
[30] Weirui Ye et al. “Mastering Atari Games with Limited Data”. In: *International Conference on Neural Information Processing Systems (NeurIPS)*. 2021.