Supercompiler Code Optimization with Zero-Shot Reinforcement Learning

Jialong Wu[1†], Chaoyi Deng[1†], Jianmin Wang[1] and Mingsheng Long[1*]

[1] School of Software, BNRist, Tsinghua University.

*Corresponding author(s). E-mail(s): mingsheng@tsinghua.edu.cn;
†These authors contributed equally to this work.

Abstract

Effective code optimization in compilers plays a central role in computer and software engineering. While compilers can be made to automatically search the optimization space without the need for user interventions, this is not a standard practice since the search is slow and cumbersome. Here we present CodeZero, an artificial intelligence agent trained extensively on large data to produce effective optimization strategies instantly for each program in a single trial of the agent. To overcome the huge range of possible test programs, we prepare a large dataset of training programs that emphasize quality, naturalness, and diversity. To tackle the vast space of possible optimizations, we adapt deep reinforcement learning to train the agent in a sample-efficient manner through interacting with a world model of the compiler environment. Evaluation on both benchmark suites and production-level code optimization problems demonstrates our agent's supercompiler performances and zero-shot generalization abilities, outperforming built-in optimization options designed by compiler experts. Our methodology kindles the great potential of artificial intelligence for engineering and paves the way for scaling machine learning techniques in the realm of code optimization.

1 Introduction

Entering the post-Moore's law era, code optimization is crucial for computer and software engineering, which plays an important role in realizing the full potential of slow-growing hardware. Developers typically rely on a compiler's ability to transform input programs into semantically equivalent but more efficient versions, improving metrics like execution time, code size, and power consumption. For example, standard optimization options `-O1`, `-O2`, and `-O3` aim to reduce execution time, while the `-Os` and `-Oz` options are crafted to reduce code size. Still, it is not common for users to explore beyond these conventional compiler options. Given the vast diversity of programs and platforms, coupled with the increasing number of optimization passes integrated into compiler frameworks, these off-the-shelf optimization strategies predefined heuristically by compiler experts may struggle to guarantee near-optimal performance across ever-changing scenarios [1, 2].

Automatic code optimization is therefore crucial in compilers. Autotuning [3] improves code by systematically searching the optimization space through iterative executing and profiling optimization strategies. This search technique can yield remarkable performance gains but must be rerun for each new program with thousands of compilations, which is too time-consuming to be practical for all but a few specialized use cases. Meanwhile, machine learning techniques hold the capabilities to generalize the optimization strategy of one program to other similar ones, thereby facilitating faster code optimization. A direct method is to use supervised learning to predict good optimization strategies of input programs [4–6], which is impractical due to prohibitive computations to construct labeled training data by search. Another more promising routine, reinforcement learning, that successfully discovered faster sorting algorithm (AlphaDev [7]) and matrix multiplication algorithm (AlphaTensor [8]), can explore the optimization space from feedback on optimization metrics without requiring optimal labeled data. For both techniques, broad generalization across different programs, even out of the training samples, arises as a major bottleneck. The community has noted that in a range of machine learning applications [9–12], training high-capacity models on large-scale datasets has yielded unprecedented performances. For example, large language models like GPT-4 have demonstrated impressive zero-shot generalization abilities [13, 14]. However, in the area of code optimization, the current practice is to learn optimization heuristics in a per-program manner [15, 16] or from a small training set with hundreds of programs [17, 18], lagging far behind the new era of solving challenging problems by scaling up machine learning models.

In this study, we focus on the LLVM [19] phase ordering problem, a longstanding challenge for compiler research, and propose CodeZero, a reinforcement-learned code optimization agent, capable of generating a sequence of optimization passes tailored to a particular input program. The

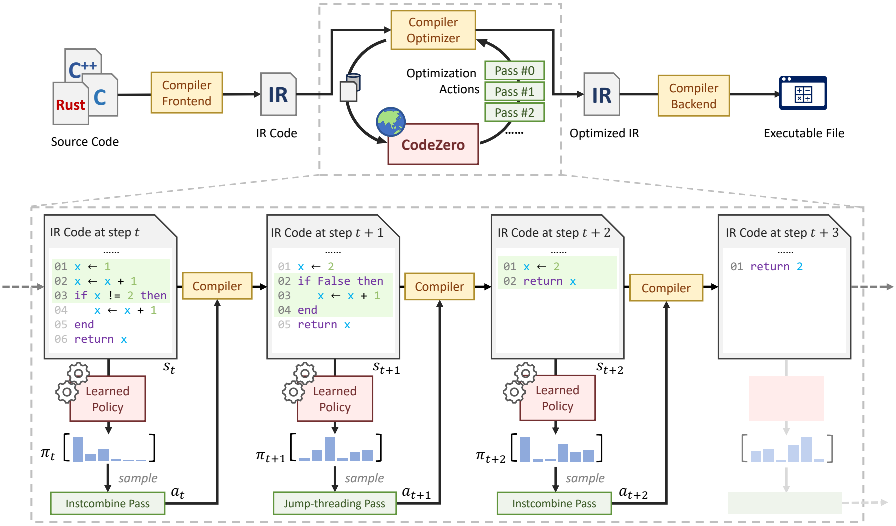

Fig. 1: **CodeZero agent performs code optimization using a learned policy in a compiler environment**. Guided by a learned policy, at each step, CodeZero analyzes the Intermediate Representation (IR) of the program, selects to apply an optimization pass, and receives a reward based on the improvement in optimization metrics. Through sequential interactions, the agent aims to maximize cumulative rewards and enhance the final IR's performance.

problem is formulated such that a code optimization agent, upon encountering a program, selects a series of optimization passes, and receives feedback based on the outcomes of applying these transformations, from the compiler environment. To tackle the huge range of possible test programs, we aim to enhance the generalization ability of trained agents by assembling a large-scale training dataset of natural programs. Several previous works rely on randomly generated programs due to data scarcity but the significant distribution shift from random-generated and human-written programs can even hurt the generalization of learned agents when tested on real-world scenarios [20]. Our training data not only includes real-world programs sourced from GitHub [21] but also incorporates complex algorithmic solutions of competitive programming [22] and diverse programs generated by large language models (LLM) [23]. To tackle the vast space of possible optimizations ($\sim 10^{73}$ sequences), we employ a state-of-the-art model-based reinforcement learning method [24] to train the code optimization agent sample-efficiently. This method not only learns a predictive world model of compilers to reduce the amount of real compiler executions but also benefits generalization by learning representations that better capture the structure of the compiler state transitions [25]. After training on massive programs via trial-and-error, the code optimization agent can generalize in a zero-shot way to previously unseen programs with superior optimization performance against off-the-shelf compiler heuristics.

We demonstrate the effectiveness of our trained CodeZero agent on a range of domains [20] from benchmark suites encompassing fundamental algorithms that are ubiquitously employed in everyday applications, to production-level open-source programs, including object files from C++ TensorFlow [26] and OpenCV [27] library. On six test datasets, our agent can produce optimization sequences in a single trial, yielding more efficient code size reduction compared to the `-Oz` flag. Detailed analysis underscores the effectiveness of both the newly introduced program datasets and the model-based reinforcement learning technique within the realm of code optimization. We posit that the agents developed through our approach could be integrated into the existing toolkits of optimization strategy in compilers alongside other manually designed heuristics, such as `-Oz` or `-O3`.

2 *CodeZero* Agent

2.1 Code Optimization as Decision Making

As illustrated in Figure 1, compilers consist of three main components: the front end which translates the source code into an intermediate representation (IR), the middle end, and the back end which converts IR to the binary code. The middle end is responsible for language- and platform-agnostic optimizations over the IR, implemented as *passes* to either collect information about the program or apply a transform on it, like function inlining, loop unrolling, and dead code elimination. For example, the LLVM-`opt` tool has more than 100 optimization passes available. A specific order of applying these passes forms an *optimization sequence* for an input program. This is critical as the right selection and ordering of passes can significantly boost the program's performance. Despite that compiler developers have provided standard optimization sequences at various levels, e.g. `-O2`, `-O3`, `-Oz`, these preset sequences may not always yield optimal results, especially for emerging programs written in new frameworks such as TensorFlow. Particularly due to the increasing number of optimization passes, it is an open challenge to determine the most effective sequence for each program.

This problem, known as *phase ordering*, can be naturally formulated as a *partially observable Markov decision process* (POMDP) [28]. In this formulation, a code optimization agent determines the optimization sequence of an input program through a series of interactions with the compiler environment, guided by its policy π. The process starts at the initial state s_0, representing the IR of the program to be optimized, which is randomly sampled from all IRs of interest. It is critical that aiming to capture the most important characteristics of the target optimization, the agent only receives partially observable information o_0 of the state. The observation space can vary widely, ranging from manually designed features (e.g. the number of basic blocks) [15] to more complex tree-based or graph-based program representations [29, 30], and even raw text strings of IR [31]. At each time step $t = 0, 1, 2, \ldots$, the agent takes an action a_t based on its policy, corresponding to selecting an optimization pass. Following this action, the agent receives an immediate reward r_{t+1}, reflecting changes in the optimization metrics, and an observation o_{t+1} of the next state $s_{t+1} = p(s_t, a_t)$ which represents the IR transformed by the compiler using the selected pass. This process can be terminated either when the agent finds no positive gains can be achieved or a maximum number of steps is reached. The goal of the agent is to learn an optimization policy $\pi(a_t \mid o_{\leq t})$ that effectively maximizes the cumulative rewards, thereby optimizing the performance metrics of the final IR.

2.2 Large-Scale Data Preparation

To ensure that our *CodeZero* agent can effectively generalize to unseen situations, a concept known as zero-shot generalization, we have identified three critical factors in preparing our training dataset. Firstly, the dataset must reflect naturalness. Training data should be within the distribution of human-written programs, otherwise, overfitting to programs that deviate significantly from this, such as those generated by tools like Csmith [32] and llvm-stress [19], could provide no benefits or even hurt the generalization to real-world scenarios. Secondly, diversity in the dataset is essential. It should encompass a wide range of human-written program styles and structures, ensuring globally comprehensive coverage of possible scenarios. Lastly, the pursuit of high-quality training data is imperative. Existing large-scale program collections [21, 29] have been proposed to serve as training data [20] but are proven unfruitful in our preliminary experiments. Instead, we focus on data that features moderate lengths of IR, complex algorithmic logic, and potential room for optimizations. This allows the agent to flexibly explore and understand the local transformation structure within the IR space. Collectively, these three properties – naturalness, diversity, and high quality – contribute to aligning the visited IR distribution during training with that will be encountered in real-world applications.

We construct our training dataset containing hundreds of thousands of programs by combining three distinct single-source program datasets, CodeContests [22], FormAI [23], and AnghaBench [21]. CodeContests comprises human-written solutions to competitive programming problems with complex and optimizable algorithmic logic. FormAI is a large collection of AI-generated C programs with various functionality types and coding styles, aiming to enrich the dataset's diversity. AnghaBench is a collection of real-world C programs extracted from GitHub. As shown by visualization in Figure D2, our training data has a broad coverage of evaluation programs in a variety of domains. Future expansions of the dataset can be conducted following the aforementioned principles.

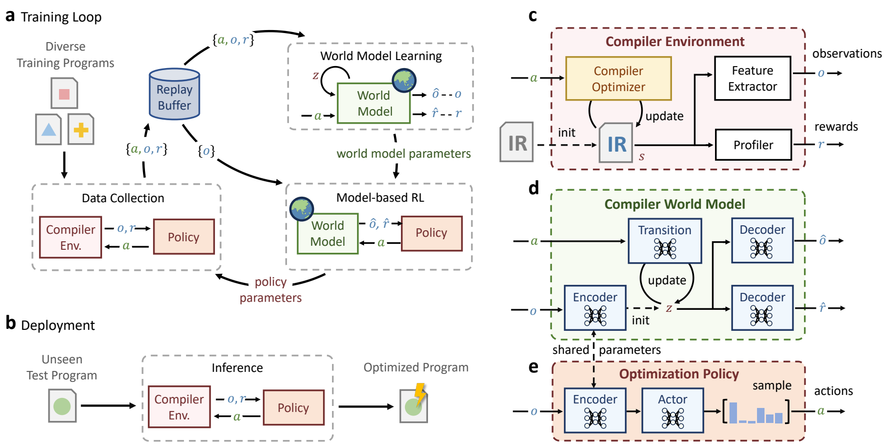

Fig. 2: **Design overview of CodeZero. (a)** In the model-based training loop, the CodeZero agent interacts with the compiler environment across diverse training programs, learns a world model from historical experience, and updates its policy efficiently through model simulations. **(b)** Deployment of the trained agent is capable of zero-shot generalization to unseen programs, delivering effective optimizations. **(c)** The compiler environment is set up with the input program's IR, s_0. At each step, upon receiving an optimization pass a_t, the environment executes the pass internally, resulting in a transformed IR s_{t+1}, and provides the agent with relevant IR features o_{t+1} and immediate rewards r_{t+1} based on optimization metric improvements. **(d)** The world model simulates the compiler environment, initiated with an observation, maintaining its internal states, and predicting future observations and rewards in response to input actions. **(e)** The policy shares with the world model a generalizable representation that captures the environment's structure.

2.3 Agent Training with Model-based RL

While existing work on reinforcement learning for code optimization prevalently focuses on model-free RL methods, model-based RL can offer advantages in terms of both sample efficiency [28] and generalization [25]. Executing and profiling extensive optimization sequences, especially for the runtime metric, can be time-consuming. This is further compounded when constructing complex observations, such as control-data flow graphs [29, 33]. Model-based RL addresses these challenges by learning a world model to approximate state transitions and reward signals of the environment. This allows the agent to learn its policy by simulating trajectories based on model predictions, rather than relying solely on trial-and-error interactions in the real compiler environment. This approach thus improves sample efficiency. Moreover, as the policy can share the representation with the world model, model learning can act as an auxiliary task and thus aid in learning representations that better capture the structure of the environment and manifest in better generalization of the policy [34].

We train the *CodeZero* agent by adapting an advanced model-based RL method, Dreamer [24], as depicted in Figure 2. This approach involves learning a predictive world model $(\hat{p}_\theta, \hat{r}_\theta)$ of the compiler environment by approximating the underlying transition dynamics $p(o_{t+1}|o_{\leq t}, a_{\leq t})$ and reward function $r(o_{\leq t}, a_{\leq t})$. Through imaginary rollouts using this world model over a horizon H, the policy $\pi_\psi(a_t|o_{\leq t})$ can be learned using the REINFORCE policy gradients [35] with an entropy regularizer H. The training objective is formulated as

$$\mathcal{L}(\psi) \doteq \mathbb{E}_{\hat{p}_\theta, \hat{r}_\theta, \pi_\psi}\left[\sum_{t=1}^{H} -\left(V_t - v_\xi\left(o_{\leq t}\right)\right)\log \pi_\psi\left(a_t \mid o_{\leq t}\right) - \eta \mathrm{H}\left[\pi_\psi\left(o_{\leq t}\right)\right]\right], \tag{1}$$

where V_t is the imagined return estimation and the value prediction v_ξ serves as a baseline for variance reduction [36]. The value prediction is learned separately by regressing the cumulative rewards that the agent should expect to receive from the current state. All components are realized as deep neural networks.

After extensive training on a wide range of programs, our agent demonstrates the capability to generalize “zero-shot” — that is, to be effective without further training — to new, unseen programs across various domains.

3 Results

3.1 Evaluation Benchmarks

Our experiments focus on code size reduction, which benefits applications targeting low-resource hardware such as embedded systems. This focus is driven by the practical advantages of code size as a metric: it is both cost-effective and convenient to construct extensive compilable training and test datasets and to evaluate the optimization performance for code size.

We evaluate our method on benchmarks from the CompilerGym platform [20]: benchmark suites including cBench [37], CHStone [38], MiBench [39], and NASA Parallel Benchmarks (NPB) [40], in addition to kernels from open source projects including BLAS [41], Linux, OpenCV [27], and TensorFlow [26]. We disregard benchmarks generated by synthetic program generators [19, 32] since they do not align with real-world scenarios.

3.2 Code Optimization Results

Figure 3 reports CodeZero’s performance in reducing code size, measured in terms of IR instruction count, compared to the expert-designed `-Oz` flag in LLVM. Remarkably, CodeZero, without any specific training on in-domain programs, outperforms `-Oz` across all but two test benchmark datasets in a

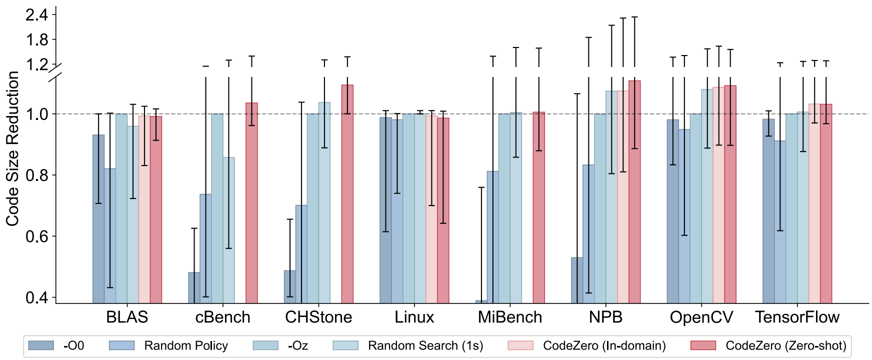

Fig. 3: **Code size reduction in terms of IR instruction count over LLVM `-Oz` under different methods.** Bars indicate the geometric mean and min-max range across test programs in each benchmark dataset.

Table 1: **Top performances of the zero-shot CodeZero agent on individual programs.** The agent's optimization sequences are streamlined, omitting passes that do not contribute to performance enhancement. Full IRs of these programs before and after optimizations are shown in the Supplementary Information.

Dataset	Program	Pass Sequences by CodeZero Agent	Code size `O0`	CodeZero	`Oz` (Reduction)
cBench	sha	-sroa -gvn -instcombine -simplifycfg -licm -early-cse -simplifycfg -indvars -gvn -simplifycfg -memcpyopt -reassociate	799	349	500 (**1.43**×)
	bzip2	-sroa -gvn -simplifycfg -instcombine -simplifycfg -early-cse -simplifycfg -reassociate -memcpyopt -jump-threading -functionattrs -licm -gvn -simplifycfg -reassociate -early-cse -jump-threading -early-cse -instcombine -simplifycfg	28748	13565	15946 (**1.18**×)
	qsort	-sroa -gvn -simplifycfg -instcombine -simplifycfg -early-cse -gvn -indvars -gvn -simplifycfg	638	280	315 (**1.12**×)
OpenCV	#41	-lowerinvoke -simplifycfg -globalopt	28	18	28 (**1.56**×)
	#9	-sroa -gvn -sroa -early-cse -simplifycfg -loop-deletion -lowerinvoke -simplifycfg -sroa -early-cse -instcombine -simplifycfg -early-cse -memcpyopt -early-cse -simplifycfg -jump-threading -early-cse -reassociate -instcombine	9510	6341	9269 (**1.18**×)
TensorFlow	#17	-sroa -gvn -simplifycfg -lowerinvoke -simplifycfg -sroa -early-cse -indvars -simplifycfg	5512	4247	5450 (**1.28**×)
	#6	-gvn -simplifycfg -lowerinvoke -simplifycfg -instcombine -early-cse -gvn -jump-threading -instcombine	16173	12643	16076 (**1.27**×)

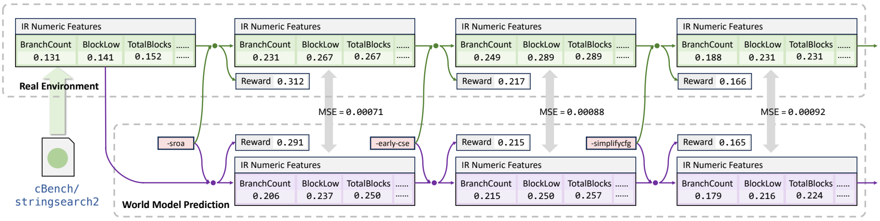

Fig. 4: **A comparison between a ground-truth code optimization trajectory and an imagined trajectory by a learned compiler world model.** The learned world model accurately captures the variations of program features and optimization metrics.

single trial. The marginal performance variations between different methods on the BLAS and Linux datasets suggest that they are already highly optimized. CodeZero also demonstrates the ability to match or even slightly surpass a competitive autotuning method, random search, within a similar time budget (in the order of seconds), while the latter blindly aggregates hundreds of trials. This notable performance, coupled with the significant advantage over a single random trial, underscores the effectiveness of our agent's policy learning. Moreover, CodeZero's zero-shot generalization capability either matches or exceeds the performance of in-domain counterparts, which are trained using the train subset of each benchmark dataset. This is particularly significant in cases like the NPB dataset, which contains only 22 training programs. In such scenarios, where data sparsity presents a challenge for in-domain agents, the zero-shot CodeZero notably achieves an extra 3% reduction in code size. Notably, CodeZero's pre-trained policy can generalize to IRs compiled from novel programming languages beyond C and C++, as evidenced in its performances on the BLAS and NPB datasets of Fortran programs. Further, in an AI-generated benchmark dataset of Objective-C, another language supported by LLVM-Clang apart from C/C++, CodeZero successfully improves upon LLVM `-Oz`, achieving an average code size reduction of 1.027$\times$, and reaching up to 2.87$\times$ in certain instances.

3.3 Program Case Study

We analyze the internal behavior of our CodeZero agent to optimize IR code via recording the sequence of passes chosen by the policy on individual programs. Table 1 presents the agent's top performance outcomes on various benchmark datasets. We observe that the agent indeed produces a specialized optimization strategy tailored for each program. Additionally, these results also highlight certain passes, such as *-sroa* (scalar replacement of aggregates), *-gvn* (global value numbering), and *-simplifycfg*, as particularly effective in code size optimization.

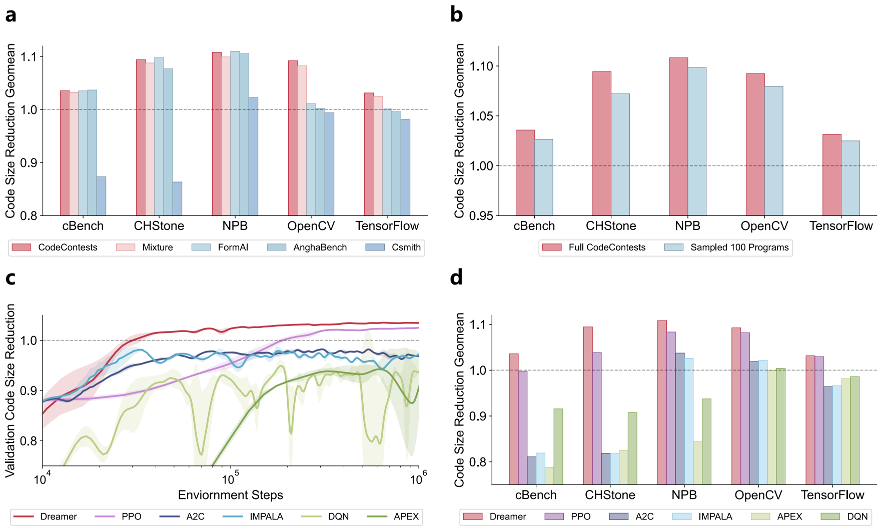

Fig. 5: **Evaluations of CodeZero with different datasets (a-b) and RL algorithms (c-d). (a)** Performance of CodeZero trained on various datasets, with "Mixture" representing a combined dataset of CodeContests, FormAI, and AnghaBench. **(b)** Performance of CodeZero when trained on the full CodeContests dataset versus a subset of 100 randomly selected training programs from CodeContests. **(c)** Learning curves of various RL algorithms, measured by the geometric mean of code size reduction on the CodeContests validation set. A Gaussian filter ($\sigma = 2.0$) is applied to enhance the visualization of trends. **(d)** Zero-shot generalization capabilities of different RL algorithms on various test benchmark datasets.

In Figure 4, we display a predicted optimization trajectory for an unseen program from cBench, as forecasted by our learned compiler world model. The model successfully forecasts numeric features of future IR, including the counts of branches and blocks, alongside future rewards that signify optimization outcomes. This instance exemplifies the capability of our learned compiler world model to serve as a viable alternative for a real compiler environment in training code optimization agents.

3.4 Effect of Training Dataset

In our quest to understand the impact of training datasets on generalization, we explored how different datasets from various domains and sizes influence performance. As shown in Figure 5a, we discovered that the CodeContests dataset stands out as the most effective for generalizing to our test benchmarks. Although comparable outcomes were observed with other datasets like FormAI and AnghaBench, agents trained on these datasets notably lag in performance when tested on OpenCV and TensorFlow benchmarks. Moreover, combining these datasets does not yield performance enhancement. Consequently, we present in Figure 3 the results based on the agent trained exclusively with the CodeContests dataset. These results underscore the significance of high-quality training data, a principle that is increasingly recognized as vital in other scalable machine learning applications, such as large language models. In Figure 5b, we compare the agent trained on the full CodeContests dataset versus the one trained on a 100-program subset sampled from the same dataset. This subset is comparable in size to the datasets used in previous research [17, 18] that applies reinforcement learning to code optimization. The suboptimal performance on this smaller dataset highlights its inadequacy for training a strong zero-shot generalizable agent and emphasizes the significance of a larger and more diverse training set prepared by this study.

3.5 Sample Efficiency and Zero-Shot Generalization

We further evaluate the sample efficiency and zero-shot generalization abilities of CodeZero based on the world model algorithm, Dreamer, against a range of model-free counterparts. These include notable algorithms such as DQN [42], A2C [43], APEX [44], IMPALA [45], and PPO [46] (see Appendix B.4 for more details). Figure 5c shows that while PPO is the most competitive among the model-free baselines, Dreamer outperforms it by learning an order of magnitude faster, requiring fewer interactions with the compiler environment to achieve excellent performance. Additionally, as Figure 5d indicates, Dreamer excels in generalizing to unseen test benchmarks, outstripping its counterparts. These findings verify our hypothesis that a world model-based agent holds a stronger capability to capture the environment's structure and can effectively enhance zero-shot generalization.

4 Discussion

We tackled the major challenge of zero-shot generalization that arises when applying artificial intelligence techniques to improve code optimization. We introduce the CodeZero agent that leverages the strengths of a diverse, high-quality training program dataset, as well as the sample efficiency and generalization capabilities of world model-based reinforcement learning. Our results have shown that CodeZero, with its zero-shot generalization ability, attains supercompiler code optimization performance, in the challenging phase ordering problem for code size reduction. Our study provides an AI-centric framework and serves as a meaningful step towards scaling machine learning techniques for code optimization. There is a substantial scope for further exploration, including expansion of the training dataset, scaling up the compiler world models, optimizing multiple objectives like execution time, and enriching feature and action spaces with deeper expert knowledge or large language models.

5 Methods

In this section, we provide the mathematical and computational details of our approach.

5.1 LLVM Phase Ordering POMDP

We formulate the phase ordering problem of LLVM as a partially observable Markov decision process (POMDP) $\mathcal{M} = (\mathcal{S}, \mathcal{A}, r, p, \mu, \mathcal{O}, \phi)$. The state space $\mathcal{S}$ covers all possible Intermediate Representations (IRs), the action space $\mathcal{A}$ comprises individual compiler optimization passes, and the reward function r is defined by the metric being optimized. The transition dynamics $p : \mathcal{S} \times \mathcal{A} \mapsto \mathcal{S}$ represents the outcome of applied IR transformations. The initial state distribution $\mu \in \Delta(\mathcal{S})$ captures all IRs of interest, which can be approximated via uniform sampling from the training dataset. The observation function $\phi : \mathcal{S} \mapsto \mathcal{O}$ maps the underlying IR into the observation space. Complex observation spaces offer comprehensive program information, while expert-designed features embed problem-specific knowledge, potentially enhancing the optimization policy's generalization by eliminating irrelevant details. The code optimization agent with a policy π interacts with the compiler environment according to the protocol described in Section 2.1. We fix the horizon of interactions to 45 steps.

Following Autophase [15] and CompilerGym [20], the *action space* consists of 42 optimization passes out of over a hundred in LLVM, which effectively prunes the vast optimization space while not sacrificing the performance of the learned policy. The *observation space* is a concatenation of two numeric features: the 56-dimension Autophase feature summarizing the statistics of the IR, such as the number of specific basic blocks, branches, and instructions, and a 42-dimension histogram of the agent's previous actions. The *reward function* is defined as the normalized change of the optimization metric $C(s)$:

$$r_{t+1} = \frac{C(s_t) - C(s_{t+1})}{C(s_0) - C(s_{\mathrm{b}})}, \tag{2}$$

where a lower C indicates better performance and $C(s_{\mathrm{b}})$ stands for the performance of a baseline policy, such as built-in `-Oz` or `-O3`. A total reward greater than 1 means that the optimization sequence performs better than the baseline policy.

5.2 Training Dataset

We construct our training datasets, by considering three datasets from distinct domains. The CodeContests dataset [22] consists of over 13,000 competitive problems, each on average having hundreds of solutions in three languages. We subsample up to ten C++ solutions for each training problem, resulting in 110,240 programs, as part of our training data, and sample one solution for each of 100 test problems from CodeContests as our validation data. We also include the full Form-AI dataset [23] with 112,000 AI-generated programs and the collection of the largest 9,998 single-function programs and 15,264 multiple-function programs from the AnghaBench [21] into our training data.

5.3 Model-based RL Method

We utilize a deep model-based RL method, DreamerV3 [24], to model the compiler environment upon which it learns an effective policy that maps observations to actions with parameterized deep neural networks.

World Model

The world model simulating the compiler environment is formulated as a latent dynamics model with the following four components:

$$\begin{array}{ll} \text{Representation model:} & z_t \sim q_\theta(z_t \mid z_{t-1}, a_{t-1}, o_t) \\ \text{Transition model:} & \hat{z}_t \sim p_\theta(\hat{z}_t \mid z_{t-1}, a_{t-1}) \\ \text{Image decoder:} & \hat{o}_t \sim p_\theta(\hat{o}_t \mid z_t) \\ \text{Reward decoder:} & \hat{r}_t \sim p_\theta(\hat{r}_t \mid z_t) \end{array} \tag{3}$$

The representation model estimates the latent state z_t based on the previous state z_{t-1}, the previous action a_{t-1} and the current observation o_t, while the transition model predicts it directly from z_{t-1} and a_{t-1}. The overall models are jointly learned by minimizing the negative evidence lower bound (ELBO) [47, 48]:

$$\begin{aligned} \mathcal{L}_{\text{model}}(\theta) \doteq \ \mathbb{E}_{q_\theta(z_{1:T} \mid a_{1:T}, o_{1:T})} \Big[\sum_{t=1}^{T} \Big(& -\ln p_\theta(o_t \mid z_t) - \ln p_\theta(r_t \mid z_t) \\ & + \mathrm{KL}\left[q_\theta(z_t \mid z_{t-1}, a_{t-1}, o_t) \,\|\, p_\theta(\hat{z}_t \mid z_{t-1}, a_{t-1}) \right] \Big) \Big]. \end{aligned} \tag{4}$$

In practice, we employ a reward smoothing technique [49] to mitigate the sparsity and the long tail distribution of rewards during an episode. This is achieved through the following equation:

$$r'_t \leftarrow \alpha r'_{t-1} + (1-\alpha) r_t, \quad t = 1, 2, \ldots \tag{5}$$

with $\alpha \in [0, 1)$. Consequently, we train a reward decoder $p_\theta(\hat{r}'_t \mid z_t)$ to predict the smoothed rewards.

Actor-Critic Learning

The actor and critic neural networks are parameterized on top of the latent representations:

$$\text{Actor: } \hat{a}_t \sim \pi_\psi\left(\hat{a}_t \mid \hat{z}_t\right) \qquad \text{Critic: } v_\xi\left(\hat{z}_t\right) \approx \mathbb{E}_{p_\theta, \pi_\psi}\left[\sum\nolimits_{\tau \geq t} \gamma^{\tau - t} \hat{r}_\tau\right]. \tag{6}$$

The actor and critic are jointly trained on the same *imagined* trajectories $\{\hat{z}_\tau, \hat{a}_\tau, \hat{r}_\tau\}$ with horizon H, generated by the interactions between the transition model and reward model in Eq. (3) and the actor: starting at the latent state $\hat{z}_t = z_t$, at each step $\tau = t, t+1, t+2, \ldots$, the policy takes an action $\hat{a}_\tau \sim \pi_\psi\left(\hat{a}_\tau \mid \hat{z}_\tau\right)$, and transits to the next latent state $\hat{z}_{\tau+1} \sim p_\theta(\hat{z}_{\tau+1} \mid z_\tau, a_\tau)$ with a reward $\hat{r}_{\tau+1} \sim p_\theta(\hat{r}_{\tau+1} \mid \hat{z}_{\tau+1})$. The critic is trained to predict the λ-return [28] through a discrete regression loss [24, 50]:

$$\mathcal{L}_{\text{critic}}(\xi) \doteq \mathbb{E}_{p_\theta, \pi_\psi}\left[\sum_{\tau=t}^{t+H} -\log v_\xi(V_\tau^\lambda \mid \hat{z}_\tau)\right], \tag{7}$$

$$V_\tau^\lambda \doteq \hat{r}_\tau + \gamma \begin{cases} (1-\lambda) v_\xi(\hat{z}_{\tau+1}) + \lambda V_{\tau+1}^\lambda & \text{if } \tau < t+H \\ v_\xi(\hat{z}_{\tau+1}) & \text{if } \tau = t+H. \end{cases} \tag{8}$$

The actor, meanwhile, is trained to maximize the imagined return through the REINFORCE policy gradient [35]:

$$\mathcal{L}_{\text{actor}}(\psi) \doteq \mathbb{E}_{p_\theta, \pi_\psi}\left[\sum_{\tau=t}^{t+H}\left(-\left(V_\tau^\lambda - v_\xi\left(\hat{z}_\tau\right)\right)\log \pi_\psi\left(\hat{a}_\tau \mid \hat{z}_\tau\right) - \eta\, \mathrm{H}\left[\pi_\psi(\hat{z}_\tau)\right]\right)\right], \tag{9}$$

where $\mathrm{H}\left[\pi_\psi(\hat{z}_\tau)\right]$ is an entropy regularization which encourages exploration, and η is a hyperparameter that adjusts the regularization strength.

References

[1] Triantafyllis, S., Vachharajani, M., Vachharajani, N. & August, D. I. Compiler optimization-space exploration. *International Symposium on Code Generation and Optimization (CGO)* 204–215 (2003).

[2] Georgiou, K., Blackmore, C., Xavier-de Souza, S. & Eder, K. Less is more: Exploiting the standard compiler optimization levels for better performance and energy consumption. *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems* 35–42 (2018).

[3] Bodin, F., Kisuki, T., Knijnenburg, P., O'Boyle, M. & Rohou, E. Iterative compilation in a non-linear optimisation space. *Workshop on profile and feedback-directed compilation* (1998).

[4] Calder, B. *et al.* Evidence-based static branch prediction using machine learning. *ACM Transactions on Programming Languages and Systems (TOPLAS)* **19** (1), 188–222 (1997).

[5] Fursin, G. *et al.* Milepost gcc: machine learning based research compiler. *GCC summit* (2008).

[6] Zanella, A. F., da Silva, A. F. & Quintão, F. M. Yacos: a complete infrastructure to the design and exploration of code optimization sequences. *Proceedings of the 24th Brazilian Symposium on Context-Oriented Programming and Advanced Modularity* 56–63 (2020).

[7] Fawzi, A. *et al.* Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature* **610** (7930), 47–53 (2022).

[8] Mankowitz, D. J. *et al.* Faster sorting algorithms discovered using deep reinforcement learning. *Nature* **618** (7964), 257–263 (2023).

[9] Devlin, J., Chang, M.-W., Lee, K. & Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. *North American Chapter of the Association for Computational Linguistics* (2019).

[10] Brown, T. *et al.* Language models are few-shot learners. *Advances in Neural Information Processing Systems (NeurIPS)* **33**, 1877–1901 (2020).

[11] Radford, A. *et al.* Learning transferable visual models from natural language supervision. *International Conference on Machine Learning (ICML)* 8748–8763 (2021).

[12] Kirillov, A. *et al.* Segment anything. *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)* 4015–4026 (2023).

[13] Ouyang, L. *et al.* Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems (NeurIPS)* **35**, 27730–27744 (2022).

[14] OpenAI. GPT-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[15] Haj-Ali, A. *et al.* Autophase: Juggling HLS phase orderings in random forests with deep reinforcement learning. *Proceedings of Machine Learning and Systems (MLSys)* (2020).

[16] Shahzad, H. *et al.* Reinforcement learning strategies for compiler optimization in high level synthesis. *2022 IEEE/ACM Eighth Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)* 13–22 (2022).

[17] Mammadli, R., Jannesari, A. & Wolf, F. Static neural compiler optimization via deep reinforcement learning. *2020 IEEE/ACM 6th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC) and Workshop on Hierarchical Parallelism for Exascale Computing (HiPar)* 1–11 (2020).

[18] Jain, S., Andaluri, Y., VenkataKeerthy, S. & Upadrasta, R. POSET-RL: Phase ordering for optimizing size and execution time using reinforcement learning. *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)* 121–131 (2022).

[19] Lattner, C. & Adve, V. Llvm: A compilation framework for lifelong program analysis & transformation. *International Symposium on Code Generation and Optimization (CGO)* 75–86 (2004).

[20] Cummins, C. *et al.* Compilergym: Robust, performant compiler optimization environments for ai research. *International Symposium on Code Generation and Optimization (CGO)* 92–105 (2022).

[21] Da Silva, A. F. *et al.* Anghabench: A suite with one million compilable c benchmarks for code-size reduction. *International Symposium on Code Generation and Optimization (CGO)* 378–390 (2021).

[22] Li, Y. *et al.* Competition-level code generation with alphacode. *Science* **378** (6624), 1092–1097 (2022).

[23] Tihanyi, N. *et al.* The formai dataset: Generative ai in software security through the lens of formal verification. *Proceedings of the 19th International Conference on Predictive Models and Data Analytics in Software Engineering* 33–43 (2023).

[24] Hafner, D., Pasukonis, J., Ba, J. & Lillicrap, T. Mastering diverse domains through world models. *arXiv preprint arXiv:2301.04104* (2023).

[25] Anand, A. *et al.* Procedural generalization by planning with self-supervised world models. *International Conference on Learning Representations (ICLR)* (2022).

[26] Abadi, M. *et al.* TensorFlow: a system for large-scale machine learning. *USENIX symposium on operating systems design and implementation (OSDI)* 265–283 (2016).

[27] Culjak, I., Abram, D., Pribanic, T., Dzapo, H. & Cifrek, M. A brief introduction to OpenCV. *Proceedings of the 35th international convention MIPRO* 1725–1730 (2012).

[28] Sutton, R. S. & Barto, A. G. *Reinforcement learning: An introduction* (MIT press, 2018).

[29] Cummins, C. *et al.* ProGraML: A graph-based program representation for data flow analysis and compiler optimizations. *International Conference on Machine Learning (ICML)* 2244–2253 (2021).

[30] Alon, U., Zilberstein, M., Levy, O. & Yahav, E. A general path-based representation for predicting program properties. *ACM SIGPLAN Notices* **53** (4), 404–419 (2018).

[31] Cummins, C., Petoumenos, P., Wang, Z. & Leather, H. End-to-end deep learning of optimization heuristics. *International Conference on Parallel Architectures and Compilation Techniques (PACT)* 219–232 (2017).

[32] Yang, X., Chen, Y., Eide, E. & Regehr, J. Finding and understanding bugs in c compilers. *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation* 283–294 (2011).

[33] Brauckmann, A., Goens, A., Ertel, S. & Castrillon, J. Compiler-based graph representations for deep learning models of code. *Proceedings of the 29th International Conference on Compiler Construction* 201–211 (2020).

[34] Mazoure, B., Ahmed, A. M., Hjelm, R. D., Kolobov, A. & MacAlpine, P. Cross-trajectory representation learning for zero-shot generalization in RL. *International Conference on Learning Representations (ICLR)* (2022).

[35] Williams, R. J. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning* **8**, 229–256 (1992).

[36] Greensmith, E., Bartlett, P. L. & Baxter, J. Variance reduction techniques for gradient estimates in reinforcement learning. *Journal of Machine Learning Research* **5** (9) (2004).

[37] Fursin, G., Cavazos, J., O'Boyle, M. & Temam, O. Midatasets: Creating the conditions for a more realistic evaluation of iterative optimization. *International conference on high-performance embedded architectures and compilers* 245–260 (2007).

[38] Hara, Y., Tomiyama, H., Honda, S., Takada, H. & Ishii, K. Chstone: A benchmark program suite for practical c-based high-level synthesis. *IEEE International Symposium on Circuits and Systems (ISCAS)* 1192–1195 (2008).

[39] Guthaus, M. R. *et al.* Mibench: A free, commercially representative embedded benchmark suite. *Proceedings of the fourth annual IEEE international workshop on workload characterization* 3–14 (2001).

[40] Bailey, D. *et al.* The NAS parallel benchmarks 2.0. Tech. Rep., Technical Report NAS-95-020, NASA Ames Research Center (1995).

[41] Lawson, C. L., Hanson, R. J., Kincaid, D. R. & Krogh, F. T. Basic linear algebra subprograms for fortran usage. *ACM Transactions on Mathematical Software (TOMS)* **5** (3), 308–323 (1979).

[42] Mnih, V. *et al.* Human-level control through deep reinforcement learning. *nature* **518** (7540), 529–533 (2015).

[43] Mnih, V. *et al.* Asynchronous methods for deep reinforcement learning. *International Conference on Machine Learning (ICML)* 1928–1937 (2016).

[44] Horgan, D. *et al.* Distributed prioritized experience replay. *International Conference on Learning Representations (ICLR)* (2018).

[45] Espeholt, L. *et al.* Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. *International Conference on Machine Learning (ICML)* 1407–1416 (2018).

[46] Schulman, J., Wolski, F., Dhariwal, P., Radford, A. & Klimov, O. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347* (2017).

[47] Jordan, M. I., Ghahramani, Z., Jaakkola, T. S. & Saul, L. K. An introduction to variational methods for graphical models. *Machine learning* **37**, 183–233 (1999).

[48] Kingma, D. P. & Welling, M. Auto-encoding variational bayes. *International Conference on Learning Representations (ICLR)* (2014).

[49] Lee, V., Abbeel, P. & Lee, Y. Dreamsmooth: Improving model-based reinforcement learning via reward smoothing. *International Conference on Learning Representations (ICLR)* (2024).

[50] Bellemare, M. G., Dabney, W. & Munos, R. A distributional perspective on reinforcement learning. *International Conference on Machine*

Learning (ICML) 449–458 (2017).

[51] Datta, K. *et al.* Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. *Proceedings of the 2008 ACM/IEEE conference on Supercomputing* 1–12 (2008).

[52] Stephenson, M., Amarasinghe, S., Martin, M. & O'Reilly, U.-M. Meta optimization: Improving compiler heuristics with machine learning. *ACM sigplan notices* **38** (5), 77–90 (2003).

[53] Luk, C.-K., Hong, S. & Kim, H. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture* 45–55 (2009).

[54] Kulkarni, S. & Cavazos, J. Mitigating the compiler optimization phase-ordering problem using machine learning. *Proceedings of the ACM international conference on Object oriented programming systems languages and applications* 147–162 (2012).

[55] Trofin, M. *et al.* Mlgo: a machine learning guided compiler optimizations framework. *arXiv preprint arXiv:2101.04808* (2021).

[56] Haj-Ali, A. *et al.* Neurovectorizer: End-to-end vectorization with deep reinforcement learning. *International Symposium on Code Generation and Optimization (CGO)* 242–255 (2020).

[57] Brauckmann, A., Goens, A. & Castrillón, J. Polygym: Polyhedral optimizations as an environment for reinforcement learning. *30th International Conference on Parallel Architectures and Compilation Techniques (PACT)* 17–29 (2021).

[58] Mirhoseini, A. *et al.* Device placement optimization with reinforcement learning. *International Conference on Machine Learning (ICML)* 2430–2439 (2017).

[59] Sutton, R. S. Dyna, an integrated architecture for learning, planning, and reacting. *ACM Sigart Bulletin* **2** (4), 160–163 (1991).

[60] Magni, A., Dubach, C. & O'Boyle, M. Automatic optimization of thread-coarsening for graphics processors. *Proceedings of the 23rd international conference on Parallel architectures and compilation* 455–466 (2014).

[61] Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S. & Dean, J. Distributed representations of words and phrases and their compositionality. *Advances in Neural Information Processing Systems (NeurIPS)* **26**

(2013).

[62] Alon, U., Zilberstein, M., Levy, O. & Yahav, E. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages* **3** (POPL), 1–29 (2019).

[63] Ben-Nun, T., Jakobovits, A. S. & Hoefler, T. Neural code comprehension: A learnable representation of code semantics. *Advances in Neural Information Processing Systems (NeurIPS)* **31** (2018).

[64] VenkataKeerthy, S. *et al.* Ir2vec: Llvm ir based scalable program embeddings. *ACM Transactions on Architecture and Code Optimization (TACO)* **17** (4), 1–27 (2020).

[65] Ansel, J. *et al.* Opentuner: An extensible framework for program autotuning. *Proceedings of the 23rd international conference on Parallel architectures and compilation* 303–316 (2014).

[66] Filho, J. F., Rodriguez, L. G. A. & da Silva, A. F. Yet another intelligent code-generating system: A flexible and low-cost solution. *Journal of Computer Science and Technology* **33**, 940–965 (2018).

[67] Liang, E. *et al.* RLlib: Abstractions for distributed reinforcement learning. *International Conference on Machine Learning (ICML)* (2018).

[68] Van der Maaten, L. & Hinton, G. Visualizing data using t-sne. *Journal of machine learning research* **9** (11) (2008).

Appendix A Related Work

In this section, we will elaborate a more extensive review of the development of machine learning methods for code optimization in compilers.

One of the key challenges for compilation is to determine which code transformations to apply, how to apply them (e.g., using suitable parameters), and in what order. This requires effectively searching and evaluating a massive number of possible options, known as iterative compilation [3] or autotuning [51]. However, this search-based approach only finds a good optimization for one specific program and does not generalize into a compiler heuristic. This limitation underscores the importance of integrating machine learning techniques.

Pioneering work has delved into supervised machine learning, adopting two main approaches. The first one learns a predictive model that can directly predict the best option. It involves iteratively compiling training programs to identify the most effective compilation strategy for each, which then serves as the labels of training data. An early example [4] used a neural network for branch prediction, and one more well-known work is Milepost-GCC [5], a practical attempt to integrate machine learning into a production compiler, GCC. It employs models trained on a large dataset of programs distributed over the Internet. The second approach aims to learn a cost or performance function capable of estimating the quality of various compiler options, which enables evaluation of a range of possible options without the need to compile and profile each one [52, 53].

Recent advancements have seen reinforcement learning (RL) techniques making strides in compiler optimization, circumventing the requirement for collecting optimal labeled data [54]. This technique has been applied to optimize individual compilation heuristics, such as inlining [55], loop transformation [56, 57], and graph partitioning [58]. Several works relevant to us have explored the full optimization pipeline, i.e. the LLVM phase ordering problem, including AutoPhase [15], CORL [17], and POSET-RL [18]. These methods predominantly utilize model-free RL algorithms while our work pioneers the use of an advanced model-based RL approach to reduce real compiler interactions. Model-based RL [59] learns a simulation model of the compiler environment, which is related in spirit to the aforementioned approach of supervised performance models. However, it goes a step further by also learning a policy capable of directly determining the best optimization option, thereby eliminating the necessity for a guided search process.

Both machine learning techniques require crafting high-quality features that capture the important characteristics of programs, a process known as feature engineering. The most prevalent feature vectors are based on the frequencies of various types of instructions within the programs [15, 60], designed by expert intuitions. Numerous studies have aimed to reduce the cost of feature design. Following the success of word2vec embeddings in natural language processing [61], methods like code2vec [62], inst2vec [63], and IR2vec [64] represent programs as distributed vectors that capture syntactic and semantic information from the abstract syntax tree (AST) or intermediate representation (IR). The surge in deep learning has enabled feeding raw information such as AST [30], control-data flow graphs (CFG) [29, 33], and code token sequences [31] into powerful deep neural networks, capable of learning useful representations end-to-end.

Platforms that expose the compiler as a playground for AI experiments have significantly reduced the entry barriers to intelligent compiler research. OpenTuner [65] and YaCoS [66] serve as autotuning frameworks with a range of compiler optimization techniques. Our experiments utilize CompilerGym [20], which offers user-friendly interfaces for researchers to interact with compilers in a reinforcement learning manner. We are optimistic that the future release of our trained code optimization agents, in conjunction with these platforms, can have a democratizing effect on applying AI techniques to compiler optimizations.

Appendix B Implementation Details

B.1 Compiler Environment

Our experiments are conducted on the CompilerGym platform [20], version 0.2.5, with LLVM-10.0.0 integration.

B.2 Features and Actions

We extract program features following the approach of Autophase [15]. As described in Section 5.1, we use a 56-dimension Autophase feature vector concatenated with a 42-dimension action histogram vector as the observation. The Autophase feature contains various statistics of the IR code, with each dimension thoroughly explained in Table B1. The action histogram vector contains the counts of actions already taken by the agent within the current episode. Both vectors are normalized to ensure that their values fall within a reasonable range. Specifically, each element of the Autophase vector is divided by the total instruction count of the program, whereas the action histogram vector is normalized by the total action count per episode, set as 45.

Our action space is derived from Autophase [15], comprising 45 LLVM optimization passes. However, CompilerGym excludes 3 actions due to updates in the latest LLVM version. Thus, we utilize a total of 42 actions. Table B2 presents the names of the LLVM optimization passes corresponding to these 42 actions.

B.3 Hyperparameters

The hyperparameters for our DreamerV3 agent implementation are outlined in Table B3. For hyperparameters not specified, we use the same value as the original DreamerV3 [24].

B.4 Model-Free Baselines

We use RLlib [67] to train and test model-free reinforcement learning algorithms including PPO [46], A2C [43], IMPALA [45], APEX [44], and DQN [42]. All the experiments are conducted with 10 CPUs and an RTX-3090 GPU over a minimum duration of 10 hours. The training process uses 5 rollout workers to interact with the environment and use 4 evaluation workers to evaluate the checkpoint on the validation set. We use default hyperparameters of algorithms in RLlib following the CompilerGym platform [20], except that we have carefully tuned the hyperparameters for our strongest baseline, PPO, as listed in Table B4.

Table B1: Descriptions of 56-dimension Autophase features (adapted from [20]).

Index	Name	Description
0	BBNumArgsHi	Number of BB where total args for phi nodes is gt 5
1	BBNumArgsLo	Number of BB where total args for phi nodes is [1, 5]
2	onePred	Number of basic blocks with 1 predecessor
3	onePredOneSuc	Number of basic blocks with 1 predecessor and 1 successor
4	onePredTwoSuc	Number of basic blocks with 1 predecessor and 2 successors
5	oneSuccessor	Number of basic blocks with 1 successor
6	twoPred	Number of basic blocks with 2 predecessors
7	twoPredOneSuc	Number of basic blocks with 2 predecessors and 1 successor
8	twoEach	Number of basic blocks with 2 predecessors and successors
9	twoSuccessor	Number of basic blocks with 2 successors
10	morePreds	Number of basic blocks with gt. 2 predecessors
11	BB03Phi	Number of basic blocks with Phi node count in range (0, 3]
12	BBHiPhi	Number of basic blocks with more than 3 Phi nodes
13	BBNoPhi	Number of basic blocks with no Phi nodes
14	BeginPhi	Number of Phi-nodes at beginning of BB
15	BranchCount	Number of branches
16	returnInt	Number of calls that return an int
17	CriticalCount	Number of critical edges
18	NumEdges	Number of edges
19	const32Bit	Number of occurrences of 32-bit integer constants
20	const64Bit	Number of occurrences of 64-bit integer constants
21	numConstZeroes	Number of occurrences of constant 0
22	numConstOnes	Number of occurrences of constant 1
23	UncondBranches	Number of unconditional branches
24	binaryConstArg	Binary operations with a constant operand
25	NumAShrInst	Number of AShr instructions
26	NumAddInst	Number of Add instructions
27	NumAllocaInst	Number of Alloca instructions
28	NumAndInst	Number of And instructions
29	BlockMid	Number of basic blocks with instructions between [15, 500]
30	BlockLow	Number of basic blocks with less than 15 instructions
31	NumBitCastInst	Number of BitCast instructions
32	NumBrInst	Number of Br instructions
33	NumCallInst	Number of Call instructions
34	NumGetElementPtrInst	Number of GetElementPtr instructions
35	NumICmpInst	Number of ICmp instructions
36	NumLShrInst	Number of LShr instructions
37	NumLoadInst	Number of Load instructions
38	NumMulInst	Number of Mul instructions
39	NumOrInst	Number of Or instructions
40	NumPHIInst	Number of PHI instructions
41	NumRetInst	Number of Ret instructions
42	NumSExtInst	Number of SExt instructions
43	NumSelectInst	Number of Select instructions
44	NumShlInst	Number of Shl instructions
45	NumStoreInst	Number of Store instructions
46	NumSubInst	Number of Sub instructions
47	NumTruncInst	Number of Trunc instructions
48	NumXorInst	Number of Xor instructions
49	NumZExtInst	Number of ZExt instructions
50	TotalBlocks	Number of basic blocks
51	TotalInsts	Number of instructions (of all types)
52	TotalMemInst	Number of memory instructions
53	TotalFuncs	Number of non-external functions
54	ArgsPhi	Total arguments to Phi nodes
55	testUnary	Number of Unary operations

Table B2: A list of LLVM transformation passes selected as actions.

Index	**Name**	**Index**	**Name**	**Index**	**Name**
0	-adce	14	-instcombine	28	-lowerinvoke
1	-break-crit-edges	15	-ipsccp	29	-lowerswitch
2	-constmerge	16	-jump-threading	30	-mem2reg
3	-correlated-propagation	17	-lcssa	31	-memcpyopt
4	-deadargelim	18	-licm	32	-partial-inliner
5	-dse	19	-loop-deletion	33	-prune-eh
6	-early-cse	20	-loop-idiom	34	-reassociate
7	-functionattrs	21	-loop-reduce	35	-sccp
8	-functionattrs	22	-loop-rotate	36	-simplifycfg
9	-globaldce	23	-loop-simplify	37	-sink
10	-globalopt	24	-loop-unroll	38	-sroa
11	-gvn	25	-loop-unswitch	39	-strip
12	-indvars	26	-lower-expect	40	-strip-nondebug
13	-inline	27	-loweratomic	41	-tailcallelim

Table B3: Hyperparameters of DreamerV3 in our experiments.

	Hyperparameter	Value
Architecture	RSSM recurrent units	1024
	RSSM number of latents	32
	RSSM classes per latent	32
	MLP layers	4
	MLP hidden units	400
	Activation	LayerNorm + SiLU
Training	Random exploration	500 environment steps
	Replay buffer capacity	2×10^6
	Reward smoothing α [49]	0.6
	Training frequency	Every 5 environment steps
	Batch size	50
	Batch length T	50
	Imagination horizon H	15
	Discount γ	0.99
	λ-target discount	0.95
	World model loss scales	100.0 for Autophase
		10.0 for action histogram
		1.0 for reward
		5.0 for discount
		0.1 for KL
	Actor entropy regularization η	3×10^{-4}
	KL balancing	0.8
	Optimizer	Adam
	World model learning rate	1×10^{-4}
	Actor-critic learning rate	3×10^{-5}
	Weight decay	1×10^{-5}
	Gradient clipping	100

Table B4: Hyperparameters for the PPO baseline, well tuned on our dataset to be deviating from the default value in RLlib.

	Hyperparameters	Value
PPO	`gamma`	1.0
	`use_gae`	True
	`lambda_`	1.0
	`train_batch_size`	9000
	`lr`	5e-5
	`kl_coeff`	0.2
	`kl_target`	0.01
	`vf_loss_coeff`	1.0
	`num_sgd_iter`	30
	`sgd_minibatch_size`	128
	`clip_param`	0.3
	`vf_clip_param`	10.0
	`weight_decay`	1e-6

Table C5: Dataset division of 8 CompilerGym benchmarks.

Dataset	Training Split	Validation Split	Test Split
BLAS	200	50	50
cBench	23	N/A	N/A
CHStone	12	N/A	N/A
Linux	13,794	50	50
MiBench	40	N/A	N/A
NPB	22	50	50
OpenCV	342	50	50
TensorFlow	1,885	50	50

Appendix C Benchmarks

CompilerGym Benchmarks

In our study, we select eight benchmarks for zero-shot test and in-domain training: BLAS, cBench, CHStone, Linux, MiBench, NPB, OpenCV, and TensorFlow. These benchmarks are part of the built-in datasets provided by CompilerGym version 0.2.5. For benchmarks with a total number of programs more than 100, we use the first 50 programs as the test set, the following 50 programs as the validation set, and all of the rest as the training set. These training and validation sets are only used for in-domain training. The datasets comprising fewer than 100 programs are not applicable for in-domain training; instead, all their programs are allocated to the test set. The number of programs in each dataset after division is detailed in Table C5.

AI-Generated Benchmarks

To further test the generalization ability of our *CodeZero* agent on different programming languages, we borrow the method from FormAI [23] and generate a dataset containing 50 unique Objective-C programs using GPT-3.5. We use the same prompt as FormAI, except that we add an instruction to ask GPT to generate programs that can be directly compiled under Clang version 10.0.0 and do not use ARC (Automatic Reference Counting), to improve the compilation pass rate of generated programs. We compile the generated programs using Clang without including any third-party libraries, and all programs that cannot pass compilation are discarded.

Appendix D Extended Experimental Results

D.1 Learning Curves

The zero-shot test performance of our CodeZero agents during training is shown in Figure D1. Note that we validate and test the agent every 10000 environment steps and report the test performance from the checkpoint that achieved the best validation results for comparison among various methods.

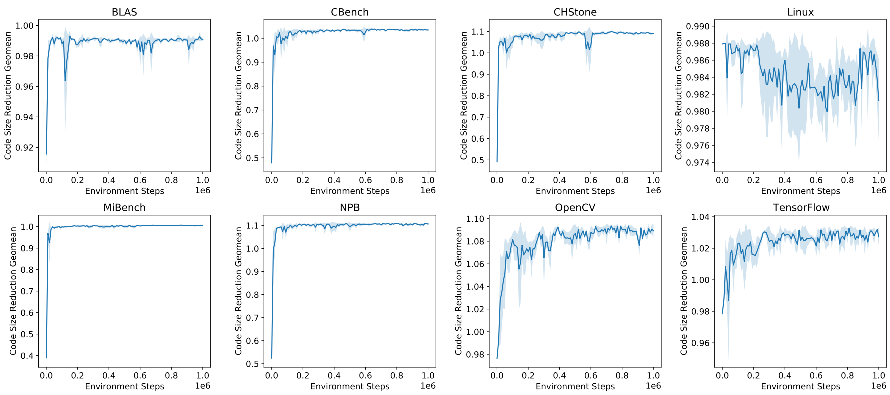

Fig. D1: Zero-shot test performance of the CodeZero agent during training. We report mean and standard deviation across three runs.

D.2 Quantitative Results

Quantitative results corresponding to Figure 3 in the main text are provided in Table D6.

Table D6: Quantitative results for code size reduction, corresponding to Figure 3. We report mean and standard deviation across three runs.

		-O0	Random Policy	Random Search (1s)	CodeZero (In-domain)	CodeZero (Zero-shot)
BLAS	geomean	0.931	0.821±0.018	0.960±0.004	0.993±0.005	**0.991**±0.000
	min	0.707	0.432±0.106	0.723±0.076	0.831±0.148	0.913±0.000
	max	1.000	1.002±0.004	1.031±0.010	1.025±0.005	1.016±0.000
cBench	geomean	0.481	0.737±0.015	0.858±0.020	N/A	**1.036**±0.002
	min	0.299	0.401±0.097	0.560±0.030	N/A	0.962±0.001
	max	0.626	1.150±0.113	1.298±0.083	N/A	1.395±0.054
CHStone	geomean	0.487	0.701±0.043	1.037±0.006	N/A	**1.094**±0.005
	min	0.402	0.278±0.108	0.889±0.044	N/A	1.000±0.006
	max	0.655	1.038±0.033	1.304±0.038	N/A	1.378±0.021
Linux	geomean	0.988	0.981±0.004	1.001±0.000	**0.993**±0.000	0.986±0.005
	min	0.615	0.740±0.126	1.000±0.000	0.700±0.001	0.642±0.041
	max	1.011	1.001±0.002	1.011±0.000	1.011±0.000	1.009±0.003
Mibench	geomean	0.389	0.812±0.022	1.005±0.001	N/A	**1.006**±0.002
	min	0.278	0.352±0.074	0.858±0.010	N/A	0.879±0.003
	max	0.760	1.392±0.074	1.603±0.015	N/A	1.588±0.000
NPB	geomean	0.530	0.833±0.011	1.074±0.010	1.075±0.017	**1.108**±0.001
	min	0.191	0.414±0.043	0.805±0.104	0.810±0.082	0.886±0.000
	max	1.066	1.848±0.193	2.141±0.062	2.315±0.028	2.343±0.012
OpenCV	geomean	0.981	0.949±0.011	1.080±0.001	1.087±0.007	**1.092**±0.000
	min	0.833	0.603±0.133	0.888±0.021	0.898±0.001	0.897±0.000
	max	1.370	1.409±0.167	1.571±0.022	1.635±0.186	1.556±0.000
TensorFlow	geomean	0.983	0.912±0.009	1.006±0.003	**1.032**±0.000	**1.032**±0.001
	min	0.927	0.618±0.046	0.877±0.009	0.970±0.010	0.968±0.011
	max	1.010	1.235±0.034	1.267±0.007	1.289±0.008	1.282±0.002

D.3 Data Distribution Visualization

In Figure D2, we visualize the distribution of our training and test datasets. To accurately represent the dynamic behavior of programs, we randomly select 1000 action sequences, each with a length of 45, from our action space. These sequences are subsequently executed on each program, with the resulting Autophase features concatenated to form a feature vector with dimensions of $1000 \times 45 \times 56$ for every program. These comprehensive feature vectors are finally processed using t-SNE [68] for dimensionality reduction and visualization.

Figure D2 illustrates that our training data (denoted as triangles) has a broad coverage of test programs (denoted as circles). This contrasts with the Csmith dataset (denoted as crosses) employed in CompilerGym experiments [20], which shows a significant deviation from real-world applications. Nonetheless, our visualization can still not perfectly capture the transferability across datasets. For instance, empirical evidence suggests that CodeContests are the most effective in generalizing to OpenCV and TensorFlow, while the visual analysis does not directly imply this.

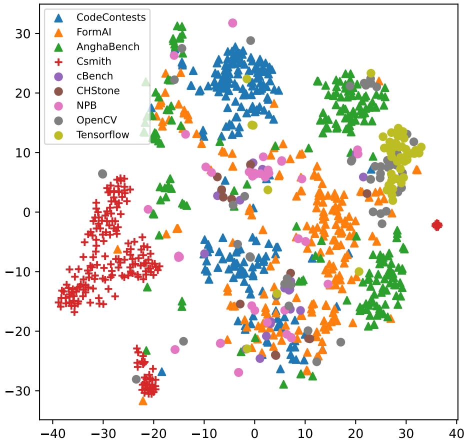

Fig. D2: t-SNE [68] visualization of programs from training and test datasets.