CompilerDream: Learning a Compiler World Model for General Code Optimization

Chaoyi Deng*
School of Software, BNRist
Tsinghua University
Beijing, China
dengcy23@mails.tsinghua.edu.cn

Jialong Wu*
School of Software, BNRist
Tsinghua University
Beijing, China
wujialong0229@gmail.com

Ningya Feng
School of Software, BNRist
Tsinghua University
Beijing, China
fny21@mails.tsinghua.edu.cn

Jianmin Wang
School of Software, BNRist
Tsinghua University
Beijing, China
jimwang@tsinghua.edu.cn

Mingsheng Long†
School of Software, BNRist
Tsinghua University
Beijing, China
mingsheng@tsinghua.edu.cn

Abstract

Effective code optimization in compilers is crucial for computer and software engineering. The success of these optimizations primarily depends on the selection and ordering of the optimization passes applied to the code. While most compilers rely on fixed pass sequences, current methods to find the optimal sequence for specific programs either employ impractically slow search algorithms or learning methods that struggle to generalize to code unseen during training. To address these challenges, we introduce CompilerDream, the first world-model-based approach for general code optimization. CompilerDream features a compiler world model with a reward smoothing technique, enabling accurate simulation of optimization processes. Built on this model, code optimization agents can then be constructed via value prediction or direct optimization sequence generation. Trained on a large-scale program dataset, these agents serve as versatile code optimizers across diverse application scenarios and source-code languages. Our extensive experiments highlight CompilerDream's strong optimization capabilities for autotuning, where it leads the CompilerGym leaderboard. More importantly, the zero-shot generalization ability of large-scale trained compiler world model and agent, excels across diverse datasets, surpassing LLVM's built-in optimizations and state-of-the-art methods in both settings of value prediction and end-to-end code optimization.

CCS Concepts

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

Keywords

World Models, Compiler Optimization, Code Data Mining

*Both authors contributed equally to this research.
†Corresponding author.

KDD '25, August 3–7, 2025, Toronto, ON, Canada

ACM ISBN 979-8-4007-1454-2/2025/08
https://doi.org/10.1145/3711896.3736887

ACM Reference Format:
Chaoyi Deng, Jialong Wu, Ningya Feng, Jianmin Wang, and Mingsheng Long. 2025. CompilerDream: Learning a Compiler World Model for General Code Optimization. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.2 (KDD '25), August 3–7, 2025, Toronto, ON, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3711896.3736887

KDD Availability Link:
The source code of this paper has been made publicly available at https://doi.org/10.5281/zenodo.15552746. The data of this paper has been made publicly available at https://doi.org/10.5281/zenodo.15549673.

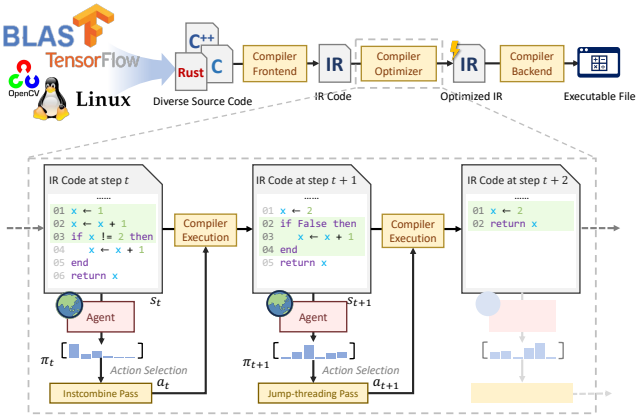

Figure 1: CompilerDream performs code optimization by interacting with the compiler using an agent powered by a world model. With strong generalization capabilities, it efficiently optimizes programs from diverse origins.

1 Introduction

Code optimization plays an important role in realizing the full potential of software and hardware. Developers desire a universal solution to transform input programs into semantically equivalent but more efficient versions without manual effort. Compilers achieve this through a front-end that translates source code into

an intermediate representation (IR), a middle-end optimizer that performs language- and platform-agnostic IR optimizations, and a back-end that converts IR to the binary code (Figure 1). The optimizer is typically implemented as a series of *passes* applying transforms on the code, where performance largely depends on the selection and order of these optimization passes. Standard compilers use a few fixed sets of optimization sequences to enhance specific aspects of program performance, such as `-O1`, `-O2`, and `-O3` for execution speed, and `-Os` and `-Oz` for program size reduction.

Obviously, given the vast diversity of programs and platforms, these off-the-shelf strategies predefined by compiler experts are suboptimal for most circumstances. Automatically optimizing the pass sequence for specific programs thus can yield significant performance gains over default compiler settings [19, 56]. To be practical, such an algorithm must produce a satisfactory pass sequence within a reasonable time and handle a wide variety of programs. However, current research often fails to meet these requirements simultaneously. Search-based methods [5] achieve near-optimality but require thousands of compilations per program to validate the optimization outcomes, making them impractical. In contrast, machine learning methods avoid these time-consuming compiler interactions, by either predicting the optimization sequences directly or estimating the outcomes of optimization sequences to guide a search.

However, these learning-based methods still risk sacrificing some optimality and, more importantly, face a significant bottleneck in broad generalization across diverse programs that may be out of the training samples. A range of machine learning applications [7, 14, 32, 49] have witnessed that training high-capacity models on large-scale datasets can yield unprecedented performance. Prior studies on compiler optimization also suggest that training on large datasets with diverse programs could be beneficial [11], yet the prevalent practice in this field is still to learn optimization strategies in a per-program manner [27, 52] or from relatively small training sets comprising only a few hundred programs with limited-capacity models [30, 44], which hinders generalization.

This paper focuses on the LLVM [34] phase ordering problem, a longstanding challenge for compiler research. We propose CompilerDream, a world-model-based approach for general code optimization, capable of handling a wide variety of programs, unlike most prior approaches focusing on narrow, domain-specific ones. It learns an accurate predictive world model to simulate compiler executions, forecasting future IR states and optimization metric improvements based on the current IR state and applied pass. We believe that employing a world model offers the following advantages for compiler optimization: First, by capturing optimization dynamics, the world model gains generalizable knowledge into our method [2], improving the overall generalization performance. Second, it replaces costly compiler invocations, significantly reducing computational overheads of search and learning algorithms and facilitating large-scale training. Additionally, its high-capacity architecture can extract deeper insights from extensive datasets.

Specifically, to better adapt the world model for compiler optimization, we closely examine the optimization process and introduce a reward smoothing technique to enhance world model training and improve simulation accuracy. Built on this accurate world model, CompilerDream supports various types of optimization agents, including value prediction and reinforcement learning. Moreover, we carefully curate a large-scale training dataset of natural programs, enhancing the world model's and agent's generalization to unseen programs, enabling our method to predict superior optimization sequences.

We demonstrate CompilerDream's effectiveness across a range of program domains [11] and problem scenarios. Our test domains include benchmark suites covering fundamental algorithms, as well as production-level open-source programs, such as object files from C++ TensorFlow [1] and OpenCV [9] libraries. Even without considering generalization, CompilerDream's strong optimization capabilities top the CompilerGym leaderboard for autotuning. For general code optimization, our large-scale trained world model accurately predicts the outcomes of pass sequences on unseen programs. Both types of optimization agents equipped with the world model outperform the built-in `-Oz` flag and state-of-the-art methods in their respective settings. In the value prediction setting, the world model serves as an accurate value predictor, enabling superior action sequence selection. In the reinforcement learning setting, the agent trained entirely in the world model directly generates optimization sequences in a single trial, achieving more efficient code across diverse datasets.

The main contributions of this work are as follows:

- We propose *CompilerDream*, the first world-model-based approach for code optimization.
- We introduce a *reward smoothing technique* that facilitates the application of world models to compiler optimization, benefiting future research in this area.
- Our approach supports *multiple optimization agents*, including value prediction and reinforcement learning agents, unifying different methods in this field.
- By leveraging the *large-scale CodeContests dataset* and the world model's generalization capability, we achieve strong performance on diverse unseen programs.
- Extensive experiments across various program domains and problem scenarios, including autotuning, value prediction, and end-to-end code optimization, show that CompilerDream outperforms state-of-the-art methods.

2 Related Work

A key challenge in compilation is determining which code transformations to apply, how to apply them (e.g., using suitable parameters), and in what order. This involves effectively searching and evaluating numerous options, a process known as iterative compilation [5] or autotuning [13]. However, this search-based approach only finds a good optimization for one specific program and does not generalize into a compiler strategy. This limitation underscores the importance of integrating machine learning techniques.

Supervised learning. Pioneering work has delved into supervised machine learning, adopting two main approaches [36]. The first approach requires an extensive search for each training program to identify the most effective optimization sequence, which then serves as the data labels. An early example [8] used a neural network for branch prediction, and one more well-known work is MilepostGCC [18], 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 the evaluation of possible options without the need to compile and profile each one [42, 53]. Coreset-NVP [41] follows this approach and achieves state-of-the-art performance.

Reinforcement learning. Recent advancements have seen reinforcement learning (RL) techniques making strides in compiler optimization, circumventing the need for collecting optimal labeled data [33]. This technique has been applied to optimize individual compilation heuristics, such as inlining [57], loop transformation [6, 26], and graph partitioning [46]. Several works relevant to us, including AutoPhase [27], CORL [44], and POSET-RL [30], have explored the full optimization pipeline, i.e., the phase ordering problem, using model-free RL without incorporating a world model.

World model. A world model [21, 37] that approximates state transitions and reward signals is typically used in two ways: (1) it enables *simulation* of "unseen" interactions that are unavailable or unaffordable [23, 31]; (2) as an auxiliary learning task, it aids in better *representation* that captures the underlying structure of the environment [2, 43, 45]. To the best of our knowledge, we are the first to introduce world models for code optimization. In this context, a world model can function as a compiler simulator, approximating IR transformations and eliminating the need for costly execution and profiling of extensive optimization sequences. Furthermore, by sharing representations with the world model, the policy can generalize more effectively to unseen programs. Thus, model-based agents [54] can be implemented, which offer superior sample efficiency and generalization compared to model-free methods.

3 Method

This section first defines the code optimization problem and design choices for observation, action, and reward (Section 3.1). We then present the design and training technique of CompilerDream's world model (Section 3.2), two optimization agents (Section 3.3), and considerations for large-scale dataset curation (Section 3.4).

3.1 Phase Ordering Decision Process

As illustrated in Figure 1, one key problem of compiler optimization is to find the optimal sequence of optimization passes for a given program, also known as the *phase ordering* problem. It can be naturally formulated as a *partially observable Markov decision process* (POMDP) $\mathcal{M} = (\mathcal{S}, \mathcal{A}, r, p, \mu, \mathcal{O}, \phi)$ [54]. 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, capturing useful features.

A *code optimization agent* determines the optimization sequence for an input program through interactions with the compiler environment. At each time step $t = 0, 1, 2, \ldots$, the agent applies an action a_t to current IR based on observation o_t, transforming current state to $s_{t+1} = p(s_t, a_t)$ and receiving reward r_t. The agent can employ effective strategies, such as search algorithms and parametric neural networks, to select the best actions, as detailed in Section 3.3.

Table 1: Observation and action space in CompilerDream.

Aspect	Name	Description	Dim.
Observation	Autophase [27]	A feature vector capturing various IR code statistics.	56
	Action Histogram	A vector where each dimension represents the execution frequency of a specific action.	124 /42
Action (Sec. 4.2 & 4.3)	Full LLVM Passes	Full action space with all LLVM optimization passes.	124
Action (Sec. 4.4)	Autophase Passes	A reduced action space derived from Autophase[27].	42

Under the POMDP formulation, we design the *observation* and *action space* as summarized in Table 1. The observation features were selected for efficiency and effectiveness. In preliminary experiments, we find that complex program features like ProGraML [10] and inst2vec [4] significantly slow down CompilerDream with marginal performance gains. In contrast, expert-designed Autophase features [27] leverage domain knowledge, enhancing generalization by filtering irrelevant details. We construct the observation by concatenating the 56-dimensional Autophase feature vector with a 42-dimensional action histogram vector, which records the number of times each action has been selected during the current episode. Both vectors are normalized for consistency: each Autophase feature is divided by the program's initial total instruction count, and the action histogram is scaled by the per-episode action limit, which is set to 45. We adopt two distinct action spaces in our experiments to align with the baseline methods. The full action space consists of all 124 LLVM optimization passes, while the reduced action space derived from Autophase [27] consists of 42 actions. Originally, this action space included 45 LLVM passes, but CompilerGym excludes 3 due to updates in the latest LLVM version, leaving 42 actions. This reduced action space is widely used and has been shown to be effective in prior studies [11, 27].

The *reward function* is defined as the normalized change of the optimization metric $C(s)$:

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

where lower C indicates better performance. Following prior work on code size reduction, we define $C(s)$ as the IR instruction count. $C(s_b)$ is the baseline performance achieved by the built-in `-Oz` flag.

3.2 Learning a Compiler World Model

Following the advanced Dreamer approach [25], we build a world model to learn the formulated POMDP process of compiler optimization, as depicted in Figure 2. Concretely, we train a model

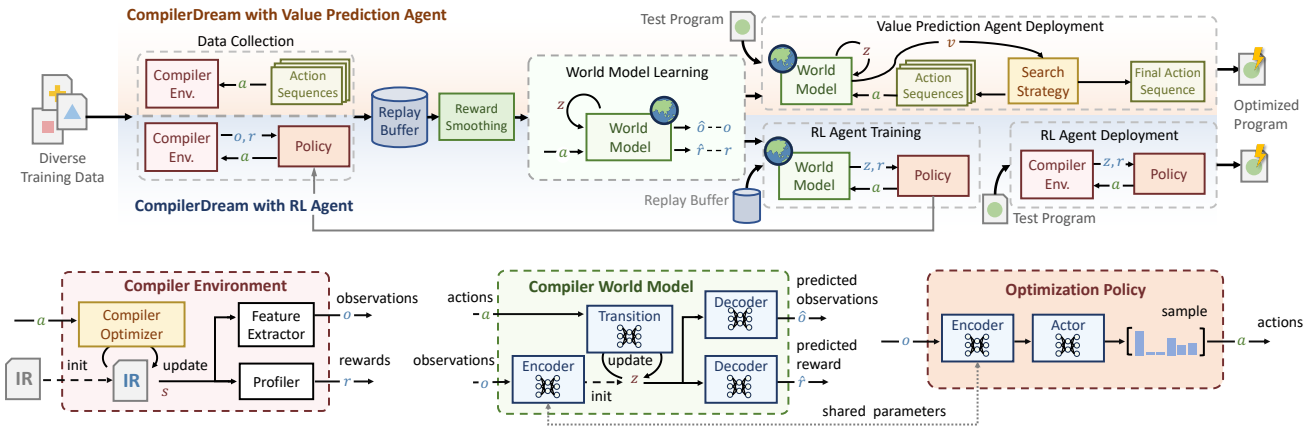

Figure 2: Design overview of CompilerDream. The world model is trained on real episodes generated by the agent, incorporating a reward smoothing process to stabilize training and enhance learning efficiency. CompilerDream supports training with RL and value prediction agents, which learn from episodes simulated by the world model.

$(\hat{p}_\theta, \hat{r}_\theta)$ of the compiler environment parameterized by θ, which approximates the underlying transition dynamics of optimization process $p(o_{t+1}|o_{\le t}, a_{\le t})$ and the reward function $r(o_{\le t}, a_{\le t})$.

The compiler world model simulating the compiler environment is formulated with the following four components:

$$\begin{aligned} &\text{Representation model:} && z_t \sim q_\theta(z_t \mid z_{t-1}, a_{t-1}, o_t), \\ &\text{Transition model:} && \hat{z}_t \sim p_\theta(\hat{z}_t \mid z_{t-1}, a_{t-1}), \\ &\text{Observation decoder:} && \hat{o}_t \sim p_\theta(\hat{o}_t \mid z_t), \\ &\text{Reward decoder:} && \hat{r}_t \sim p_\theta(\hat{r}_t \mid z_t). \end{aligned} \tag{2}$$

The representation model estimates a *neural compiler state* z_t from the current observation o_t of *real compiler state*, the previous state z_{t-1} and the previous optimization action a_{t-1}. A representation loss $\mathcal{L}_{\text{repr}}$ is used to train the neural compiler states to accurately reconstruct the observation and reward by two decoders:

$$\mathcal{L}_{\text{repr}}(\theta) \doteq -\ln p_\theta(o_t|z_t) - \ln p_\theta(r_t|z_t). \tag{3}$$

The transition model captures the dynamics of the compiler world model, predicting future neural compiler state $\hat{z}_t$ directly from z_{t-1} and a_{t-1}. A prediction loss minimizes the difference between the estimated neural compiler state z_t and the predicted compiler state $\hat{z}_t$, simultaneously enhancing the transition model's accuracy in predicting future states and making the neural compiler state produced by the representation model easier to predict:

$$\mathcal{L}_{\text{pred}}(\theta) \doteq \text{KL}\left[q_\theta(z_t|z_{t-1}, a_{t-1}, o_t) \,\|\, p_\theta(\hat{z}_t|z_{t-1}, a_{t-1})\right]. \tag{4}$$

The overall models are jointly learned by minimizing the sum of the representation loss and the prediction loss.

Compiler simulation. We can simulate the behavior of the real compiler using our trained compiler world model. Specifically, a *simulated* compiler optimization trajectory $\{\hat{z}_\tau, \hat{a}_\tau, \hat{r}_\tau, \hat{o}_\tau\}$ with horizon H can be generated by the interactions between the world model and an optimization agent: starting at a neural state $\hat{z}_t \sim q_\theta(z_t|o_t)$, at each step $\tau = t, t+1, t+2, \ldots$, the agent takes an action $\hat{a}_\tau \sim \pi_\psi(\hat{a}_\tau|\hat{z}_\tau)$, and transits to the next latent state $\hat{z}_{\tau+1} \sim p_\theta(\hat{z}_{\tau+1}|z_\tau, a_\tau)$ with a reward $\hat{r}_{\tau+1} \sim p_\theta(\hat{r}_{\tau+1}|\hat{z}_{\tau+1})$. Predicted real compiler state $\hat{o}_{\tau+1}$ can be optionally reconstructed by the observation decoder.

Reward smoothing. We discover that most optimization passes in an optimization sequence do not change the program IR instruction count, leading to *sparse* rewards. Despite this, these passes are essential as they may modify critical properties, such as instruction order or replacements, that affect the effectiveness of subsequent passes. Furthermore, improvements tend to *saturate* over time, as large non-zero rewards typically appear early in the sequence and diminish as optimization progresses.

In Figure 3, we present a typical episode of the optimization process: out of 45 actions, only 8 yielded non-zero rewards, whose values decrease as optimization progresses. These properties pose challenges for training the agent effectively, as it receives little to no guidance from most actions, even when some are vital to the final result. Moreover, the world model may struggle to predict reward values accurately, often defaulting to zero. This challenge

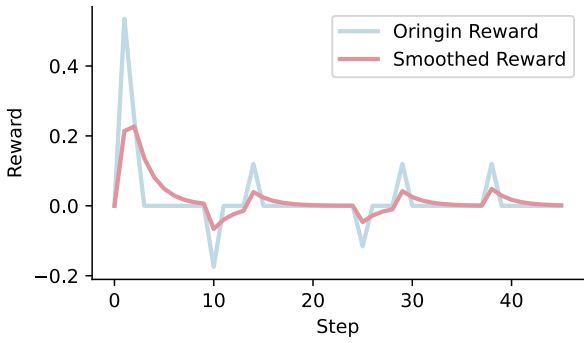

Figure 3: Comparison of the original and smoothed rewards during an episode on a program from the BLAS dataset [35].

arises from the significant class imbalance between zero and non-zero rewards, making it difficult to determine both the timing and magnitude of non-zero rewards.

Therefore, we applied reward smoothing to mitigate their sparsity and long-tailed distribution within an episode. This is achieved by adding an exponential decay to rewards:

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

with $\alpha \in [0, 1)$. Consequently, we train a reward decoder $p_\theta(\hat{r}'_t \mid z_t)$ to predict the smoothed rewards. As shown in Figure 3, the smoothed rewards provide non-zero feedback for most steps, making them easier for CompilerDream to learn. It is worth noting that the total reward of an episode remains approximately unchanged after smoothing. For all $t \geq 1$, its contribution to the total reward after smoothing approximates the original reward value if the horizon is infinite: $\lim_{n\to\infty}(1-\alpha)\sum_{i=1}^{n}\alpha^i r_t = r_t$. Since most episodes exhibit a long-tailed reward distribution and we use a relatively small smoothing factor ($\alpha = 0.6$), the smoothed total reward is expected to closely match the original total reward.

3.3 Learning a Code Optimization Agent

Building on the world model, we implement two agent designs, unifying supervised and reinforcement learning methods for compiler optimization into our world-model-based approaches.

Value prediction agent. This kind of agent adopts a classic approach in compiler optimization, employing a heuristic to guide search methods [41, 42, 53]. It comprises a *value prediction model* v_θ to estimate the effect of an optimization sequence and a *search strategy* that leverages this model to identify the best sequence.

The *value* of a program state s is defined as the expected cumulative reward of applying an action sequence to the program. This aligns with the ratio of the reduced IR instruction count achieved between applying action sequence $\{a_\tau\}_{\tau=1}^{m}$ and standard $-Oz$ flag, according to the reward function r defined in Eq. (1):

$$v(s_0, \{a_\tau\}) = \sum_{\tau=1}^{m} r_\tau = \sum_{\tau=1}^{m} \frac{C(s_{\tau-1}) - C(s_\tau)}{C(s_0) - C(s_b)} = \frac{C(s_0) - C(s_m)}{C(s_0) - C(s_b)}. \tag{6}$$

We leverage our world model to build a *value prediction model*. As shown Section 3.2, from an initial observation o_0, and an action sequence $\{a_\tau\}$, our world model can simulate an optimization trajectory $\{\hat{z}_\tau, \hat{a}_\tau, \hat{r}_\tau, \hat{o}_\tau\}_{\tau=1}^{m}$. Summing up all simulated rewards $\hat{r}_\tau$ yields the predicted value:

$$v_\theta(s_0, \{a_\tau\}) = \sum_{\tau=1}^{m} \hat{r}_\tau \approx \sum_{\tau=1}^{m} r_\tau \tag{7}$$

The search strategy then seeks the optimal optimization sequence $\{a_\tau\}^*$ by evaluating candidates using the value prediction model v_θ. Although various search algorithms are applicable, we employ a simple strategy that enumerates action sequences within a fixed search space (detailed in Section 4.3).

Reinforcement learning agent. RL is another popular approach to compiler optimization, learning a policy to select passes sequentially. Unlike prior methods that rely on costly compiler interactions, CompilerDream trains RL agents on world model-simulated compiler optimization trajectories, significantly improving efficiency.

Table 2: Comparison of different training datasets for compiler optimization.

Dataset Name	Number of IR files	Large-scale	Human-written	Code quality
cBench [17]	23	No	Yes	High
Mibench [20]	40	No	Yes	Low
Csmith [59]	∞	Yes	No	High
llvm-stress [34]	∞	Yes	No	Low
AnghaBench [12]	1,041,333	Yes	Yes	Low
CodeContests [39]	110,240	**Yes**	**Yes**	**High**

Our RL agent comprises an actor and a critic neural networks, both parameterized based on the neural compiler state:

$$\begin{aligned} &\text{Actor: } \hat{a}_t \sim \pi_\psi(\hat{a}_t \mid \hat{z}_t) \\ &\text{Critic: } v_\xi(\hat{z}_t) \approx \mathbb{E}_{p_\theta, \pi_\psi}\left[\sum_{\tau \geq t} \gamma^{\tau-t} \hat{r}_\tau\right]. \end{aligned} \tag{8}$$

The critic evaluates the γ-discounted value $v_\xi(\hat{z}_t)$ of simulated neural state $\hat{z}_t$ under policy π_ψ. It is trained by minimizing the difference between the predicted value $v_\xi(\hat{z}_t)$ and the Monte-Carlo or more advanced bootstrapped return [54], denoted as V_t:

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

The actor produces an optimization policy π_ψ that predicts an action distribution of the best pass to choose, which is trained to maximize the simulated return through the REINFORCE policy gradient [58] with an entropy regularization [22]:

$$\begin{aligned} \mathcal{L}_{\text{actor}}(\psi) \doteq \mathbb{E}_{p_\theta, \pi_\psi}\Bigg[\sum_{\tau=t}^{t+H} \Bigg(&-\Big(V_\tau - v_\xi(\hat{z}_\tau)\Big) \log \pi_\psi(\hat{a}_\tau \mid \hat{z}_\tau) \\ &- \eta\, \mathbb{H}\left[\pi_\psi(\hat{z}_\tau)\right] \Bigg)\Bigg], \end{aligned} \tag{10}$$

3.4 Data Curation

To facilitate that our *CompilerDream* method can effectively generalize to unseen situations, a concept known as zero-shot generalization, we have identified three critical factors in preparing our training dataset. First, the dataset should reflect *naturalness*. During our preliminary experiments, we found that those generated code datasets like Csmith [59] and llvm-stress [34] provide no benefits or even hurt the generalization to real-world scenarios. Second, the dataset should be *large* to prevent the agent from overfitting to a small number of codes and failing to generalize. Last, the code must exhibit *high quality* from an optimization perspective. We need data with complex algorithmic logic and potential for optimizations to help the world model and the agent better understand the problem's intricacies. Datasets like AnghaBench [12] which consist of millions of human-written codes collected from GitHub, are often too simple to allow for significant optimization improvements.

Therefore, we choose to construct our training datasets on top of the CodeContests dataset released with AlphaCode [39], which consists of over 13,000 problems of coding competition, and each problem, on average, has hundreds of solutions in multiple languages. We subsample up to ten C++ solutions for each training

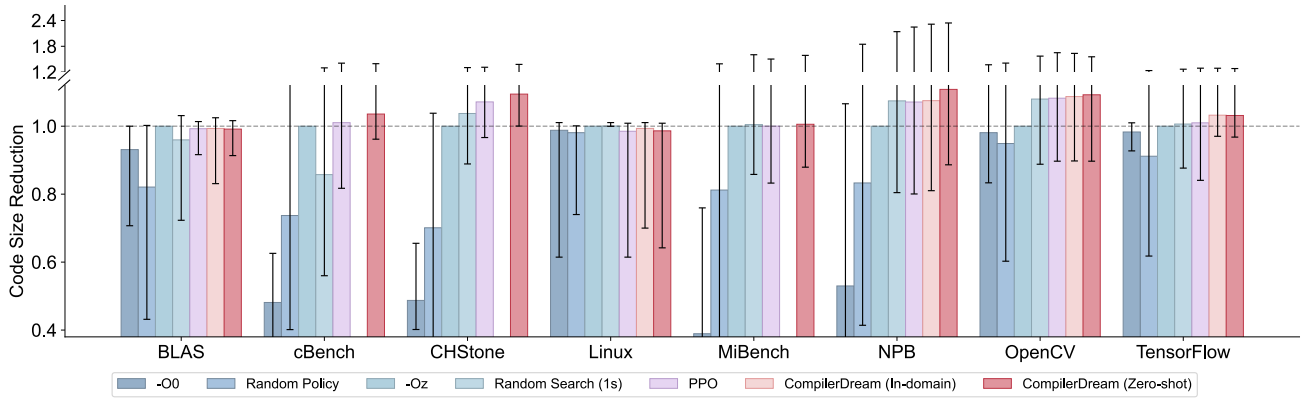

Figure 4: Results of general code optimization with RL: Code size reduction in terms of IR instruction count over LLVM -Oz under different methods. Bars indicate the geometric mean and min-max range across test programs in each benchmark dataset.

problem, resulting in 110,240 programs, as our training data, and sample one solution for each of 100 test problems as our validation data. As shown in Table 2, CodeContests meets all our criteria, whereas other datasets fall short in one or more aspects.

4 Experiments

We conducted extensive experiments across various settings, comparing CompilerDream against state-of-the-art methods. Our study aims to answer the following key research questions (RQs):

RQ 1 *Optimality*: Can CompilerDream's joint training process of the compiler world model and optimization agent discover superior optimization sequences?

RQ 2 *Accuracy*: Does CompilerDream's world model produce accurate simulations of the real optimization process?

RQ 3 *Generalization*: Is the policy learned with the world model still effective on various unseen programs?

RQ 4 *Effectiveness*: Are all the techniques employed in CompilerDream effective in enhancing optimization results?

In the following sections, we first show that CompilerDream excels as a powerful autotuning method, leading the CompilerGym leaderboard by discovering superior optimization sequences and leveraging accurate world model (Section 4.2), addressing RQ 1 and partially RQ 2. We then demonstrate that its value prediction agent (Section 4.3) and RL agent (Section 4.4) both outperform state-of-the-art methods in various unseen test datasets, addressing RQ 2 and RQ 3. Finally, Section 4.5 presents ablation studies for RQ 4.

4.1 Evaluation

Our experiments focus on code size reduction, benefiting applications on low-resource hardware like embedded systems. This focus stems from the practical advantages of code size as a metric: It is cost-effective to construct compilable training and test datasets and to evaluate the optimization performance for code size.

Metric. To be more robust to outliners, we evaluate the code size optimization results by the geometric mean of IR instruction count reduction of program s in a dataset $\mathcal{D}$:

$$R_{\mathcal{D}}(\text{agent}) = \left(\prod_{s \in \mathcal{D}} \frac{C(s_b)}{C(s_{\text{agent}})} \right)^{\frac{1}{|\mathcal{D}|}} \tag{11}$$

where C denotes the IR instruction count, s_b is the IR state optimized by LLVM's `-Oz` flag serving as a baseline, and s_{agent} is the final IR state produced by agent. A value of R above 1 indicates superior performance compared to LLVM's `-Oz` option.

Table 3: Dataset division of 8 CompilerGym benchmarks.

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

Benchmarks. We evaluate our method mainly on benchmarks from the CompilerGym platform [11]: benchmark suites including cBench [17], CHStone [28], MiBench [20], and NASA Parallel Benchmarks (NPB) [3], as well as kernels from open source projects such as BLAS [35], Linux, OpenCV [9], and TensorFlow [1]. Synthetic benchmarks from program generators [34, 59] are excluded as they lack real-world relevance. We adhere to the standard data splits of CompilerGym. 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 3.

Table 4: Autotuning results on cBench, i.e., the CompilerGym leaderboard [16], where learning-based methods report only inference time following leaderboard convention.

Method	Walltime	Code Size Reduction
CompilerDream + Guided Search	60.8s	**1.073×**
PPO + Guided Search	69.8s	1.070×
CompilerDream	2.9s	1.068×
Random Search (t=10800)	10,512.4s	1.062×
Random Search (t=3600)	3,630.8s	1.061×
Greedy Search	169.2s	1.055×
GATv2 + DD-PPO	258.1s	1.047×

Additionally, we evaluate CompilerDream on a large-scale dataset named FormAI [55]. FormAI comprises a vast collection of AI-generated C programs with diverse functionalities and coding styles. We filtered out codes that failed to compile into CompilerGym benchmarks, resulting in a test set of 109,016 programs.

Inspired by FormAI, we also constructed a dataset of 50 Objective-C programs generated by a Large Language Model to further evaluate CompilerDream's ability to generalize to different programming languages. Details can be found in Appendix A.7.

4.2 Autotuning: CompilerGym Leaderboard

We first validate CompilerDream as an autotuning method to demonstrate its ability to discover high-quality optimization sequences, supporting our goal of achieving superior code optimization.

Implementation. We target the CompilerGym leaderboard task [16], optimizing pass sequences for 23 cBench programs. Following the common setup adopted by other methods on the leaderboard, we train CompilerDream using our RL agent on all cBench programs except *ghostscript*, which is excluded due to its large size that would significantly slow down training. The action space is set to the full action space of all 124 actions in LLVM. CompilerDream is trained for 25 hours, averaging about one hour per program—comparable to the 3600-second wall time of the random search algorithm. Beyond evaluating single-trial optimization performance, we test *CompilerDream+Guided Search*, inspired by the leaderboard's leading approach. This method leverages the RL agent's policy $\pi_\psi(a_t|s_t)$ to guide random search, limiting search time to 1 minute per program (details in Appendix A.5).

Results. As shown in Table 4, our RL agent trained on world model simulations achieves an average 1.068× code size reduction on cBench in a single trial, surpassing random search methods despite their longer execution times, even when accounting for CompilerDream's training time. This demonstrates the world model's ability to accurately simulate optimization processes and support effective agent learning. Furthermore, when combined with guided search, CompilerDream outperforms the top leaderboard method while using less wall time.

Table 5: General value prediction results: The geometric mean of code size reduction achieved by the best sequences selected by different methods across 4 datasets.

Method	Dataset			
	cBench	CHStone	MiBench	NPB
Coreset-NVP	1.028	**1.101**	1.003	1.085
Coreset -CompilerDream	**1.038**	**1.101**	**1.017**	**1.140**
Coreset-Oracle	1.041	1.106	1.020	1.159

4.3 General Value Prediction

In this section, we demonstrate that CompilerDream's large-scale trained world model accurately simulates the optimization process of unseen programs, enabling the construction of an excellent value prediction agent that surpasses state-of-the-art methods.

Implementation. To ensure a fair comparison of the value prediction capabilities, we adopt the setting of the state-of-the-art Coreset-NVP method [41], which evaluates pass sequences from a fixed set of 50 distinct sequences (the *core set*). We train our world model to build a value prediction agent (detailed in Section 3.3) that predicts the value of applying sequences from the same core set. We follow the same search strategy as Coreset-NVP, enumerating the prefixes of roughly 3 or 4 sequences with the highest predicted value and selecting the best by compiler validation. Both methods are evaluated on four datasets distinct from their training sets, with an identical number of compiler validation calls to ensure a fair comparison. Additional details are provided in Appendix A.6.

Results. Table 5 presents the geometric mean reduction achieved by different methods. Despite being tested on datasets distinct from the training set, CompilerDream surpasses Coreset-NVP on three datasets, demonstrating superior compiler simulation and generalization capabilities. On MiBench, CompilerDream performs nearly optimally compared to the Oracle baseline, which uses a brute-force search and serves as the problem's upper bound. Both our method and Coreset-NVP achieve an average reduction of 1.101 on CHStone, close to the upper bound of 1.06.

4.4 General Code Optimization with RL

We finally demonstrate the ability of CompilerDream's reinforcement learning agent, which can generate optimization sequences end-to-end in a single trial for a wide variety of programs unseen during training. This scenario mirrors real-world use cases where the optimization algorithm has limited time to compute the best sequence for arbitrary programs. Therefore, all methods in this section generate only one optimization sequence per program in the test datasets unless otherwise specified.

Implementation. We train CompilerDream with its RL agent, and compare it with the following baselines: a random pass sequence, an autotuning approach using random search, LLVM's `-O0` and `-Oz` flag, and a state-of-the-art learning-based method [27] using Proximal Policy Optimization (PPO) [51]. The random search conducts hundreds of trials within a time budget similar to our RL

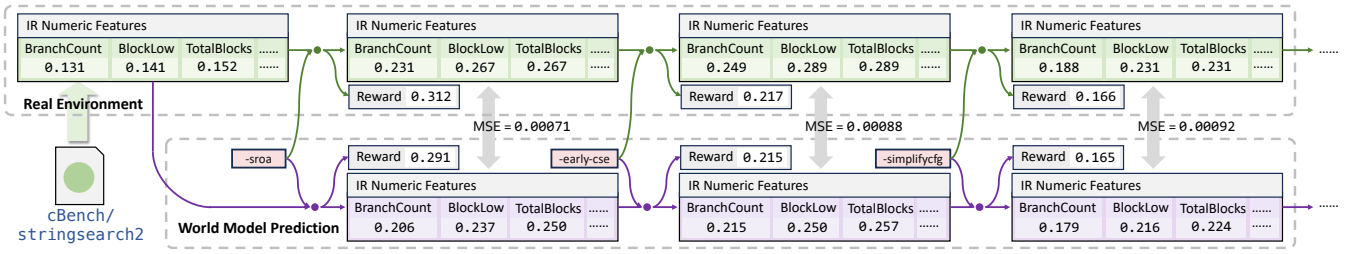

Figure 5: A comparison between a ground-truth code optimization trajectory and an imagined trajectory by a learned compiler world model. The learned world model accurately captures the variations of program features and optimization metrics.

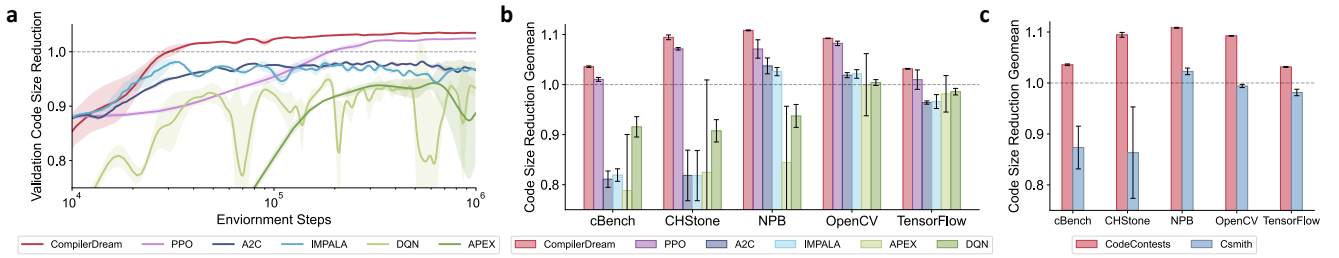

Figure 6: Analysis. Evaluations of different RL algorithms: (a) Learning curves of various RL algorithms, measured by the geometric mean of code size reduction on the CodeContests validation set. A Gaussian filter ($\sigma = 2.0$) is applied to enhance the visualization of trends. (b) Generalization capabilities of different RL algorithms on various test datasets. Effect of training dataset: (c) Test performance of CompilerDream trained on CodeContests and Csmith. Bars indicate the standard deviation.

agent. The `-Oz` flag represents LLVM's highest level of code size optimization while the `-O0` flag represents no optimization. The learning-based method using PPO and CompilerDream are trained on the large-scale CodeContests dataset and zero-shot generalized to unseen test programs. Following the state-of-the-art approach, all methods use the reduced action space (Table 1).

Results. Figure 4 presents the code size reduction achieved by CompilerDream's RL agent, measured by the geometric mean of IR instruction count reduction. Without in-domain training, CompilerDream surpasses `-Oz` in all but two benchmarks in a single trial and consistently outperforms PPO, except on the BLAS datasets. It also outperforms random search on most datasets within a comparable time budget, except for Linux. The minimal performance differences across various methods on BLAS and Linux suggest these datasets are already highly optimized.

Moreover, CompilerDream's zero-shot generalization matches or surpasses in-domain training. This advantage is particularly evident in the NPB dataset, where data sparsity limits in-domain agents, yet CompilerDream achieves an additional 3% code size reduction. Its robust generalization extends to new languages, as shown by results on the Fortran-based BLAS and NPB datasets. On the AI-generated Objective-C dataset, CompilerDream achieves an average code size reduction of 1.027×, reaching up to 2.87× in certain test cases.

On the large-scale FormAI test set (Figure 7), CompilerDream outperforms PPO, matching or outperforming `-Oz` on more programs and achieving higher optimization levels, demonstrating its superior and consistent performance.

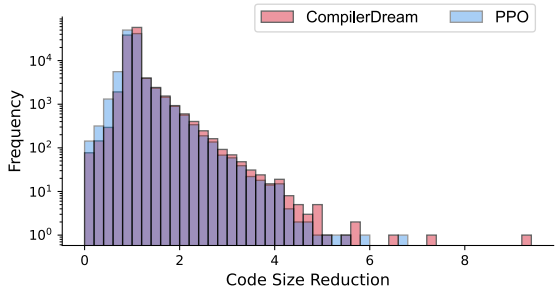

Figure 7: Histograms of large-scale evaluations comparing CompilerDream and PPO on the FormAI dataset.

4.5 Analysis

Comparison with model-free methods. We further evaluate the sample efficiency and zero-shot generalization abilities of our model-based CompilerDream (with RL agent) against various model-free counterparts, including PPO [51], DQN [48], A2C [47], APEX [29], and IMPALA [15, 51]. Figure 6a illustrates that while PPO is the strongest model-free baseline, our RL agent trained on world model simulation learns an order of magnitude faster with fewer compiler interactions. Figure 6b further highlights its superior generalization to unseen benchmarks, supporting our claim that world model-based agents better capture the dynamics of compiler optimization and enhance generalization to unseen datasets.

Comparison with model-based methods. To demonstrate the necessity and effectiveness of learning a compiler world model with deep representations of the compiler. We compare our approach with a classical model-based RL method, MBPO [31], under the RL agent setting described in Section 4.4. MBPO adopts standard MLPs for its dynamics model and does not share a latent representation space with the actor or critic. For a fair comparison, we apply reward smoothing to MBPO using the same smoothing factor as in our method. The results are shown in Table 6. While MBPO outperforms the best model-free baseline, PPO, on some tasks, it consistently underperforms CompilerDream, except on MiBench, where the performance gap across all methods remains marginal. These results suggest that model-based RL alone is insufficient, and that both a more expressive world model and a shared latent space are crucial for better optimization performance and generalization to unseen programs.

Table 6: Comparison between CompilerDream with MBPO and PPO for general code optimization.

Dataset	PPO	MBPO	CompilerDream
BLAS	**0.993**	0.988	0.991
cBench	1.010	1.020	**1.036**
CHStone	1.071	1.066	**1.094**
MiBench	0.985	**0.988**	0.986
NPB	1.000	0.996	**1.006**
Linux	1.071	1.069	**1.108**
OpenCV	1.082	0.998	**1.092**
TensorFlow	1.010	0.996	**1.032**

Effect of reward smoothing. To evaluate the effectiveness of the reward smoothing technique described in Section 3.2, we compare the performance of the RL agent trained with the complete CompilerDream method against a variant without reward smoothing. Table 7 presents these results alongside PPO for reference. The results demonstrate that reward smoothing consistently enhances optimization performance across all datasets and enables CompilerDream to outperform PPO on MiBench, Linux, and OpenCV.

Table 7: Comparison between CompilerDream with and without reward smoothing for general code optimization.

Dataset	PPO	Ours w/o Reward Smoothing	Ours w/ Reward Smoothing
BLAS	**0.993**	0.987	0.991
cBench	1.010	1.021	**1.036**
CHStone	1.071	1.076	**1.094**
MiBench	1.000	0.995	**1.006**
NPB	1.071	1.093	**1.108**
Linux	0.985	0.980	**0.986**
OpenCV	1.082	1.080	**1.092**
TensorFlow	1.010	1.022	**1.032**

Effect of training dataset. To assess the impact of the CodeContests dataset on the generalization ability, we compared it with the commonly used Csmith [59] dataset, a large LLVM IR dataset generated by rules. We train CompilerDream with RL agent on both datasets and the results shown in Figure 6 indicate that CompilerDream trained on CodeContests significantly outperforms the Csmith-trained version across all five test datasets. This advantage is particularly evident in the manually curated cBench [17] and CHStone [28] datasets, demonstrating that the CodeContests-trained model can generalize more effectively to human-written programs.

Program showcase. In Figure 5, 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.

5 Discussion

We aim to address the major challenge of generalization in learning-based code optimization by introducing the CompilerDream approach, which leverages the simulation and generalization capabilities of a world model. By incorporating a reward smoothing technique and a large-scale training program dataset, we enable effective world model training for compiler optimization tasks. Our method supports two types of optimization agents: a value prediction agent and an RL agent. Experimental results demonstrate that CompilerDream achieves superior optimization performance across diverse problem scenarios and program datasets, surpassing built-in compiler optimization flags and state-of-the-art methods.

Limitation. Although our method can naturally extend to optimization objectives such as execution time and object file size reduction, we focus solely on code size optimization for scalability and stability, as discussed in Section 4.1. Execution time measurements often exhibit high variance and biases due to hardware and operating system states, introducing significant noise that additional sampling cannot mitigate. Furthermore, existing open-source frameworks, such as CompilerGym, lack robust support for accurate runtime measurement, making it challenging to obtain reliable signals for effective learning. As a result, we only focus on code size reduction for this paper.

Future Works. There is substantial scope for further exploration, including expansion of the training dataset, scaling up the compiler world model, optimizing multiple objectives like execution time, and enriching feature and action spaces with deeper expert knowledge or large language models. Additionally, our approach could support more types of optimization agents, such as search agents leveraging advanced tree search algorithms [50] based on world model simulations.

6 Acknowledgments

This work was supported by the National Natural Science Foundation of China (62021002).

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 265–283.

[2] Ankesh Anand, Jacob C Walker, Yazhe Li, Eszter Vértes, Julian Schrittwieser, Sherjil Ozair, Theophane Weber, and Jessica B Hamrick. 2022. Procedural generalization by planning with self-supervised world models. In *Proceedings of the 10th International Conference on Learning Representations (ICLR '22)*.

[3] David Bailey, Tim Harris, William Saphir, Rob Van Der Wijngaart, Alex Woo, and Maurice Yarrow. 1995. *The NAS parallel benchmarks 2.0.* Technical Report. Technical Report NAS-95-020, NASA Ames Research Center.

[4] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. 2018. Neural code comprehension: a learnable representation of code semantics. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems (NeurIPS '18)*. 3589–3601.

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

[6] Alexander Brauckmann, Andrés Goens, and Jeronimo Castrillon. 2024. PolyGym: Polyhedral Optimizations as an Environment for Reinforcement Learning. In *Proceedings of the 30th International Conference on Parallel Architectures and Compilation Techniques* (Atlanta, GA, USA) *(PACT '21)*. IEEE Press, 17–29.

[7] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. In *Proceedings of the 34th International Conference on Neural Information Processing Systems (NeurIPS '20)*. Article 159, 25 pages.

[8] Brad Calder, Dirk Grunwald, Michael Jones, Donald Lindsay, James Martin, Michael Mozer, and Benjamin Zorn. 1997. Evidence-based static branch prediction using machine learning. *ACM Transactions on Programming Languages and Systems (TOPLAS '97)* 19, 1 (1997), 188–222.

[9] Ivan Culjak, David Abram, Tomislav Pribanic, Hrvoje Dzapo, and Mario Cifrek. 2012. A brief introduction to OpenCV. In *2012 Proceedings of the 35th International Convention MIPRO (MIPRO '12)*. 1725–1730.

[10] Chris Cummins, Zacharias V. Fisches, Tal Ben-Nun, Torsten Hoefler, Michael F P O'Boyle, and Hugh Leather. 2021. ProGraML: A Graph-based Program Representation for Data Flow Analysis and Compiler Optimizations. In *Proceedings of the 38th International Conference on Machine Learning (ICML '21)*. 2244–2253.

[11] Chris Cummins, Bram Wasti, Jiadong Guo, Brandon Cui, Jason Ansel, Sahir Gomez, Somya Jain, Jia Liu, Olivier Teytaud, Benoit Steiner, Yuandong Tian, and Hugh Leather. 2022. CompilerGym: robust, performant compiler optimization environments for AI research. In *Proceedings of the 20th IEEE/ACM International Symposium on Code Generation and Optimization (CGO '22)*. IEEE Press, 92–105.

[12] Anderson Faustino da Silva, Bruno Conde Kind, José Wesley de Souza Magalhães, Jerônimo Nunes Rocha, Breno Campos Ferreira Guimarães, and Fernando Magno Quintão Pereira. 2021. AnghaBench: a suite with one million compilable C benchmarks for code-size reduction. In *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO '21)*. IEEE Press, 378–390.

[13] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. 2008. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing* (Austin, Texas) *(SC '08)*. IEEE Press, Article 4, 12 pages.

[14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, (NAACL-HLT '19)*. Association for Computational Linguistics, 4171–4186.

[15] Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Vlad Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, et al. 2018. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. In *International Conference on Machine Learning (ICML '18)*. 1407–1416.

[16] Facebook. 2022. CompilerGym Leaderboard. https://github.com/facebookresearch/CompilerGym?tab=readme-ov-file#leaderboards. Accessed: 2024-8-09.

[17] Grigori Fursin, John Cavazos, Michael O'Boyle, and Olivier Temam. 2007. Midatasets: Creating the conditions for a more realistic evaluation of iterative optimization. In *International conference on high-performance embedded architectures and compilers (HiPEAC '07)*. Springer-Verlag, Berlin, Heidelberg, 245–260.

[18] Grigori Fursin, Cupertino Miranda, Olivier Temam, Mircea Namolaru, Ayal Zaks, Bilha Mendelson, Edwin Bonilla, John Thomson, Hugh Leather, Chris Williams, et al. 2008. MILEPOST GCC: machine learning based research compiler. In *GCC summit*. Ottawa, Canada.

[19] Kyriakos Georgiou, Craig Blackmore, Samuel Xavier-de Souza, and Kerstin Eder. 2018. Less is more: Exploiting the standard compiler optimization levels for better performance and energy consumption. In *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems (SCOPES '18)*. 35–42.

[20] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the fourth annual IEEE international workshop on workload characterization (WWC '01)*. 3–14.

[21] David Ha and Jürgen Schmidhuber. 2018. Recurrent world models facilitate policy evolution. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems (NeurIPS '18)*. 2455–2467.

[22] Tuomas Haarnoja, Aurick Zhou, Kristian Hartikainen, George Tucker, Sehoon Ha, Jie Tan, Vikash Kumar, Henry Zhu, Abhishek Gupta, Pieter Abbeel, and Sergey Levine. 2019. Soft Actor-Critic Algorithms and Applications. arXiv:1812.05905 [cs.LG] https://arxiv.org/abs/1812.05905

[23] Danijar Hafner, Timothy Lillicrap, Jimmy Ba, and Mohammad Norouzi. 2020. Dream to Control: Learning Behaviors by Latent Imagination. arXiv:1912.01603 [cs.LG] https://arxiv.org/abs/1912.01603

[24] Danijar Hafner, Timothy P Lillicrap, Mohammad Norouzi, and Jimmy Ba. 2021. Mastering Atari with Discrete World Models. In *International Conference on Learning Representations (ICLR '21)*.

[25] Danijar Hafner, Jurgis Pasukonis, Jimmy Ba, and Timothy Lillicrap. 2023. Mastering diverse domains through world models. *arXiv preprint arXiv:2301.04104* (2023).

[26] Ameer Haj-Ali, Nesreen K Ahmed, Ted Willke, Yakun Sophia Shao, Krste Asanovic, and Ion Stoica. 2020. Neurovectorizer: End-to-end vectorization with deep reinforcement learning. In *International Symposium on Code Generation and Optimization (CGO '20)*. IEEE Press, 242–255.

[27] Ameer Haj-Ali, Qijing (Jenny) Huang, William S. Moses, John Xiang, Krste Asanovic, John Wawrzynek, and Ion Stoica. 2020. AutoPhase: Juggling HLS Phase Orderings in Random Forests with Deep Reinforcement Learning. In *Proceedings of Machine Learning and Systems (MLSys '20)*. 70–81.

[28] Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, Hiroaki Takada, and Katsuya Ishii. 2008. CHStone: A benchmark program suite for practical C-based high-level synthesis. In *2008 IEEE International Symposium on Circuits and Systems (ISCAS '08)*. 1192–1195.

[29] Dan Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado van Hasselt, and David Silver. 2018. Distributed Prioritized Experience Replay. In *International Conference on Learning Representations (ICLR '18)*.

[30] Shalini Jain, Yashas Andaluri, S VenkataKeerthy, and Ramakrishna Upadrasta. 2022. POSET-RL: Phase ordering for optimizing size and execution time using reinforcement learning. In *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS '22)*. IEEE, 121–131.

[31] Michael Janner, Justin Fu, Marvin Zhang, and Sergey Levine. 2019. When to trust your model: Model-based policy optimization. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems (NeurIPS '19)*. Article 1122.

[32] Alexander Kirillov, Eric Mintun, Nikhila Ravi, Hanzi Mao, Chloe Rolland, Laura Gustafson, Tete Xiao, Spencer Whitehead, Alexander C. Berg, Wan-Yen Lo, Piotr Dollar, and Ross Girshick. 2023. Segment Anything. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV '23)*. 4015–4026.

[33] Sameer Kulkarni and John Cavazos. 2012. Mitigating the compiler optimization phase-ordering problem using machine learning. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*. 147–162.

[34] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization (CGO '04)*. IEEE Press, 75–86.

[35] Chuck L Lawson, Richard J. Hanson, David R Kincaid, and Fred T. Krogh. 1979. Basic linear algebra subprograms for Fortran usage. *ACM Transactions on Mathematical Software (TOMS '79)* 5, 3 (1979), 308–323.

[36] Hugh Leather and Chris Cummins. 2020. Machine learning in compilers: Past, present and future. In *2020 Forum for Specification and Design Languages (FDL)*. IEEE, 1–8.

[37] Yann LeCun. 2022. A path towards autonomous machine intelligence version 0.9. 2, 2022-06-27. *Open Review* 62, 1 (2022), 1–62. https://openreview.net/pdf?id=BZ5a1r-kVsf

[38] Vint Lee, Pieter Abbeel, and Youngwoon Lee. 2024. DreamSmooth: Improving Model-based Reinforcement Learning via Reward Smoothing. In *International Conference on Learning Representations (ICLR '24)*.

[39] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science* 378, 6624 (2022), 1092–1097.

[40] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph E. Gonzalez, Michael I. Jordan, and Ion Stoica. 2018. RLlib: Abstractions for Distributed Reinforcement Learning. In *International Conference on Machine Learning (ICML '18)*, Vol. 80. 3053–3062.

[41] Youwei Liang, Kevin Stone, Ali Shameli, Chris Cummins, Mostafa Elhoushi, Jiadong Guo, Benoit Steiner, Xiaomeng Yang, Pengtao Xie, Hugh Leather, and

Yuandong Tian. 2023. Learning Compiler Pass Orders using Coreset and Normalized Value Prediction. In *Proceedings of the 40th International Conference on Machine Learning (ICML '23)*.
[42] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. 2009. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '09)*. 45–55.
[43] Haoyu Ma, Jialong Wu, Ningya Feng, Chenjun Xiao, Dong Li, HAO Jianye, Jianmin Wang, and Mingsheng Long. 2024. HarmonyDream: Task Harmonization Inside World Models. In *International Conference on Machine Learning (ICML '24)*.
[44] Rahim Mammadli, Ali Jannesari, and Felix Wolf. 2020. Static Neural Compiler Optimization via Deep Reinforcement Learning. In *Proceedings of the 2020 IEEE/ACM 6th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC '20) and Workshop on Hierarchical Parallelism for Exascale Computing (HiPar '20)*. 1–11.
[45] Bogdan Mazoure, Ahmed M Ahmed, R Devon Hjelm, Andrey Kolobov, and Patrick MacAlpine. 2022. Cross-Trajectory Representation Learning for Zero-Shot Generalization in RL. In *International Conference on Learning Representations (ICLR '22)*.
[46] Azalia Mirhoseini, Hieu Pham, Quoc V Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. 2017. Device placement optimization with reinforcement learning. In *International Conference on Machine Learning (ICML '17)*. 2430–2439.
[47] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. 2016. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning (ICML '16)*. 1928–1937.
[48] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. 2015. Human-level control through deep reinforcement learning. *Nature* 518, 7540 (2015), 529–533.
[49] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. 2021. Learning transferable visual models from natural language supervision. In *International Conference on Machine Learning (ICML '21)*. 8748–8763.
[50] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, et al. 2020. Mastering atari, go, chess and shogi by planning with a learned model. *Nature* 588, 7839 (2020), 604–609.
[51] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal Policy Optimization Algorithms. arXiv:1707.06347 [cs.LG] https://arxiv.org/abs/1707.06347
[52] Hafsah Shahzad, Ahmed Sanaullah, Sanjay Arora, Robert Munafo, Xiteng Yao, Ulrich Drepper, and Martin Herbordt. 2022. Reinforcement Learning Strategies for Compiler Optimization in High Level Synthesis. In *Proceedings of the 2022 IEEE/ACM 8th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC '22) and Workshop on Hierarchical Parallelism for Exascale Computing (HiPar '22)*. 13–22.
[53] Mark Stephenson, Saman Amarasinghe, Martin Martin, and Una-May O'Reilly. 2003. Meta optimization: Improving compiler heuristics with machine learning. *ACM sigplan notices* 38, 5 (2003), 77–90.
[54] Richard S Sutton and Andrew G Barto. 2018. *Reinforcement learning: An introduction*. MIT press.
[55] Norbert Tihanyi, Tamas Bisztray, Ridhi Jain, Mohamed Amine Ferrag, Lucas C Cordeiro, and Vasileios Mavroeidis. 2023. The formai dataset: Generative ai in software security through the lens of formal verification. In *Proceedings of the 19th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE '23)*. 33–43.
[56] Spyridon Triantafyllis, Manish Vachharajani, Neil Vachharajani, and David I August. 2003. Compiler optimization-space exploration. In *International Symposium on Code Generation and Optimization (CGO '03)*. IEEE Press, 204–215.
[57] Mircea Trofin, Yundi Qian, Eugene Brevdo, Zinan Lin, Krzysztof Choromanski, and David Li. 2021. MLGO: a Machine Learning Guided Compiler Optimizations Framework. arXiv:2101.04808 [cs.PL] https://arxiv.org/abs/2101.04808
[58] Ronald J Williams. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning* 8 (1992), 229–256.
[59] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation (PLDI 2018)*. 283–294.

A Implementation Details

A.1 Compiler Environment

Our experiments are conducted on the CompilerGym platform [11], version 0.2.5, with LLVM-10.0.0 integration.

Table 8: Hyperparameters in our experiments.

Hyperparameter	Value
RSSM recurrent units	1024
RSSM number of latents	32
RSSM classes per latent	32
MLP layers	4
MLP hidden units	400
Activation	LayerNorm + SiLU
Random exploration	500 environment steps
Replay buffer capacity	2×10^6
Reward smoothing α [38]	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

A.2 Hyperparameters

CompilerDream. The hyperparameters for our world model and agent implementation are outlined in Table 8. For hyperparameters not specified, we use the same value as the DreamerV3 [25]. Most of the listed hyperparameter values are directly taken from DreamerV2 [24] or previous works, with minimal tuning. However, the *loss scales* are carefully tuned, as the observation loss is relatively small compared to the reward loss in our setting, and the balance between these losses has a significant impact on CompilerDream's performance. All experiments share the same set of hyperparameters unless otherwise specified.

Model-based Baseline. For MBPO in Section 4.5, the dynamics model is a 4-layer MLP with a hidden size of 1024. We adopt the implementation from https://github.com/Xingyu-Lin/mbpo_pytorch, with most hyperparameters left unchanged. The modified hyperparameters are listed in Table 10.

Model-free Baselines. We use RLlib [40] to train and test model-free reinforcement learning algorithms including PPO [51] [27], A2C [47], IMPALA [15], APEX [29], and DQN [48]. We use default hyperparameters of algorithms in RLlib following the CompilerGym platform [11], except that we have carefully tuned the hyperparameters for PPO, as listed in Table 9. We explored roughly 2 or 3

Table 9: Hyperparameters for the PPO baseline, well-tuned on our dataset to be deviating from the default value in RLlib.

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

Table 10: Modified hyperparameters in our MBPO baseline.

	Hyperparameters	Value
MBPO	target_update_interval	10
	lr	1e-3
	rollout_batch_size	50000
	rollout_max_length	45
	init_exploration_steps	3600

values for each hyperparameter listed and selected the set of hyperparameters that yielded the best performance on the validation set. The results of the PPO baseline in Section 4.5 and Figure 6 are exactly the same as those in Section 4.4 and Figure 4.

A.3 Hardware and Training Time

CompilerDream. We train all CompilerDream-based methods with 64 CPUs and an RTX-3090 GPU. In Section 4.4, we trained CompilerDream on the CodeContests dataset for around 1 day and 20 hours. In Sections 4.2 and 4.3, CompilerDream was trained for about 1 day.

Random Search. The random search baseline in Figure 4 is conducted with 4 CPUs, which is sufficient since it is a single-thread program. The random search baseline in Table 4 utilizes 80 CPUs, as specified in the write-up attached to CompilerGym's leaderboard [16]. Our *CompilerDream + Guided Search* method in Table 4 is tested on the same machine used for training CompilerDream, equipped with 64 CPUs and an RTX-3090 GPU.

Model-free and Model-based baselines. The hardware setting is the same as used for training CompilerDream. For the PPO baseline in Section 4.4, we train for about 7 hours, as longer training leads to overfitting on CodeContests. Other model-free methods in Section 4.4 are trained for at least 10 hours. We use 5 workers for environment interaction and 4 evaluation workers to assess checkpoints on the validation set. In Section 4.5, we train MBPO for approximately 10 hours, as its evaluation score converges within this time.

A.4 Random Seeds

All experiments reporting a min-max range or standard deviation are conducted with three different random seeds. The results in Table 5, Table 6, and Table 7 are also averaged over three runs with different seeds. The seeds are randomly selected, and the results are generally consistent across other random seeds as well.

A.5 Detail of Guided Search in Autotuning

Our guided search in Section 4.2 follows the *PPO+Guided Search* design. Actions are sampled from the learned policy for up to 45 steps per episode, recording code size reductions after each step to track the maximum. To encourage exploration, 5% of actions are sampled uniformly at random. We record the best reduction so far and monitor elapsed time to stay within the 1-minute budget per benchmark. Unlike *PPO+Guided Search* (Table 4), which uses a 200-step horizon and an extra 500-step search on the best sequence, we omit the latter and use only 45 steps, making our wall time slightly shorter.

A.6 Detail of Value Prediction Experiment

We provide additional details for the experiment in Section 4.3. The baseline, Coreset-NVP[41], trains a value model to score sequences from a fixed *core set*, obtained via extensive search on its training set. For each program, it evaluates the top-scored sequences in order, executing up to 45 passes across them. After each full sequence, the IR resets to the initial state. Code size reductions after each pass are tracked, and the best-performing pass and prefix are selected. The core set includes 50 sequences (625 passes total, 12.5 passes each on average), so around 3–4 sequences are tried per benchmark. The 79 passes used correspond to the optimization actions in Section 3.1.

In our setup, the action space includes all 124 LLVM passes. To train the world model to predict cumulative reward values for the core set passes, we apply each core set sequence to training programs and collect trajectories. We use the CodeContests dataset (Section 3.4), consistent with the experiments in Section 4.4. Reward loss weight is set to 100.0, with other hyperparameters unchanged. For evaluation, we compile the top 45 predicted prefixes per program and report the best.

A.7 AI-Generated Benchmarks

To further test the generalization ability of our CompilerDream agent on different programming languages, we borrow the method from FormAI [55] 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.