# Exponentially Expanding the Phase-Ordering Search Space via Dormant Information

Ruobing Han
Georgia Institute of Technology
USA
hanruobing@gatech.edu

Hyesoon Kim
Georgia Institute of Technology
USA
hyesoon@cc.gatech.edu

## Abstract

Applying compilation transformations in optimal sequences can significantly improve program speed and reduce code size. However, finding these optimal sequences—a problem known as the phase-ordering problem—remains a long-standing challenge. Specifically, modern compilers offer hundreds of available transformations, making the search space too large to explore efficiently within a reasonable timeframe. Existing solutions address this problem by grouping transformations into short sequences based on prior knowledge from human experts, and then searching for optimal orders among these sequences. Such pruning methods are aggressive, potentially excluding optimal solutions from the search space. Additionally, they rely on prior knowledge and lack scalability when applied to new transformations.

In this paper, we propose a more conservative pruning approach. The insight of this new approach is to capture the dormant information and utilize it to guide the search process. By excluding dormant transformations, this approach significantly prunes the search space while retaining the optimal solutions. Moreover, it does not rely on any prior human knowledge, making it scalable to new transformations.

To demonstrate the efficacy of the conservative approach, we integrate it with a classical Reinforcement Learning model, which was previously used with aggressive pruning methods. Our solution, named FlexPO, is capable of exploring a search space exponentially larger than those considered in existing solutions. Experimental results show that FlexPO generates programs that are 12% faster or 17.6% smaller than the programs produced by modern compilers.

***CCS Concepts:*** • **Software and its engineering → Compilers**.

***Keywords:*** compiler, phase ordering, reinforcement learning



*CC '24, March 2–3, 2024, Edinburgh, United Kingdom*

ACM ISBN 979-8-4007-0507-6/24/03
https://doi.org/10.1145/3640537.3641582

**ACM Reference Format:**
Ruobing Han and Hyesoon Kim. 2024. Exponentially Expanding the Phase-Ordering Search Space via Dormant Information. In *Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction (CC '24), March 2–3, 2024, Edinburgh, United Kingdom.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3640537.3641582

## 1 Introduction

Modern compilers offer hundreds of compilation transformations to modify programs. Extensive research [2, 3, 26, 28] has shown that strategically applying these transformations in an optimal sequence can significantly enhance program runtime, with improvements ranging from 2.4% to 60.37%.

Most modern compilers offer predefined transformation sequences (e.g., O3, Oz, Os), which are based on empirical evidence. However, several studies [3, 17, 22, 29] indicate that applying a uniform transformation sequence across a diverse range of programs often leads to suboptimal performance. This suggests that these sequences are overly generalized and lack the specificity needed to achieve the best possible performance for each unique program.

The phase-ordering problem, which involves selecting and applying transformations in optimal sequences, remains an unresolved challenge. This complexity stems from the vast and intricate search space, expanding exponentially with the number of transformations involved. For $N$ available transformations, the search space encompasses $N^L$ potential sequences when applying $L$ transformations. Furthermore, predicting sequence outcomes is difficult due to the complex interactions among transformations [17, 37].

Some researchers [3, 22, 29] propose a pruning mechanism to narrow the search space. This method clusters transformations into short sequences based on prior human knowledge, then applies search algorithms to select and sequence these short sequences. This strategy effectively shrinks the search space when compared to the ordering of individual transformations, facilitating the discovery of satisfactory solutions within a feasible timeframe. For instance, given $N$ transformations and a target sequence length $L$, by grouping $G$ transformations together, we form $\frac{N}{G}$ sequences. Consequently, we only need to choose $\frac{L}{G}$ of these sequences to construct the final solution. Hence, the revised search space is $\left(\frac{N}{G}\right)^{\frac{L}{G}}$, a significant reduction from the original $N^L$.

Nevertheless, this approach has two limitations. First, the pruning mechanism is **aggressive**, potentially excluding optimal solutions from the search space. Second, the mechanism relies on human experts to form transformation sequences, making it **less scalable** for new transformations. We refer to this mechanism as *aggressive pruning*.

In this paper, we propose a new pruning mechanism. This mechanism detects transformations that will not change the program (a.k.a. dormant transformations). It then guides the search process to focus solely on transformations that are likely to change the program (a.k.a. active transformations). Compared to the aggressive pruning solution, the new mechanism is **conservative**, as it only prunes non-optimal sequences that contain dormant transformations from the search space. Furthermore, it offers **scalability** for new transformations, since it does not rely on prior knowledge of the transformations.

The proposed conservative pruning can replace the existing aggressive pruning mechanism. Specifically, we introduce FlexPO, a framework that integrates conservative pruning with a classical RL model that has been used along with aggressive pruning mechanisms [3, 22, 29]. With the new pruning approach, FlexPO can explore a search space that is exponentially larger than those of other solutions, as shown in Figure 1.

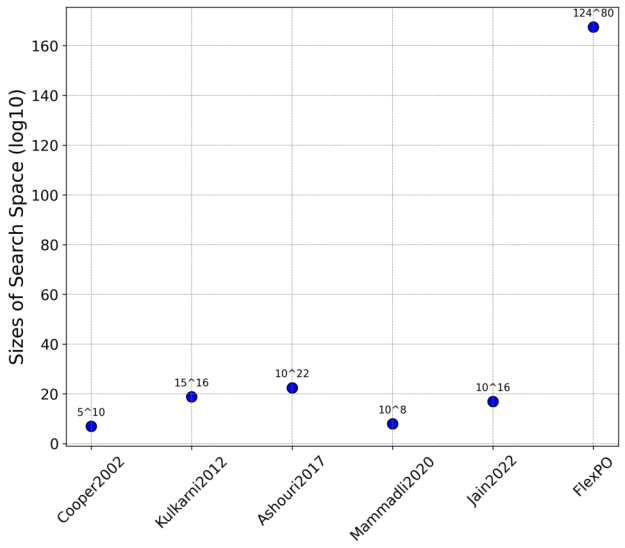



**Figure 1.** FlexPO supports a search space that is exponentially larger than existing solutions.

The contributions of our paper are listed below:

- Highlight the importance of utilizing dormant information, which has been long overlooked by phase-ordering solutions.
- Propose the utilization of a ML model to ascertain the active/dormant status of transformations.
- Integrate the conservative pruning with a RL model and evaluate the effectiveness of the search process.

## 2 Background

### 2.1 Phase Ordering

As summarized in [11], the best compilation sequence depends on the following factors: source code, target machine, available transformations, and optimization targets (e.g., code size and runtime performance). For modern compilers that contain hundreds of transformations, the search space is too large to be fully explored. Additionally, the interaction between transformations makes the problem more difficult, and exchanging the order of two transformations sometimes generates different outputs. As shown in Figure 2, in the upper part, both loop-invariant code motion (LICM) and loop unroll change the program. However, in the lower part, after the loop unroll transformation, there is no loop in the program. Thus, applying LICM after loop unroll does not change the program.

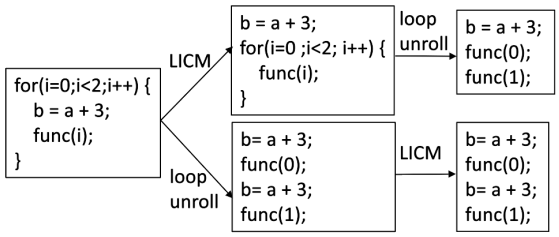



**Figure 2.** Applying the same transformations in different orders results in varied outcomes.

### 2.2 Reinforcement Learning Solution

Reinforcement Learning (RL) has emerged as a promising approach to solving the phase-ordering problem. In essence, RL involves constructing an **agent** that executes **actions** in response to **observations** to maximize the cumulative **reward** within a specified **environment**. Key concepts in RL include:

- Agent: The agent should be a model that accepts observation as input and outputs an action that is expected to achieve the maximum total rewards.
- Reward: The metrics to evaluate whether or not the actions selected by the agent are good. For the phase-ordering problem, the reward could be a decrease in code size or a decrease in runtime.
- Action: The actions could be applied to change the environment. For the phase-ordering problem, the actions are compilation transformations.
- Environment: The environment is the object that is changed by the actions. It also generates observations fed to the agent. For the phase-ordering problem, the environment is the input program.

- Observation: The features of the environment, which serve as the input for the agent. The agent relies on these observations to select actions.

The traditional RL process, when applied to the phase-ordering problem, is illustrated in Figure 3. This process involves multiple iterations, during each of which the agent observes the intermediate representation (IR), selects and applies a transformation to the IR, and then evaluates the effectiveness of the applied transformation. The methodologies presented in [3, 22, 29] adopt similar RL processes to solve the phase-ordering problem.

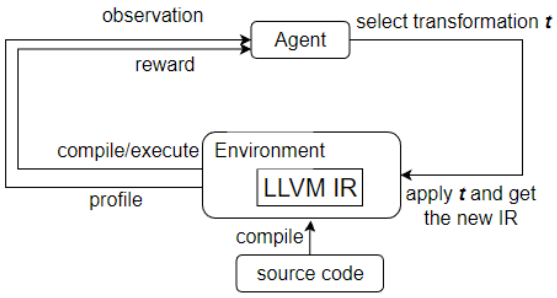



**Figure 3.** Workflow of the classical RL process in solving the phase-ordering problem.

## 3 Dormant History

A transformation is termed 'dormant' when its application does not change the program; otherwise, it is referred to as 'active.' For instance, in the lower part of Figure 2, the loop unroll is active, whereas LICM is dormant.

In this paper, we introduce a new term to address the phase-ordering problem: dormant history. This term denotes a record of the active/dormant status for each applied transformation. Figure 4 provides an illustration of dormant history. When an identical sequence of five transformations is applied to three different programs, it results in varying dormant histories for each program.

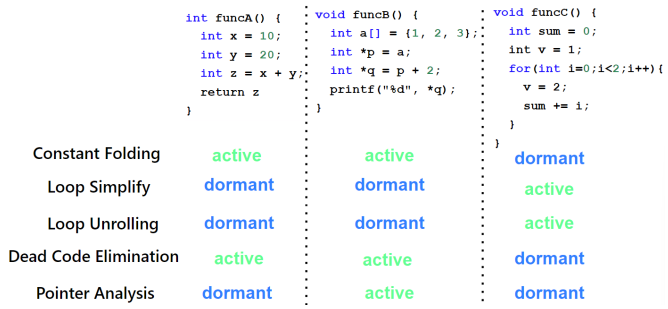



**Figure 4.** Dormant histories of the same sequence across three different programs.

### 3.1 The Importance of Dormant History

The dormant history can reveal significant features of the programs. For instance, in Figure 4, the dormant statuses for both applied loop transformations (Loop Simplify and Loop Unrolling) in the first two programs suggests that these programs likely contain no loop structures. Furthermore, the activation of the Constant Folding typically followed by the activation of Dead Code Elimination. This indicates that programs, with the application of Constant Folding, often contain dead code that can be removed. Essentially, dormant history provides insight into two pivotal aspects: the features of the programs being transformed and the likelihood of subsequent transformations being active. Both elements are crucial for the phase-ordering problem. To the best of our knowledge, this is the first attempt to leverage dormant history for solving the phase-ordering problem.

Capturing the dormant history is straightforward. As shown in Figure 3, within the search process, a transformations is applied in each iteration. Recording whether these transformations are dormant incurs only a negligible cost.

### 3.2 Utilizing Dormant History

Given that dormant history reflects essential characteristics of programs, in this section, we propose using it to apply conservative pruning. The core idea is to construct an ML model (`activation predictor`) that takes dormant history as input and predicts the likelihood of subsequent transformations being active.

Figure 5 illustrates an example. The input (dormant history) is represented as a vector with the length same as the number of transformations. Each element in the vector can be 1, 0, or −1, indicating whether the corresponding transformation has been applied and, if so, whether it was active. For example, the first element of the input vector is 1, signifying that the corresponding transformation (Loop Unrolling) was applied and modified the program (active). The second element is 0, indicating that Dead Code Elimination has not been applied. The third element is −1, suggesting that the inline transformation was applied but had no effect (dormant). The output is also a vector, where each element represents the probability that the corresponding transformation will be active if applied subsequently.

In the example shown, the predictor notes that the inline transformation has been applied and it was dormant, making it unlikely to have an effect if applied again. Consequently, its probability (the third element in the output vector) is close to zero. The predicted active probabilities are crucial in the phase-ordering search process, as they guide the search process to explore transformations more likely to be active.

## 4 FlexPO

To illustrate the significance of utilizing dormant history in addressing the phase-ordering problem, we incorporate

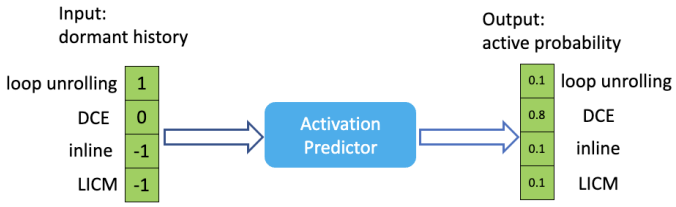



**Figure 5.** The activation predictor predicts the probability of whether transformations are active, by observing the dormant history of the applied transformations. Each item represents a value in the range [0,1], indicating the likelihood of the corresponding transformation being active.

the activation predictor into a conventional RL model. We intentionally employ a RL model that is identical or similar to those used in other studies [3, 15, 22, 29], to highlight the enhancements attributed to the activation predictor and to isolate the impact from the RL algorithms themselves.

The workflow of FlexPO, as depicted in Figure 3, begins with compiling the input program into LLVM IR without applying any transformations, producing what we call *raw_IR*. FlexPO conducts a search for optimal transformation sequences on *raw_IR* over $N$ episodes, with each episode comprising $L$ iterations. It is important to note that $N$ and $L$ are hyperparameters, which are set empirically. At the beginning of each episode, the current LLVM IR (*curr_IR*) is set to the initial *raw_IR*. FlexPO then carries out $L$ iterations sequentially. At the start of each iteration, *curr_IR*, along with the dormant history, is provided to the RL agent as an observation. The agent analyzes the observation and selects a transformation $t$. The agent also outputs an estimated value, denoted as *estimated_reward*, which represents the expected total rewards obtainable by applying the selected transformation. This estimated value is recorded for the purpose of updating the agent. Subsequently, FlexPO applies the chosen transformation $t$ to *curr_IR*, resulting in a new LLVM IR termed *new_IR*. FlexPO then compiles *new_IR* and compares its runtime performance or code size to that of *curr_IR*. Any improvements are recorded as rewards for updating the RL agent. After completing an iteration, *curr_IR* is set to *new_IR*. Once all $L$ iterations are finished, FlexPO updates the RL agent using the recorded data. For RL model updating, FlexPO applies the generic Proximal Policy Optimization algorithm [32].

The details of FlexPO are outlined as follows:

**Observation**: The LLVM IR in FlexPO is regarded as a string, which cannot be directly processed by DNNs. Thus, FlexPO utilizes the method proposed in AutoPhase [18] to convert an LLVM IR string into a vector with length 56. Each element in the vector represents a compilation feature (e.g., the number of branch instructions, the number of critical edges, and the number of binary operations with a constant operand). All features are static and can be extracted directly from the IRs, eliminating the need for compilation and execution. Other methods exist for parsing LLVM IRs, which we discuss in Section 6. We opted for AutoPhase due to its lightweight and straightforward nature. Furthermore, our evaluations show that AutoPhase sufficiently enables FlexPO to identify sequences that surpass LLVM O3 and Oz pipelines.

**Agent**: FlexPO uses the Actor-Critic approach. In addition to the actor and critic components, the Agent also incorporates the activation predictor introduced in Section 3. All three components are implemented as four-layer fully-connected DNNs with residual connections [20]. The structure of the Agent is depicted in Figure 6.

The actor accepts the feature vector generated by AutoPhase as input and generates an output vector (beneficial probability), whose length matches the number of transformations. This output vector represents the probability distribution for all transformations, with higher probabilities assigned to transformations that are likely to yield greater benefits. To generate the probability distribution, the actor uses the attention mechanism that is widely used for Computer Vision [6] and Natural Language Process [34]. The insight of the attention mechanism is that different compiler transformations are related to different features. For example, to determine whether or not to apply $-break-crit-edges$ transformations, the number of critical edges in the input vector should be an important factor, while other features, like the number of call instructions, shouldn't have much effect. With the attention mechanism, the probabilities of different transformations are affected by different elements in the input vector.

To facilitate conservative pruning, the activation predictor is integrated. It uses the dormant history to predict the dormant/active status of subsequent transformations. The output vector (active probability) of the activation predictor is then element-wise multiplied with the actor's output vector (beneficial probability), creating the final distribution (transformation probability). This final distribution is used to select the transformation to be applied next.

The critic estimates how much speed up or code size decrease can be achieved from the current IRs. The estimated values is used to update the RL model.

**Environment:** In FlexPO, the environment applies the chosen transformation to the current IR to generate a new IR. Subsequently, the new IR is compiled to obtain the reward value. If FlexPO is oriented towards runtime optimization, the environment will additionally execute the compiled programs and profile the runtime data. This procedure can be manually replicated as delineated in Listing 1.

When optimizing for runtime, in each iteration, the environment needs to compile the IR and execute the program, resulting in heavy workloads, especially when the programs take a long time to execute. To accelerate the search process, FlexPO utilizes caching to store previous evaluation results. As reported in [25], many sequences generate identical IRs

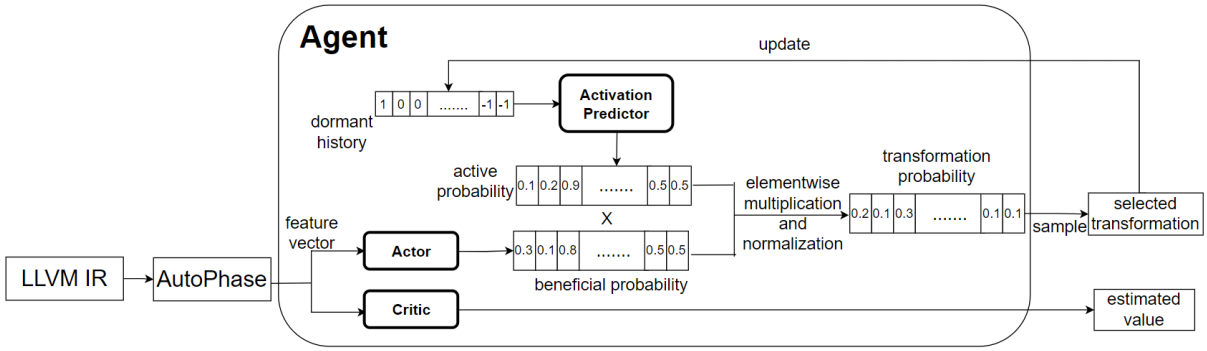



**Figure 6.** The workflow of the Agent in FlexPO.

during the search process. This occurs for two reasons: first, some transformations are dormant; second, certain pairs of transformations are independent. Therefore, changing the order of these pairs results in the same IRs. In FlexPO, a dictionary is utilized to record the executed IRs and their runtime results. When the environment requires the runtime for an LLVM IR, it first searches the dictionary to determine if this IR has been evaluated previously. If the IR is found, its evaluation result is immediately returned. Otherwise, the environment compiles and executes the program, updates the dictionary, and then returns the results.

```
# apply the selected transformation (LICM)
opt -licm LLVM_IR.bc -o LLVM_IR.bc
# compile the transformed IR with LLC. The O3
 here is for hardware dependent optimizations,
  not the same as the O3 in Clang
llc -O3 -filetype=obj LLVM_IR.bc -o LLVM_IR.o
```

**Listing 1.** The instructions used in the Environment, which apply transformations and compile LLVM IRs.

**Reward**: The definition of the reward is the improvement obtained from applying the selected transformations. For runtime optimization, the reward is defined as the normalized decrease in runtime; for code size optimization, it is the normalized reduction in code size.

$$R_{time} = \frac{runtime_{old} - runtime_{new}}{runtime_{unoptimized}}$$

$$R_{size} = \frac{size_{old} - size_{new}}{size_{unoptimized}}$$

The formulas for calculating the rewards are presented above. The term $runtime_{unoptimized}$ ($size_{unoptimized}$) refers to the runtime (code size) of the program compiled using -O0. In FlexPO, the rewards are normalized with $runtime_{unoptimized}$ and $size_{unoptimized}$, enabling the search of programs of various scales (e.g., those executing in milliseconds versus minutes) with consistent parameters.

## 5 Experiment

In this section, we attempt to address the following questions through our experiments:

- What is the accuracy of the activation predictor?
- Can FlexPO discover solutions that outperform manually designed optimization sequences (e.g., -Oz, -O3)?
- Is conservative pruning more effective than aggressive pruning?
- What benefits does the activation predictor provide in the context of FlexPO?
- How sensitive are the results to changes in hyperparameters?

### 5.1 Experimental Setup

**5.1.1 Implementation.** We implement FlexPO based on CompilerGym [15], a toolkit designed for applying RL to compiler optimization tasks. CompilerGym uses LLVM-10 for evaluation. Although LLVM-10 is not the latest version, the optimization transformations between LLVM-10 and the most recent LLVM 17 are mostly identical. There are 124 transformations in FlexPO's search space, which are listed in [31]. FlexPO employs the default settings for all transformation parameters (e.g., pragma-unroll-threshold=16 * 1024, max-uses-for-sinking=30), aligning with the methodologies of other phase-ordering solutions based on the LLVM project [3, 22]. Additionally, we use AutoPhase [18] from CompilerGym to extract static features from LLVM IR.

**5.1.2 Benchmark.** The Ctuning CBench benchmark [16] is utilized for evaluation. It encompasses a wide range of applications spanning automotive, security, office, and telecommunications domains. For all runtime results, we evaluate the same program five times, and manually analyze the results to make sure the variance and the measurement error are small enough to be ignored. For code size, we measure the size of the code sections in the generated binary files.

FlexPO is evaluated to find the optimal sequences for x86 architectures. We compile and execute the transformed LLVM IRs on an 11th Gen Intel Core i7-11700 CPU backend to obtain the code size and runtime performance.

### 5.2 Activation Predictor

In this section, we verify the feasibility of using the dormant history to predict the active status of subsequent transformations. To accomplish this, we generate a training dataset using the `qsort` program, and validation datasets using the `dijkstra` and `sha` programs. Employing different programs for training and validation allows us to demonstrate the generalization capability of the activation predictor. While a leave-one-out cross-validation approach for all benchmarks would be more convincing, it demands considerable time for dataset generation and predictor training. Given that our primary focus is on searching for optimal sequences rather than solely predicting active status, we allocate the majority of computational resources to the phase-ordering search process (Section 5.3 to 5.5). Furthermore, the success of the search process serves as additional evidence supporting the validation of the activation predictor.

To construct the datasets, we randomly generate transformation sequences of 1000 transformations in length. These sequences are applied to programs, and we record the active statuses to construct the dataset. Each entry in the dataset consists of a tuple with three elements: the dormant history, the transformation under prediction, and the ground truth label indicating the active status of the transformation. The process of constructing the dataset is depicted in Figure 7.

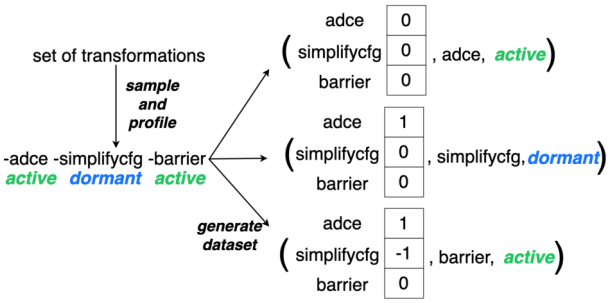



**Figure 7.** The process of constructing datasets.

We construct a training dataset comprising 921,000 tuples, and validation datasets consisting of 392,000 tuples from the dijkstra program and 248,000 tuples from the sha program, respectively. The activation predictor is trained using the Adam optimizer [24] with a learning rate of 1e-4. The learning curve, illustrated in Figure 8, demonstrates consistent behavior between the training and validation datasets. This consistency indicates that the activation predictor possesses strong generalization capabilities across different programs.

Moreover, we also present the confusion matrix in Table 1. We observe that 78% of the samples in the dijkstra dataset and 76% in the sha dataset are negative, indicating that most transformations are dormant. This suggests that conservative pruning has the potential to prune a significant portion of the search space.

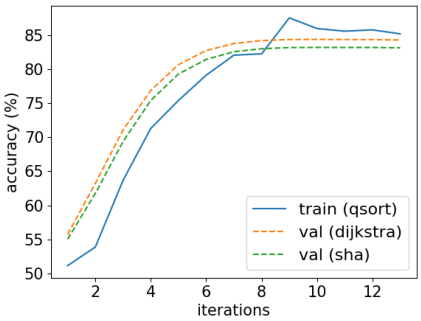



**Figure 8.** The learning curves for training the activation predictor.

**Table 1.** The confusion matrix for two validation datasets.

| Programs | True Positive | True Negative | False Positive | False Negative |
|---|---|---|---|---|
| Dijkstra | 57693 | 272344 | 35115 | 26848 |
| SHA | 37516 | 168456 | 19813 | 22215 |

### 5.3 FlexPO vs Oz and O3

LLVM offers a set of optimization levels, each defining a fixed sequence of transformations. By applying these sequences, programs are expected to achieve improved results. Among these levels, O3 and Oz are the most commonly used. The O3 optimization level is designed to enhance runtime performance, while Oz focuses on reducing code size.

In the following sections, FlexPO is utilized to find the sequences for either runtime or code size optimization. FlexPO integrates the pretrained activation predictor (Section 5.2) and initializes the rest of the model with random weights.

Generally, there is a high variance in the optimal sequences identified by FlexPO across different programs, with no common patterns emerging from these sequences. This variability underscores the critical importance of conducting searches for optimal sequences that are specifically tailored to individual programs.

**5.3.1 Runtime Optimization.** In this section, FlexPO is utilized to search for sequences that achieve the shortest runtime. To limit the search space, FlexPO typically terminates an episode after applying 80 transformations (iteration=80) for most programs. However, for bitcount, patricia, qsort, and tiff2rgba programs, we find that applying more iterations can significantly decrease runtime. Consequently, these programs undergo 350 iterations per episode. FlexPO conducts the search process over 20 episodes. All these hyperparameters are set based on empirical observations.

The results are presented in Figure 9. In 15 applications, FlexPO successfully finds sequences that outperform O3 in 13 of them. Specifically, for the stringsearch and bitcount

programs, FlexPO identifies sequences that achieve 10x and 7x higher performance, respectively. On average, excluding stringsearch and bitcount, programs compiled with FlexPO are 12% faster than those compiled with O3.

The programs evaluated for runtime optimization represent a subset of those discussed in Section 5.3.2, which focused on code size optimization. This discrepancy arises due to the inability to execute certain programs, attributable to issues such as input and runtime errors, which hinder the measurement of their runtime performance.

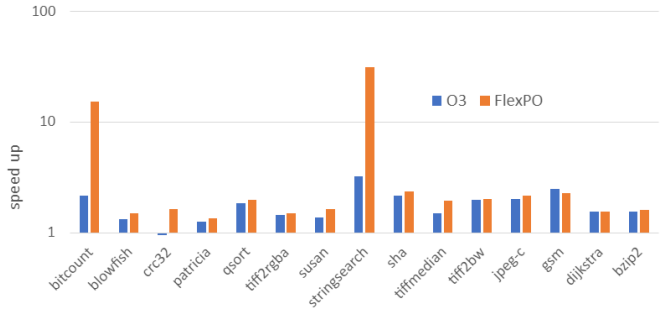
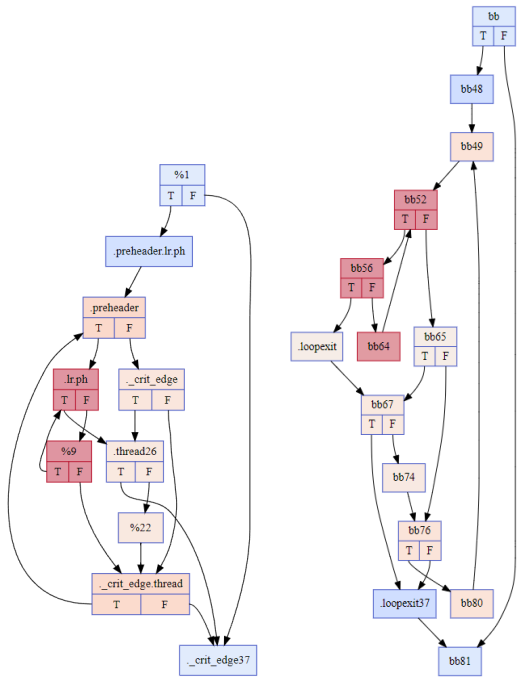



**Figure 9.** The speed up of programs compiled by O3 and FlexPO, compared with unoptimized programs.

```
char *strsearch(const char *string) {
  register size_t shift;
  register size_t pos = len - 1;
  size_t limit=strlen(string);
  char *here;
  while (pos < limit) {
    while( pos < limit && (shift = table[(unsigned
      char)string[pos]]) > 0)
      pos += shift;
    if (0 == shift) {
      if (0 == strncmp(findme,here=(char *)&string
     [pos-len+1],len))
        return here;
      else  pos++;
    }
  }
  return NULL;
}
```

**Listing 2.** The major function for the stringsearch program.

The stringsearch benchmark serves as a case study to demonstrate the superiority of FlexPO over the O3 pipeline. The function that dominates 99% of the execution time is shown in Listing 2. This function features a complex control flow, with its performance critically dependent on branch instructions. The Control Flow Graphs (CFGs) of the programs, processed by both O3 and FlexPO, are compared in Figure 10. FlexPO generates a CFG with fewer branch instructions. As observed in Listing 2, both the outer and inner loops share the same condition ('pos < limit'). This similarity presents an opportunity for more aggressive branch elimination. FlexPO identified a sequence that generates a CFG with 6 conditional branches, as opposed to the CFG with 7 conditional branches created by O3. Performance metrics detailed in Table 2 reveal that the program optimized by FlexPO executes fewer branch and overall instructions.




**(a)** CFG after O3. **(b)** CFG after FlexPO.

**Figure 10.** FlexPO finds a transformation sequence that has fewer branch instructions been executed, compared with O3.

**Table 2.** The profiling data for the stringsearch program.

| metric | unoptimized | O3 | FlexPO |
|---|---|---|---|
| runtime (sec) | 12.231 | 3.692 | 0.104 |
| # of instructions | 8.1*1e10 | 3.6*1e10 | 1*1e9 |
| # of branches | 1.5*1e10 | 1.0*1e10 | 4.4*1e8 |
| # of branch-misses | 5.4*1e7 | 1.2*1e7 | 4.5*1e5 |

### 5.3.2 Code Size Optimization.

FlexPO is also employed to achieve the smallest code size, measured by the number of bytes in the text segment. We set the number of *episodes* to 15 and *iterations* to 150. The results are presented in Figure 11. On average, FlexPO is capable of generating programs with text segments that are 17% smaller compared to those compiled with Oz.

## 5.4 Aggressive vs. Conservative Pruning

This section aims to compare the differences between utilizing predefined sequences (aggressive pruning) and individual transformations (conservative pruning) in the search process. We use the sequences proposed in MiCOMP [3] and POSET-RL [22] as representatives of aggressive pruning. MiCOMP categorizes the transformations within the O3 pipeline into five sequences, while POSET-RL formulates 15 sequences by splitting the Oz pipeline.

To ensure comparability, the same RL model and search algorithm are employed for both pruning strategies. The

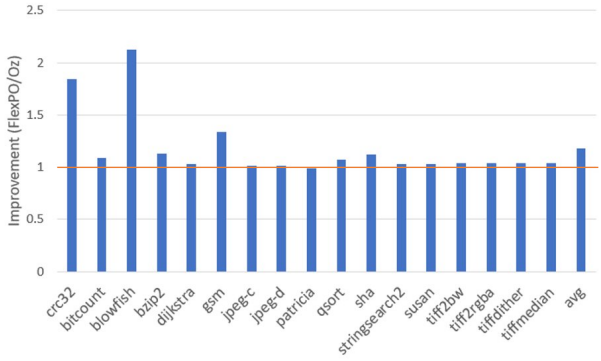



**Figure 11.** The code size improvement. The value is computed as $\frac{Oz\ code\ size}{FlexPO\ code\ size}$.

primary distinction lies in the agent's choice of action: it selects sequences of transformations for aggressive pruning and individual transformations for conservative pruning.

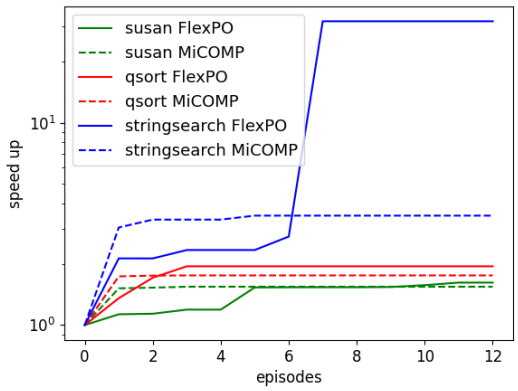
To ensure a fair comparison, all search episodes are terminated after applying 80 transformations. MiCOMP employs five sequences with lengths of 36, 2, 3, 1, and 4. To form a sequence of length 80, approximately $3 * 10^{22}$ different combinations are possible, representing the size of the search space. In the case of POSET-RL, which includes 15 sequences, the search space is approximately $9 * 10^{16}$. For FlexPO, the search space expands to $124^{80}$, significantly larger than both MiCOMP and POSET-RL.

In Figure 12a, we visualize the search process for runtime improvement. The figure illustrates the speed-up compared to unoptimized programs. The sequences proposed by MiCOMP, derived from the O3 pipeline, incorporate prior knowledge from human experts. Consequently, searches based on these sequences can identify effective solutions within a few episodes. Conversely, FlexPO explores a larger search space and requires more episodes to converge to an optimal solution. Nonetheless, the solutions obtained by FlexPO consistently outperform those from MiCOMP. A similar conclusion is drawn when comparing FlexPO with POSET-RL for code size improvement, as shown in Figure 12b.

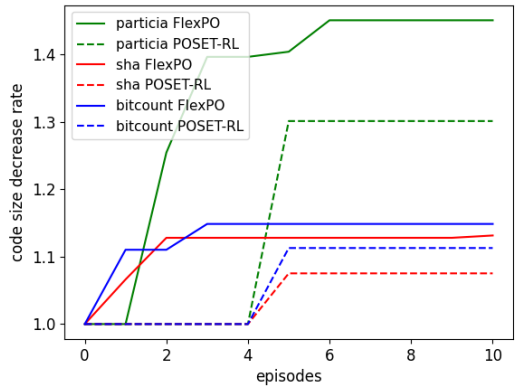
The evaluations indicate that FlexPO (conservative pruning) can produce superior solutions, albeit with an increased search time. In contrast, aggressive pruning relies on predefined sequences derived from human expertise, which tends to yield suboptimal solutions more quickly.

### 5.5 Validation of the Activation Predictor

The activation predictor is utilized to bypass dormant transformations, thereby potentially facilitating the search of optimal solutions with fewer search iterations. This section compares the efficacy of searching with and without the assistance of the activation predictor.




**(a)** FlexPO vs MiCOMP for runtime improvement.




**(b)** FlexPO vs POSET-RL for code size improvement.

**Figure 12.** Comparison of FlexPO with aggressive pruning.

We execute FlexPO for 10 episodes, with each episode consisting of only 10 iterations. The search process is visualized in Figure 13 for three applications: susan, qsort, and stringsearch. Without the activation predictor, the highest improvements are close to the unoptimized programs. This indicates that, in the absence of the activation predictor, the transformations explored by FlexPO do not significantly change the programs. Conversely, with the activation predictor, FlexPO avoids exploring these dormant transformations, enabling significant runtime improvement by applying only 10 transformations.

### 5.6 Cost for Search

The search time depends on the following four factors: the number of episodes $E$, the number of iterations $L$, the time to get reward information $R$, and the process $T$ for updating the RL model. The amount of time can be roughly calculated as $E*L*R+E*T$. Since FlexPO uses lightweight DNNs, $T$ is much smaller than $R$. Thus, the amount of time can be regarded as $E * L * R$. For code size improvement, $R$ is the time to compile

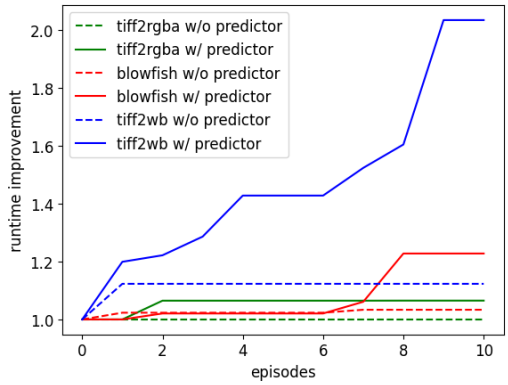
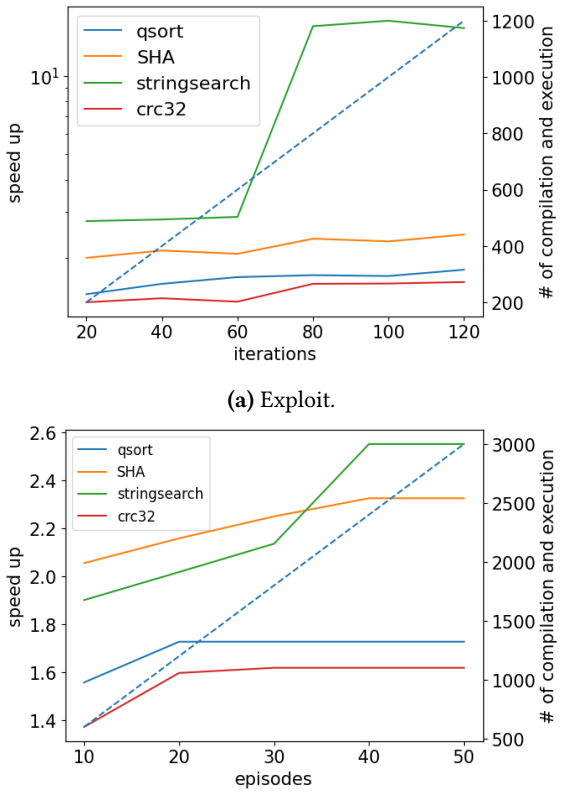



**Figure 13.** The search process with/without the activation predictor with only 10 iterations.

the program, while for runtime improvement, $R$ includes the execute time additionally. Both $E$ and $L$ are hyperparameters. $E$ is the number of times to search from the unoptimized IRs (explore), while $L$ is the depth for each search (exploit). For FlexPO, each iteration selects a transformation. Thus, $L$ is also the maximum length of the transformation sequence.

To investigate the relationship between search time and search outcomes, as well as the sensitivity of the search results to the number of episodes and iterations, we conduct two experiments. Firstly, we set the number of episodes $E$ to 10 and vary the number of iterations $L$. Secondly, we fix the number of iterations $L$ at 60 and vary the number of episodes $E$. The results of these experiments are presented in Figure 14. In general, FlexPO finds better solutions with larger numbers of iterations or episodes. However, the overhead ($E * L$) increases linearly with the number of episodes or iterations. Thus, these hyperparameters serve as adjustable knobs for users to balance the trade-off between search time and output quality.

In our evaluation, we find that setting $E = 20$ and $L = 80$ is sufficient to discover solutions that outperform O3 for most applications. The evaluations in Section 5.3 employ this configuration and successfully identify sequences that surpass O3. The dijkstra program, having the longest execution time (the largest $R$), consequently requires the longest search time. With $E = 20$ and $L = 80$, the search takes approximately 66 minutes. However, as discussed in Section 4, some IRs appear repeatedly during the search. FlexPO only executes them upon their first occurrence. Therefore, in practical situations, the actual search time is significantly shorter, approximately 15 minutes in our evaluations.




**(a)** Exploit.

**(b)** Explore.

**Figure 14.** Generally, FlexPO identifies better solutions with a greater number of episodes (exploration) or iterations (exploitation). However, the cost of the search increases linearly with the number of episodes or iterations.

## 6 Related work

### 6.1 History of Phase-Ordering Problem

In the early days, researchers utilized *predictive heuristics* to directly find an optimal transformation sequence without the need for multiple compilations of a program. However, due to the rapidly increasing number of transformations [5] and the unpredictable interactions between them [37], these methods often resulted in suboptimal outcomes [33]. In more recent times, most researchers [1, 4, 8, 9] have shifted to *iterative compilation*. This approach involves compiling programs with various transformation sequences, evaluating the generated code, and selecting the best performer. Compared to predictive heuristics, iterative compilation bases its evaluations on the actual generated code, leading to more effective solutions [33].

Recently, Machine Learning has become a popular choice. Some researchers use Genetic Algorithms (GA) to search the optimal sequences [10, 11, 25]. These methods generate a number of *genes* (transformation sequences) and compile the programs accordingly to evaluate the fitness value for each gene and select the best ones. These methods cannot utilize the information about programs and always require a long time to get good solutions.

Some researchers propose solutions to utilize the programs' information during the search. Agakov et al. [1] form several classical programs into a dataset and use either the

Markov model or the Independent Identically Distributed model to learn the distribution of which transformations work well for each program in the dataset. For a new program, the algorithm captures the program's features and finds which program $p$ in the dataset has the closest feature. Then, use the distribution learned by $p$ to sample a sequence as the output. Martins et al. [30] use a similar method to implement the search process with GA. Their method uses static features such as whether the loops have calls and the number of instructions in loops. The static features can be captured directly from the programs, without the need to execute. These features are easily captured but contain limited information. Cavazos et al. [9] utilize dynamic features (e.g., cache miss, branch miss) to build the model. These dynamic features depend on special hardware, which cannot be migrated to other architectures. Some projects [2, 4] use microarchitecture-independent workload characterization [21] to make the trained model portable to different hardware. Instead of relying on profiling tools to analyze the programs, some researchers use Deep Neural Networks to parse source code. Cummins et al. [13] use a Recurrent Neural Network to map source code into a fixed-length vector. This method regards code as strings, which do not capture the topology information. Neural Code Comprehension [7] proposes a contextual flow graph to capture dataflow information as well. ProGraML [12] further captures the control flow information. The outputs of ProGraML are graphs, which need to be trained with Graph Neural Networks. IR2Vec [36] maps LLVM IRs to vectors.

ML is also used for the search process. [28] trains a Neural Network, which accepts the feature vectors for the programs and generates the selected transformations. [2] uses ML models to predict the speed up for a given transformation sequence; [35] uses ML models to predict the performance by using the hardware features as inputs. Both methods can avoid the overhead to execute programs on real hardware. MiCOMP [3] uses the recommendation system knowledge to explore the search space to avoid the local minima. [14] proposes to directly pass the LLVM IR to the Large Language Model and get the optimal sequences to reduce the code size.

As an important area in Machine Learning, Reinforcement Learning also has been used for solving the phase-ordering problem. Mammadli et al. [29] utilize Deep Reinforcement Learning to solve the phase-ordering problem. Their framework supports searching on different granularities: sequences of transformations (coarse-grained), transformations with default arguments (fine-grained), and transformations and their arguments (finer-grained). However, as reported by the authors, the framework can hardly find sequences that significantly surpass O3. POSET-RL [22] uses RL to search for transformation sequences to reduce the code size. They implement the search process with the aggressive pruning; they cluster the transformations in the Oz pipeline into 15 or 34 sequences and search on them.

Compared with these related works [3, 22], FlexPO searches on transformations instead of sequences. Thus, FlexPO has a larger search space. Although the framework in [29] also searches on transformations, the framework finds sequences that surpass the LLVM O3 pipeline only around 3% in a reasonable time. This is due to it doesn't apply pruning during searches. Instead, FlexPO applies conservative pruning and finds sequences 12% better than the LLVM O3 pipeline.

### 6.2 Dormant Transformation

The insight of conservative pruning is that some transformations are dormant. There are other researchers who also utilize the dormant information. Some researchers [25, 27] manually collect the rules about which transformations are dormant after applying given transformations. For example, if *register allocation* has been applied, the *register allocation* will become dormant until any transformations that change the register pressure are applied. They integrate these rules into the Genetic Algorithm, which prunes the sequences that contain dormant transformations and makes GA coverage with fewer generations. These rules are collected manually, which is impractical when there are a large number of transformations. Thus, they only evaluate a search space that contains 15 transformations. In contrast, FlexPO implements an activation predictor to learn these rules, enabling our solution to efficiently handle large search spaces without reliance on prior knowledge.

[23] trains a predictor to predict which transformations in the LLVM O3 pipeline are dormant for a given program. By skipping these transformations, the compiler can save compilation time. [19] records dormant transformations during the build process to speed up incremental builds by skipping these dormant transformations.

## 7 Conclusion

The phase-ordering problem is challenging due to the large search space. Existing solutions rely on prior knowledge from human experts to aggressively prune the search space, which may exclude optimal solutions and is not scalable to new transformations.

In this paper, we propose conservative pruning, which ensures that the optimal solutions remain within the search space during the pruning process. The insight behind conservative pruning is utilizing the dormant history to predict the dormant status for subsequent transformations, to guide the search process focusing on transformations that are likely to be activated. Conservative pruning does not rely on human expertise and is scalable to new transformations. We introduce FlexPO, a toolkit that integrates conservative pruning with a RL model to solve the phase-ordering problem. Our experimental results demonstrate that FlexPO generates programs that are 12% faster than those optimized with O3 and 17.6% smaller than those optimized with Oz on average.

## References


[1] Felix Agakov, Edwin Bonilla, John Cavazos, Björn Franke, Grigori Fursin, Michael FP O'Boyle, John Thomson, Marc Toussaint, and Christopher KI Williams. 2006. Using machine learning to focus iterative optimization. In *International Symposium on Code Generation and Optimization (CGO'06)*. IEEE, 11–pp.

[2] Amir Hossein Ashouri, Andrea Bignoli, Gianluca Palermo, and Cristina Silvano. 2016. Predictive modeling methodology for compiler phase-ordering. In *Proceedings of the 7th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and the 5th Workshop on Design Tools and Architectures For Multicore Embedded Computing Platforms*. 7–12.

[3] Amir H Ashouri, Andrea Bignoli, Gianluca Palermo, Cristina Silvano, Sameer Kulkarni, and John Cavazos. 2017. Micomp: Mitigating the compiler phase-ordering problem using optimization sub-sequences and machine learning. *ACM Transactions on Architecture and Code Optimization (TACO)* 14, 3 (2017), 1–28.

[4] Amir Hossein Ashouri, Giovanni Mariani, Gianluca Palermo, Eunjung Park, John Cavazos, and Cristina Silvano. 2016. Cobayn: Compiler autotuning framework using bayesian networks. *ACM Transactions on Architecture and Code Optimization (TACO)* 13, 2 (2016), 1–25.

[5] David F Bacon, Susan L Graham, and Oliver J Sharp. 1994. Compiler transformations for high-performance computing. *ACM Computing Surveys (CSUR)* 26, 4 (1994), 345–420.

[6] Hangbo Bao, Li Dong, and Furu Wei. 2021. Beit: Bert pre-training of image transformers. *arXiv preprint arXiv:2106.08254* (2021).

[7] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. 2018. Neural code comprehension: A learnable representation of code semantics. *Advances in Neural Information Processing Systems* 31 (2018).

[8] 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*.

[9] John Cavazos, Grigori Fursin, Felix Agakov, Edwin Bonilla, Michael FP O'Boyle, and Olivier Temam. 2007. Rapidly selecting good compiler optimizations using performance counters. In *International Symposium on Code Generation and Optimization (CGO'07)*. IEEE, 185–197.

[10] Keith D Cooper, Philip J Schielke, and Devika Subramanian. 1999. Optimizing for reduced code space using genetic algorithms. In *Proceedings of the ACM SIGPLAN 1999 workshop on Languages, compilers, and tools for embedded systems*. 1–9.

[11] Keith D Cooper, Devika Subramanian, and Linda Torczon. 2002. Adaptive optimizing compilers for the 21st century. *The Journal of Supercomputing* 23, 1 (2002), 7–22.

[12] Chris Cummins, Zacharias V Fisches, Tal Ben-Nun, Torsten Hoefler, and Hugh Leather. 2020. Programl: Graph-based deep learning for program optimization and analysis. *arXiv preprint arXiv:2003.10536* (2020).

[13] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. 2017. End-to-end deep learning of optimization heuristics. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 219–232.

[14] Chris Cummins, Volker Seeker, Dejan Grubisic, Mostafa Elhoushi, Youwei Liang, Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Kim Hazelwood, Gabriel Synnaeve, et al. 2023. Large language models for compiler optimization. *arXiv preprint arXiv:2309.07062* (2023).

[15] Chris Cummins, Bram Wasti, Jiadong Guo, Brandon Cui, Jason Ansel, Sahir Gomez, Somya Jain, Jia Liu, Olivier Teytaud, Benoit Steiner, et al. 2022. CompilerGym: robust, performant compiler optimization environments for AI research. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 92–105.

[16] Grigori Fursin and Olivier Temam. 2010. Collective optimization: A practical collaborative approach. *ACM Transactions on Architecture and Code Optimization (TACO)* 7, 4 (2010), 1–29.

[17] Zhangxiaowen Gong, Zhi Chen, Justin Szaday, David Wong, Zehra Sura, Neftali Watkinson, Saeed Maleki, David Padua, Alexander Veidenbaum, Alexandru Nicolau, et al. 2018. An empirical study of the effect of source-level loop transformations on compiler stability. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–29.

[18] Ameer Haj-Ali, Qijing Jenny Huang, John Xiang, William Moses, Krste Asanovic, John Wawrzynek, and Ion Stoica. 2020. Autophase: Juggling hls phase orderings in random forests with deep reinforcement learning. *Proceedings of Machine Learning and Systems* 2 (2020), 70–81.

[19] Ruobing Han, Jisheng Zhao, and Hyesoon Kim. 2024. Enabling Fine-Grained Incremental Builds By Making Compiler Stateful. In *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE.

[20] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[21] Kenneth Hoste and Lieven Eeckhout. 2007. Microarchitecture-independent workload characterization. *IEEE micro* 27, 3 (2007), 63–72.

[22] 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)*. IEEE, 121–131.

[23] Tarindu Jayatilaka, Hideto Ueno, Giorgis Georgakoudis, EunJung Park, and Johannes Doerfert. 2021. Towards compile-time-reducing compiler optimization selection via machine learning. In *50th International Conference on Parallel Processing Workshop*. 1–6.

[24] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[25] Prasad Kulkarni, Stephen Hines, Jason Hiser, David Whalley, Jack Davidson, and Douglas Jones. 2004. Fast searches for effective optimization phase sequences. *ACM SIGPLAN Notices* 39, 6 (2004), 171–182.

[26] Prasad Kulkarni, Wankang Zhao, Hwashin Moon, Kyunghwan Cho, David Whalley, Jack Davidson, Mark Bailey, Yunheung Paek, and Kyle Gallivan. 2003. Finding effective optimization phase sequences. *ACM SIGPLAN Notices* 38, 7 (2003), 12–23.

[27] Prasad A Kulkarni, David B Whalley, Gary S Tyson, and Jack W Davidson. 2009. Practical exhaustive optimization phase order exploration and evaluation. *ACM Transactions on Architecture and Code Optimization (TACO)* 6, 1 (2009), 1–36.

[28] 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.

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

[30] Luiz GA Martins, Ricardo Nobre, Joao MP Cardoso, Alexandre CB Delbem, and Eduardo Marques. 2016. Clustering-based selection for the exploration of compiler optimization sequences. *ACM Transactions on Architecture and Code Optimization (TACO)* 13, 1 (2016), 1–28.

[31] Facebook research. 2020. Available transformations in CompilerGym. https://compilergym.com/llvm/index.html.

[32] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347* (2017).

[33] Spyridon Triantafyllis, Manish Vachharajani, Neil Vachharajani, and David I August. 2003. Compiler optimization-space exploration. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.* IEEE, 204–215.

[34] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[35] Kapil Vaswani, Matthew J Thazhuthaveetil, YN Srikant, and PJ Joseph. 2007. Microarchitecture sensitive empirical models for compiler optimizations. In *International Symposium on Code Generation and Optimization (CGO'07)*. IEEE, 131–143.

[36] S VenkataKeerthy, Rohit Aggarwal, Shalini Jain, Maunendra Sankar Desarkar, Ramakrishna Upadrasta, and YN Srikant. 2020. Ir2vec: Llvm ir based scalable program embeddings. *ACM Transactions on Architecture and Code Optimization (TACO)* 17, 4 (2020), 1–27.

[37] Deborah L Whitfield and Mary Lou Soffa. 1997. An approach for exploring code improving transformations. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 19, 6 (1997), 1053–1084.

Received 13-NOV-2023; accepted 2023-12-23