DeCOS: Data-Efficient Reinforcement Learning for Compiler Optimization Selection Ignited by LLM

Tianming Cui
University of Minnesota
Minneapolis, MN, USA
cuixx327@umn.edu

Pen-Chung Yew
University of Minnesota
Minneapolis, MN, USA
yew@umn.edu

Stephen McCamant
University of Minnesota
Minneapolis, MN, USA
smccaman@umn.edu

Antonia Zhai
University of Minnesota
Minneapolis, MN, USA
zhai@umn.edu

Abstract

Machine learning methods have proven their effectiveness in a wide range of program optimization tasks. These methods selectively map program feature spaces to carefully defined optimization spaces to identify effective optimizations. However, the size and complexity of these spaces often necessitate large amounts of training data to achieve effective mappings. For certain optimization tasks, obtaining accurate training data can be costly, making data efficiency a critical concern. Reinforcement learning (RL) offers a promising solution by dynamically adjusting exploration strategies and selectively requesting training data. In this paper, we propose leveraging reinforcement learning to optimize compilation sequences.

This paper presents the Data-efficient Compiler Optimization Selection (DeCOS) system, which utilizes a reinforcement learning engine to perform a guided search of the optimization spaces. To improve the data efficiency in training DeCOS, we utilize synthesized data to configure the RL-architecture; and incorporate simulation results to refine profiling information. To overcome the slow start-up issue in RL-processes, we integrate an LLM into the workflow, leveraging its knowledge to accelerate the initial training phase of the RL-agent. Our experiments show that DeCOS efficiently generates compiler optimization sequences that either match or outperform that of the state-of-the-art optimizer `Opentuner`. Furthermore, the DeCOS reinforcement learning engine, once trained, demonstrates its versatility by showing portability across different target applications and hardware platforms, highlighting its broad applicability and adaptability.

CCS Concepts

• **Software and its engineering** → **Compilers**; • **Theory of computation** → *Reinforcement learning*; • **Information systems** → *Language models.*

Keywords

Compilers, Reinforcement Learning, Language models

ACM Reference Format:
Tianming Cui, Pen-Chung Yew, Stephen McCamant, and Antonia Zhai. 2025. DeCOS: Data-Efficient Reinforcement Learning for Compiler Optimization Selection Ignited by LLM. In *2025 International Conference on Supercomputing (ICS '25), June 08–11, 2025, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3721145.3725765

1 Introduction

Modern compilers perform the complex task of mapping programs written in high-level programming languages onto the underlying hardware and optimizing the program along the way. The compiler must select a set of optimizations and apply them in a specific sequence to achieve the desired performance goal [6, 33]. Achieving the optimal optimization options not only requires analyzing program characteristics and determining how to take advantage of the underlying hardware, but also requires taking into consideration the constructive or destructive interaction among optimizations. The permutation of optimization passes creates a large search space that is highly irregular and difficult to search [31].

ICS '25, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-1537-2/25/06
https://doi.org/10.1145/3721145.3725765

Programmers, limited by time and resources, often use pre-selected sets of optimizations, such as `-O2` and `-O3`, thus being unable to achieve the full potential of the compiler's optimization infrastructure.

Some recent works [2, 12, 17, 34] have made significant progress in exploring the compiler optimization sequence space and efficiently identifying optimal sequences. These techniques are tailored to specific classes of applications, because they are either specifically trained for a single task or relying on application-specific techniques [2, 34]. Thus, their applicability is limited and cannot be generalized to other applications.

Existing works targeting general applications can be classified into two main categories: search-based algorithms [3, 9, 20, 30] and supervised-learning-based [12] algorithms. While these methods have achieved success in certain scenarios, they also face inherent limitations.

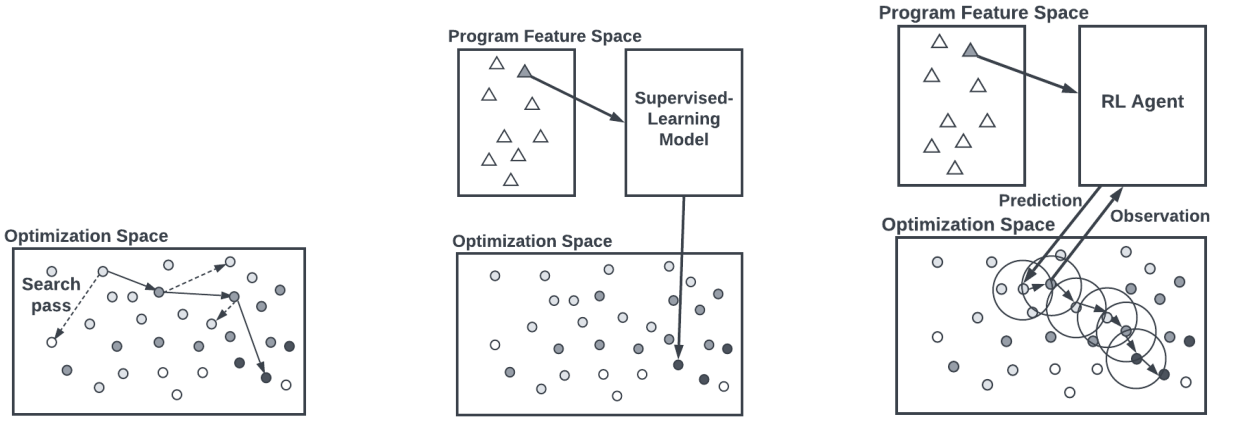
Search-based algorithms, such as hill climbing [3] and genetic algorithms (GAs) [9, 20, 30], search for the optimal compiler optimization sequence by exploring the space defined by all possible compilation sequences, as shown in Figure 1(a). For efficient searches, these algorithms explore the search space either along the direction of gradual descent or by introducing mutations. However, the greedy nature of hill climbing often leads to local optimum, while the undirected mutations in genetic algorithms demand massive generation and filtering, leading to significant computational overhead. Moreover, the search conducted by these algorithms is tailored to a specific application and hardware environment, necessitating re-initialization for every new task. In other words, the knowledge gained during the exploration of one program cannot be retained or transferred to subsequent explorations, resulting in no cumulative learning from past experiences.

Supervised learning-based approaches, alternatively, can learn the impact of various optimizations from previous experience. When presented with a new program, they produce an optimization sequence based on the learned mapping, as shown in Figure 1(b). In supervised learning-based approaches, the model maps code features to predicted optimization sequences. The program feature space and the optimization space are both high dimensional. For example, the LLVM Optimizer `opt` [21] uses 85 optimization options in its `-O3`. Given that the same option can recur multiple times and that the order in which options are applied matters, the number of potential optimization sequences for a sequence of 16 distinct options is an astronomical 85^{16}. Thus, training a supervised learning model to perform a precise mapping is exceptionally challenging. The high cost of obtaining training data in this problem exacerbates the problem, as each program must be compiled and profiled thousands of times with different optimization options, yet only a small subset of sequences yielding optimal results is used to train the model. Thus, previous works that attempted to employ supervised learning algorithms for optimization set selection, such as MilepostGCC [12], struggle to obtain sufficient training data.

In conclusion, there remains a pressing need for a machine learning tool that can be effectively and efficiently applied to any target program, delivering substantial performance improvements [11]. The timeliness requirement limits the feasibility of large-scale exploration of the search space, and thus a guided search, based on program characteristics and prior compilation experience is preferred. A machine learning model that incorporates target program characteristics and the effects of compiler optimization passes can serve as the foundation for a more effective and efficient search process.

Reinforcement Learning (RL), often referred to as *interactive* machine learning, in which the RL-agent learns the optimal actions by interactively exploring an environment and collecting observations from the actions [29]. In other words, RL is able to efficiently explore a search space while learning, making it an ideal candidate for addressing the requirements of compiler optimization selection. Figure 1(c) illustrates an RL-agent searching through the optimization space of a given application. The RL-agent makes optimization decisions using a combined observation that includes both program features and the current position within the optimization space. The RL-agent predicts an optimization step, illustrated as an arrow, within the optimization space. Compared to Supervised Learning (SL) models, which map program features to a point in the entire optimization space in one single step, the RL-agent breaks this complex task into a sequence of smaller sub-tasks. In each sub-task, the RL-agent's mapping target is a much smaller sub-space (illustrated as circles in Figure 1(c)). This approach significantly reduces the complexity of the mapping the RL-model needs to learn. In addition, taking small steps in the search space enables the RL-agent to benefit from additional information observed in each step. Moreover, the RL-agent demonstrates higher data efficiency compared to SL models during training. While SL models rely solely on pre-collected training data, RL-agents learn iteratively by collecting observations from the optimization space. These observations allow the RL-agent to continuously refine its mapping strategy, enabling it to make improved decisions over time and self-train on data with improved efficiency.

However, applying RL algorithms to the optimization task presents several challenges. To develop a practical RL-based optimization approach, the following issues must be addressed:



(a) Search-based algorithms explore the entire optimization space.

(b) Supervised-Learning Algorithms create a mapping from the program features space to the optimization space.

(c) DeCOS conducts a guided search through the optimization space using an RL-agent that maps observations to an exploration step. The RL-agent dynamically updates its strategy in the search process.

Figure 1: Different strategies for compiler optimization sequence search.

Selecting hyper-parameters for the RL-model: Hyper-parameters define the architecture of the RL-model, including the number of layers, learning rate, etc. Hyper-parameter tuning can be a labor-intensive process and incur substantial costs. Therefore, an efficient mechanism for hyper-parameter tuning is essential.

Observing efficient information: RL-agents rely on observations from the search space to make informed decisions. To obtain adequate information about the current optimization state, the content of the observation must be carefully selected.

Accelerating slow startup: Although RL-agents can efficiently learn and adapt their strategies to the current task through accumulated experience, the initial exploration can still be inefficient. This often results in a slow and expensive startup phase.

Adjusting noisy data: RL-agents make decisions and adapt their strategies based on the outcomes of previous decisions, known as rewards. Earlier results significantly influence subsequent decisions, leading to cumulative impacts over time. Thus, RL-agents are highly sensitive and vulnerable to noise and errors in the observations and rewards. Minimizing noise in the collected data is critical to ensuring reliable learning.

In this paper, we introduce Data-efficient reinforcement learning for Compiler Optimization Selection (DeCOS) ignited by LLM, a reinforcement learning-based approach designed to efficiently navigate the optimization space of a target program by leveraging learned experience. Under this context, DeCOS makes the following contributions toward the development and implementation of a reinforcement learning framework for compiler optimization sequence selection:

(1) DeCOS introduces a training data synthesizer to synthesize training data with similar characteristics as realistic data for hyper-parameter tuning and reward function refinement. This enables the RL-agent to operate at optimized settings without requiring extensive and time-consuming preliminary experimentation on realistic data.
(2) DeCOS incorporates performance counter information into the observation of the optimization state, enhancing traditional code representation typically used for optimization. This integrated performance information provides insights for DeCOS to understand the reasons for changes in performance, and in turn enable more efficient decision-making.
(3) DeCOS addresses the challenge of slow startup in RL-systems by replacing time-consuming random explorations with decisions guided by Large Language Models (LLMs). The program analysis and reasoning capabilities of LLMs enable DeCOS to establish a reasonable starting point without incurring significant costs.
(4) DeCOS mitigates the noise inherent in performance data collected from real machines by refining profiling information using data from cache simulators. This

cleaner data enables DeCOS to avoid mishaps during space exploration.

2 Related Works

In this section, we will discuss prior works on producing compiler optimization sequences. We will also discuss related work in areas of code representation.

2.1 Prior Optimization Approaches

Search-based algorithms [3, 9, 20, 26, 30], such as Hill-climbing and Genetic Algorithms (GA), have been applied to explore the optimization space of programs as a classic solution. ACOVEA [20] is a GA-based compiler optimization selection tool that has been widely adopted by the programming community. Almagor et al. [3] demonstrate that hill-climbing algorithms can be applied to find high-performing optimization sequences. Although demonstrated successful in different contexts, these search-based algorithms are generally expensive for general-purpose optimization selection because they incur significant data collective overhead when applied to new applications. For example, ACOVEA requires 5.5 days to search the optimization space for a program that takes just one minute to compile and execute, highlighting the high computational cost of such methods.

On the other hand, Supervised learning (SL) algorithms [8, 12, 28], such as Milepost GCC [12], are built on prior compilation experience and aim to predict the optimal optimization sequence based on the program features, as shown in Figure 1(b). By avoiding exploring the optimization space, these algorithms can be more efficient than search-based algorithms on new applications. However, these SL-based algorithms suffer from the lack of exploring in the unseen space, and the limit of training data. As a result, supervised learning approaches either cannot provide competitive optimization results compared to search-based approaches, or are only reserved for problems characterized by a relatively small optimization space such as determining optimal loop unroll factors [28].

2.2 Reinforcement Learning Algorithms

Reinforcement Learning [15, 23] algorithms combine searching and learning when presented with an optimization task, as shown in Figure 1(c). RL algorithms search the optimization space guided by an RL-agent, which can learn and adjust itself during the optimization process to better fit the current optimization task.

The key components of an RL-infrastructure can be summarized as the RL-agent, observation, prediction, and reward gain. RL-agent acts as the core of the RL-infrastructure, the RL-agent processes observations as input and generates predictions as output. Observations represent the information provided to the RL-agent about the current state of its environment. This input is critical for enabling the agent to make informed decisions about the next action to take. The prediction is the action decided by the RL-agent based on the given observation. This action reflects the RL-agent's current strategy and its understanding of how to maximum the reward gain in its current state. The reward gain evaluates the effectiveness of an action (or a series of actions). It can be provided immediately after each action or cumulatively after completing a sequence of actions. Higher rewards encourage the agent to replicate similar actions under similar conditions, while lower rewards discourage undesirable actions, guiding the agent toward more effective strategies. This feedback loop among observation, prediction and reward gain enables the RL-agent to iteratively improve its decision-making ability and its efficiency in exploring the environment over time.

As an instance, Autophase [17] successfully uses RL to solve options set selecting problems for HLS (High-Level Synthesis) designs. Autophase follows a similar idea to most of the SL approaches that statistical facts of a program should be the most important guidance when doing optimization, it only considers statistical features and already-used passes of the program when predicting the next optimization pass, with no additional information or feedback information provided for each step. In other words, the advantage of RL-infrastructure is not being fully exploited to approach high data efficiency. While showing impressive optimization results, Autophase is still not able to solve the general optimization task, as it still requires a large amount of data to be trained. Such data is only available under a special scenario of optimizing HLS designs where there is a fast method to estimate the performance without requiring time-consuming simulation [18]. There are also other RL-based optimization tools, such as CHAMELEON [2] and DYNATUNE [34], which focus on optimizing deep neural networks. However, most of these tools are only targeted at one program or one type of program, and cannot provide a general approach for all programs, due to their dependencies on necessary domain knowledge, to estimate the performance of the target program without execution to reduce the cost, etc. These reinforcement learning solutions emphasize the ability of RL-agents to explore the search space effectively for a single target program but overlook RL's learning ability to reproduce and update its learned strategy on unseen programs.

In summary, RL stands out as the preferred framework for program optimization tasks. However, existing RL approaches fall short of fully utilizing the inherent strengths of the RL workflow, particularly its adaptability to unseen tasks. This highlights the need for further advancements to develop a general-purpose program optimization tool.

2.3 Code Representation

Machine learning algorithms operate by learning and performing a mapping from input data to a predicted output. For these models to make accurate and efficient predictions, the input to the model that serves as a source of information must be represented effectively. In program optimization, an effective code representation that accurately captures the features of the given program is crucial for the success of machine learning algorithms in predicting potential optimizations. To enable efficient optimization, such representation must effectively convey the complex characteristics of the target application to the model. Many Machine-learning-based [1, 12, 17, 28] optimization tools, either SL-based or RL-based, rely on statistical analysis metrics such as instruction count, basic block count, and loop nest levels as their input. The statistical metrics, as a code representation, capture only a limited aspect of program features, as they fail to provide detailed insights into the logic and structure of the code. For instance, a program with a rarely used nested loop that has minimal impact on its performance might share similar statistical metrics to a program that spends most of its execution time in a computationally intensive nested loop. These two programs require different optimization strategies but cannot be effectively distinguished by an ML model that relies solely on statistical metrics as input.

Fortunately, with the widespread application of machine learning models which utilizes an array form of representation called embedding as inputs, significant advancements have been made in code embedding techniques. Novel embedders delicately designed for generating code embeddings, such as Code2vec [4] and IR2vec [32], have been developed, so that the generated code embeddings can better represent the features of the target programs. Research has shown that a good code embedding model can significantly improve the performance of downstream applications, such as program classification, etc. However, even these state-of-the-art embedding models cannot flawlessly address the optimization task. Unlike program classification, which relies solely on the program's static features, optimal optimizations often depend on the runtime characteristics of the application and the underlying hardware platform. The absence of this runtime information highlights a critical gap in existing code representations, suggesting opportunities for further improvements tailored to the needs of optimization tasks. This gap becomes even more pronounced in an RL-based optimizer, as runtime characteristics provide detailed, real-time feedback to the RL model, reflecting the impact of each decision it makes.

2.4 Large Language Models and Reinforcement Learning

Large Language Models (LLMs) are renowned for their capabilities in handling code-related tasks and have already been applied to program optimizations [13] [10]. To effectively integrate LLMs into complex, domain-specific tasks, fine-tuning becomes essential. Fine-tuning refers to the process of adapting a pre-trained model to perform a specific task by updating its parameters using task-specific data. This process ensures that the model's general knowledge can be refined to address the unique requirements of a particular task. The fine-tuning process can be broadly categorized into two main types: Supervised Fine-Tuning (SFT) and Reinforcement Fine-Tuning (RFT). Among these, RFT offers the advantages of requiring less data and adapting more quickly to unseen tasks. Current RFT applications, such as Reinforcement Learning with Human Feedback (RLHF) [19], have gained popularity for aligning LLM outputs with human preferences and enhancing task-specific performance. Potentially program optimization.

However, applying RL-based infrastructures to fine-tune LLMs faces a significant challenge: high computational costs. LLMs are deep neural networks, and each weight update is computationally intensive, requiring substantial hardware resources. This challenge becomes even more significant when the target task necessitates frequent model updates, such as adapting to the optimization process for individual target programs. Suggesting the importance of developing an alternative LLM-RL integration approach with lower overhead and better flexibility, on these special use cases.

3 DeCOS: Data-Efficient RL Infrastructure

In this section, we present DeCOS, a compiler optimization framework driven by an LLM-ignited reinforcement learning engine that searches the space defined by all optimization sequences, designed with a focus on high data efficiency.

As illustrated in Figure 2, the system takes C programs as input and compiles them into LLVM IRs [21] for the RL-driven optimization process. To explore the optimization space and formulate sequences of optimization options for the target codes, DeCOS operates through a nested loop structure. Each inner loop incrementally constructs a sequence of optimization options by iteratively adding single optimization options to the sequence. Such loops are repeated to identify an optimal sequence of optimization passes.

In the kth iteration of an inner optimization loop, the core component of DeCOS, the RL-agent, predicts the kth optimization option P_k to apply to the kth optimized LLVM IR based on the corresponding observation O_k as input. This predicted optimization option is then performed by `Clang opt` on the kth optimized LLVM IR, producing the $(k+1)$th

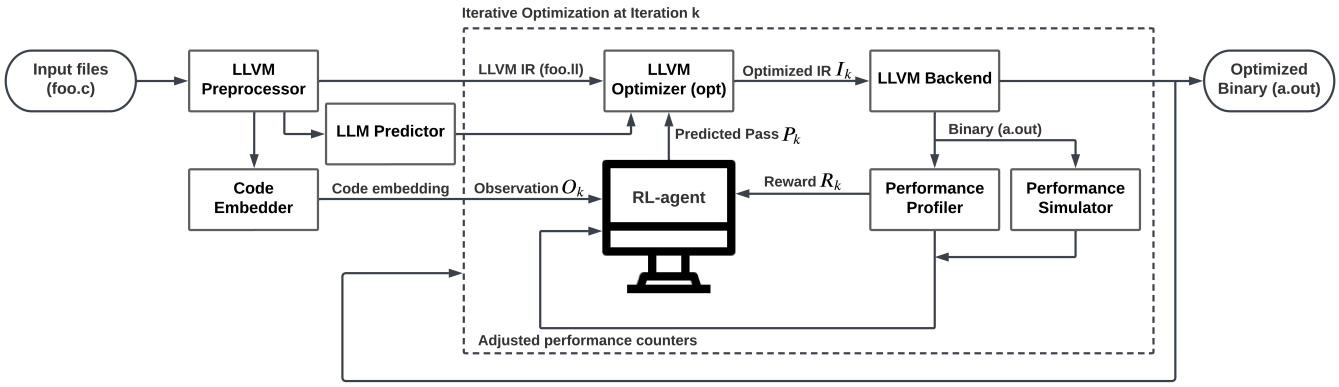


Figure 2: DeCOS Workflow: The RL-agent iteratively optimizes the LLVM IR of the target functions by applying one optimization option per iteration, while continuously updating its optimization strategy throughout the process.

optimized IR. The resulting IR is subsequently compiled into a binary file and profiled or simulated. Based on the performance of the optimized binary, a reward R_k is generated to evaluate the effectiveness of P_k. This reward is used to guide the RL-agent in refining its prediction strategy. Additionally, detailed performance counters profiled (or simulated) from the binary are combined with the code embedding of the IR to formulate the observation O_{k+1}, enabling the RL-agent to predict the next optimal pass. The inner loop continues until the maximum optimization options sequence length (denoted as N, set to 16 in our evaluation) is reached. At this point, the DeCOS model restarts the search using the original unoptimized LLVM IR, initiating a new loop. To balance the cost and frequency of model updates, DeCOS updates the strategy of its RL-agent after constructing every three completed optimization sequences. Consequently, we define three loops as one training epoch.

DeCOS repeats such optimization loops until one of two conditions is met: (1) the performance of the target program reaches an expected level, where the model determines that any further optimizations of the target program can hardly be done, or (2) the maximum allowable time for the optimization process is consumed. The best-performing optimization sequences identified during the search are then presented as the final result.

As an RL-based framework, DeCOS is designed to *learn on the job*, efficiently training itself using high-quality data generated during the optimization process while simultaneously exploring the optimization space and identifying optimal sequences. This capability necessitates that DeCOS be able to effectively adapt its strategy toward better directions during the task based on the experience it accumulates. To enable this adaptability, we implemented specific design features within DeCOS.

During its preprocessing step, DeCOS identifies and extracts the LLVM IRs of hot functions from the target programs, concentrating its optimization efforts on these critical functions. After extracting the LLVM IRs, DeCOS uses a code embedding tool to generate embeddings for each function, preparing them for further optimization. This implementation makes DeCOS operate at the function level, allowing for more fine-grained and accurate optimization, while also generating more training data from each costly program test. Then, before the RL-agent starts to iteratively optimize the target functions, its hyper-parameters were pre-tuned with a synthesized dataset to ensure the learning efficiency of the RL-agent on the optimization task. In addition, although the DeCOS's RL-agent can be sufficiently trained to guide its own exploration and learning effectively, external assistance is required for the initial phase of exploration. During this early stage, DeCOS employs an LLM predictor to provide guidance. By analyzing the target functions to be optimized, the LLM predictor suggests potential optimization sets to try. With this guidance, the RL-agent can more quickly identify and focus on promising exploration areas, significantly reducing the time required to accumulate sufficient experience for initialization. Once DeCOS enters its main optimization loops, it integrates simulated performance data of the target functions with real profiled performance data. This combination helps adjust noisy results and ensures accurate reward feedback R_k to the RL-agent. By maintaining clean and reliable performance records, DeCOS ensures that decisions made in later loops are not affected by potentially polluted profiling data from earlier loops. Meanwhile, to provide the RL-agent with sufficient observation O_k for effective learning and accurate predictions, DeCOS integrates multiple

sources of information, combining static features and dynamic feedback to create a comprehensive representation of the optimization status.

A detailed explanation of these synergistic design features will be provided in the following section.

4 DeCOS: Challenges and Solutions

This section presents a detailed implementation of DeCOS and discusses how it addresses the inherent limitations of the RL framework, corresponding to each challenge outlined in Section 1.

4.1 Tuning Hyper-parameters with Synthesized Data

At the outset, designing a high-quality RL-agent for the optimization task requires determining the appropriate hyper-parameters before proceeding with its actual implementation. The efficacy of Machine Learning approaches, especially Reinforcement Learning approaches, is highly dependent on the choice of hyper-parameters, including the size of the neural network, learning rate, formulation of the reward function, etc. Unfortunately, determining hyper-parameters is often an ad-hoc process. In DeCOS, the high cost associated with training data collection prohibits an extensive exploration of the hyper-parameters. Thus, we derived a hyper-parameter selection process based on synthesized training data.

To create the synthesized dataset, we first randomly optimized several small programs from MiBench [14], profiling and collecting the compiler-benchmark interactions. By extracting key statistical characteristics from this small dataset, we generated a synthesized dataset simulating relationships among optimization sets, hardware counters, and code features. While this synthesized dataset does not perfectly reflect the real optimization tasks, it replicates the task's general complexity and logical structure. As such, it may not provide sufficient knowledge to train the RL model directly but is effective for determining its hyper-parameters.

Using the synthesized dataset, we tuned the hyper-parameters of DeCOS's RL model. This approach not only facilitated efficient hyper-parameter tuning but also enabled us to evaluate the RL-agent's efficacy comprehensively without incurring a resource-intensive training phase. By leveraging this process, we established a robust foundation for the RL-agent, ensuring it is well-prepared for real-world optimization tasks.

Figure 3(a) and Figure 3(b) show the results of hyper-parameter evaluation on the synthesized dataset. Figure 3(a) illustrates the impact of neural network architecture on the training performance. Smaller networks, such as NN256-L3 (3 layers with 256, 128, and 64 nodes) and NN512-L3 (3 layers with 512, 256, and 128 nodes), fail to train efficiently on the synthesized task. However, increasing the size beyond NN512-L4 (4 layers with 512, 256, 256, and 128 nodes) results in diminishing returns. Based on this evaluation, the architecture of the RL-agent in DeCOS was set to NN512-L4 to reach a balance between performance and computational efficiency. NN512-L4, identified as the smallest efficient neural network during hyper-parameter evaluation, can be trained and executed quickly on typical CPUs, even in the absence of GPUs.

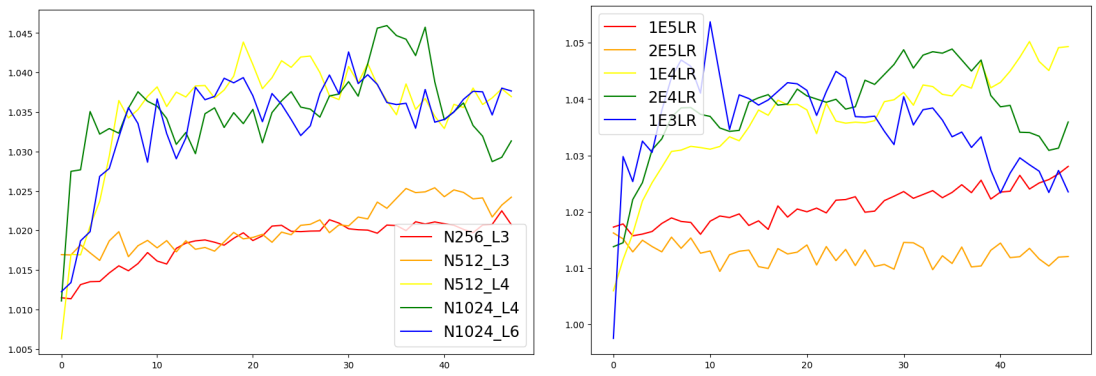
Figure 3(b) evaluates the effect of learning rates on training performance. Low learning rates, such as 1E5LR (1×10^{-5}), are characterized by flatter slopes, indicating lower learning efficiency on the synthesized dataset. Conversely, higher learning rates, such as 2E4LR (2×10^{-4}) or 1E3LR (1×10^{-3}), initially exhibit promising performance in the first few epochs but then lead to overfitting and performance degradation in later stages. To balance learning efficiency and model stability, DeCOS selects a learning rate of 1E4LR (1×10^{-4}) during its early training phase, and selects the learning rate of 1E3LR for a trained model when applied to an unseen optimization target and aims for faster convergence. This dedicated model design significantly contributes to DeCOS's performance, enabling it to achieve a balance of efficiency and stability. This task would be difficult to accomplish without the synthesized dataset and corresponding hyper-parameter tuning. Additionally, DeCOS also evaluated several reward functions using the synthesized dataset and selected the most effective one, that for each individual optimization pass P_k applied, the reward R_k is calculated as the difference in CPU cycles before and after the pass. Once an entire optimization sequence is completed, an additional reward based on the total optimization result is applied, directing the RL-agent toward sequences that maximize overall performance improvements.

4.2 Augmented Code Representation for RL Optimization

As the key component of DeCOS, the RL-agent in DeCOS is a neural network that performs a mapping to predict the next optimization pass based on an observation of the current compilation and execution status of the target application. To ensure efficient predicting, the observation given as the input to the RL-agent needs to be carefully designed to contain sufficient information.

In DeCOS, we refer to the combined static and dynamic information as Augmented Code Representation.

DeCOS uses IR2vec [32], a scalable encoding infrastructure to encode the intermediate representation (IR) as vectors ready for the RL-agent. The default IR2vec produces a 50-dimensional vector for a given input code segment. In DeCOS, the representation is reduced to eight dimensions. The RL-agent is sensitive to input space size, as high-dimensional



(a) Neural network size (b) Learning rate

Figure 3: Hyper-parameter evaluation: X-axis is training epoch; Y-axis is the average performance for the epoch.

inputs lead to a complex mapping that is hard to learn. This constraint is exacerbated by the high-cost associated with generating training data. Thus, reducing the embedding dimension enables better data efficiency. Our evaluations on Splash-3 [27] indicate that with the reduction of embedding dimensions to 8, IR2vec retains its capability to cluster distinct functions. This indicates that the dimensionality reduction preserves sufficient information in the embedding, as DeCOS operates at the function level, where the representation inherently captures less complexity than the entire program. Once generated, the code embedding of the target functions remains unchanged throughout the optimization process, to provide static program information.

Performance information serve as another crucial information source for the DeCOS model, guiding its exploration of the optimization space for the target application. Cavazos et al. [8] demonstrated that performance information such as Hardware Performance Counters (HPCs) can help address the problem of predicting effective compiler optimizations as the input for a logistic regression model, a form of supervised learning. However, the dynamic nature of performance information, which reflects the current execution state of the target application, makes it particularly better suited for RL frameworks. This information provides detailed insights into the impact of each applied optimization, delivering comprehensive feedback that enables the RL-engine to learn and adapt effectively.

Furthermore, integrating performance information not only enhances decision-making but also enables DeCOS to perform cross-platform optimization. The optimization of a target application is fundamentally based on a vector of {program, environment}, while a code representation alone only contains static information about the program. The challenge of applying a trained model to different hardware platforms can be addressed by incorporating runtime performance information into the workflow, as it reflects the current state of the environment. For instance, while unrolling options may generally improve performance, they can degrade it on devices with insufficient instruction cache. Without observing performance data in the new environment, a model cannot make such accurate decisions. In summary, a general model capable of optimizing multiple types of programs across diverse hardware platforms can only be realized by incorporating performance feedback into the optimization process. It is worth noticing that including all potential performance information can significantly increase the complexity of the input, and thus slow down the training process. Therefore we only incorporate the most commonly used information, that is CPU cycles, number of instructions, cache-misses, branch-misses, L1-dcache-load-misses, L1-dcache-write-misses, and L1-icache-load-misses, where the number of CPU cycles is also the optimization target and is used to calculate the reward. We normalize these counters at first by calculating the ratio of $COUNTER/CYCLE$ to show overall how relatively large these counters are instead of their absolute values. Since most counters are significantly less than cycles, we use the logarithm of the $COUNTER/CYCLE$ ratio as the observation.

Additionally, we profile the 3 most time-consuming instructions as part of the observation, which is also a piece of information commonly used by human developers. Knowing the type of the most time-consuming instructions (for example, whether they are floating-point multiplications, or load instructions) can help decide the best optimization to take. These instructions are represented in the form of a word embedding generated by the Word2vec [22] model, which provides additional hints for optimization.

Last but not least, as evaluated in Autophase [17], including already-taken options into the observation helps the model make better decisions. By also including already-taken options as part of the observation, the RL-agent can have a better view of the current compilation state, and therefore make a better prediction.

In conclusion, DeCOS constructs an information-rich observation to equip the RL-agent with the data necessary for making well-informed decisions. This observation integrates code embeddings to represent the static features of the target program, augmented with hardware performance counters to capture its dynamic runtime behavior and current optimization state. Additionally, the observation is enhanced with details about previously applied optimization options and time-consuming instructions, further guiding the search toward efficient directions.

4.3 Boosting Early Performance: Integrating LLMs for Start-up Acceleration

RL-agents require exploration experience within the target environment to effectively get trained. While an experienced RL-agent can guide its own exploration after initial training, it cannot perform efficient exploration during the early phases, often relying on random exploration.

In a large search space, random exploration is highly inefficient, leading to a slow startup. To overcome this limitation, DeCOS integrates Large Language Models (LLMs) to guide the initial exploration phase. LLMs are employed to analyze the target codes, and generate potential optimizations correspondingly, allowing the RL-agent to learn from the resulting optimization outcomes. These LLM-suggested optimizations are typically more effective and provide more valuable information for subsequent exploration compared to randomly generated ones. Additionally, DeCOS enables further interaction between its RL-agent and the LLM model. This collaboration allows the LLM to analyze optimization options selected by the RL-agent and suggest potential improvements. This dynamic feedback loop enables the LLM also be able to benefit from the RL-agent's learned strategies without requiring costly fine-tuning or weight updates. By leveraging the knowledge and reasoning capabilities of LLMs, DeCOS enables the RL-agent to quickly acquire meaningful experience and identify promising directions, significantly accelerating the learning process. During its training and evaluation, DeCOS employs GPT-4o-mini [25] as its collaborating LLM.

4.4 Handling Noise with Simulated Results

Performance information serves multiple critical purposes in DeCOS: while providing runtime behavior data of the optimization target as part of the efficient observation input to the RL-agent for decision-making, it also functions as the ultimate objective of optimization, being used to train and evaluate the performance of DeCOS, towards fewer CPU cycles.

Performance information can be collected using profiling tools like Linux `perf`. Previous approaches, such as Milepost-GCC [12], typically profile a program multiple times and use the average of the results. However, DeCOS iteratively applies one optimization at a time, resulting in relatively small differences between successive optimized binaries. This amplifies the impact of profiling noise. Moreover, as an RL-based system, DeCOS is particularly sensitive to noisy data, as it can introduce long-term bias in the search directions, negatively affecting the overall optimization process. These features suggest the necessity of a stabler performance measurement for training DeCOS.

Simulators provide an effective solution for ensuring accurate and stable performance measurements. For instance, Autophase [16] uses a domain-specific HLS simulator to estimate the performance of optimized HLS designs. However, general-purpose simulators might not be able to accurately estimate the runtime performance of target programs. Therefore, DeCOS utilizes Cachegrind, a component of the Valgrind simulation suite [24], to provide stable output, alongside the profiled results, to provide corrected feedback. After each optimization pass is applied, DeCOS uses Cachegrind to simulate the updated program and collect updated simulated runtime and hardware performance counters. Once a complete optimization sequence is formulated, DeCOS calculates an additional accumulated performance gain based on real profiled data, ensuring that the system remains focused on achieving the ultimate goal of improved profiled performance. If the profiled performance varied significantly from the simulated performance, DeCOS records such results for further validation and correction. Since simulation results are noise-free and used only for intermediate observation and decision-making, DeCOS employs a smaller input dataset for simulations, such as the `test` input for `SPEC`, to significantly reduce the computational cost of using the simulator.

5 Evaluation

In this section, we evaluate the effectiveness of DeCOS using `Clang-12` as the compilation infrastructure. Optimization and performance results are collected on AMD EPYC 7763 processors. All results are measured in CPU-cycles and are averaged over four repeated runs. When the deviation of the profiling results is greater than a threshold of 5%, an additional 4 runs are invoked. Simulated results, generated by Cachegrind [24], are used solely as intermediate performance

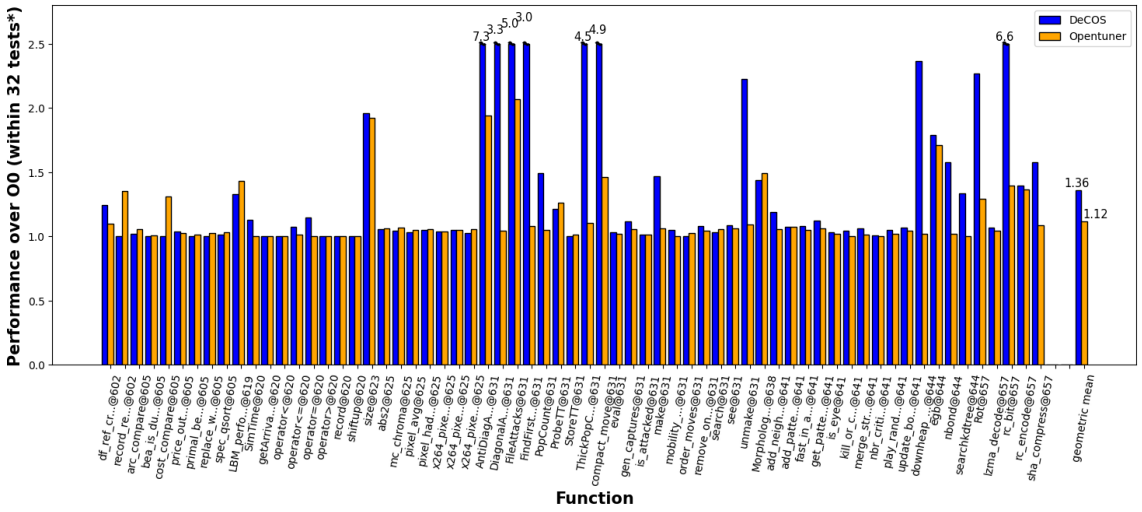


Figure 4: Best performance achieved by DeCOS and `Opentuner` on `SPEC` (hot functions), evaluated on the AMD EPYC 7763 processor, where DeCOS is trained. The Y-axis represents performance normalized to that of `-O0`.

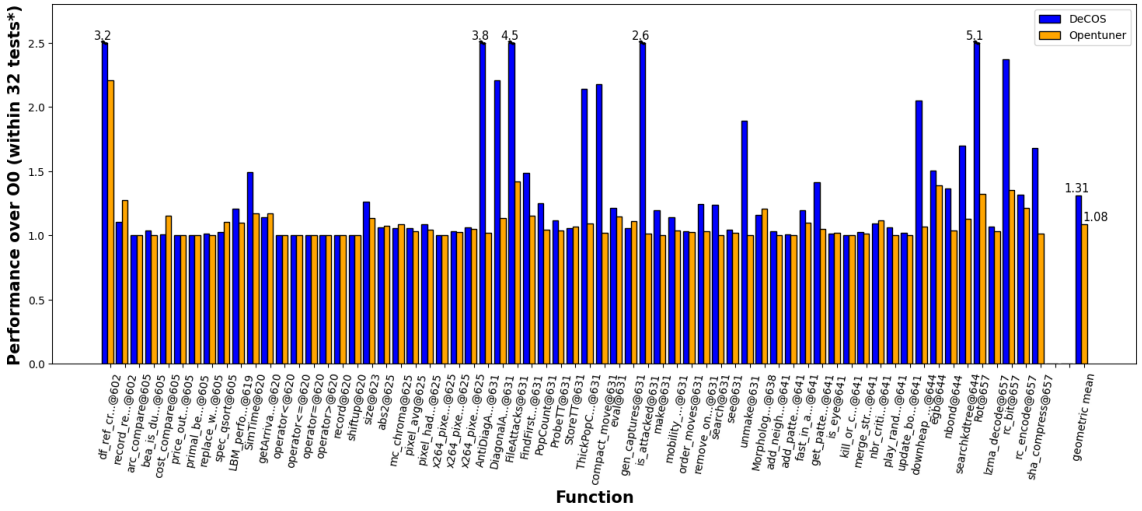


Figure 5: Best performance achieved by DeCOS and `Opentuner` on `SPEC` (hot functions), evaluated on an AMD Ryzen 9950X CPU, which is a hardware environment that DeCOS is not exposed to during training. The Y-axis represents performance normalized to that of `-O0`.

information to minimize noise and are not included in the reported performance.

We aim to evaluate the performance of DeCOS on all benchmarks from `SPEC CPU 2017 Speed` [7] that are written in C or C++. 600.PERLBENCH_S from `SPEC INT` is excluded due to compilation challenges with the Clang infrastructure. For each benchmark, we evaluate DeCOS on all hot functions, i.e., functions that correspond to more than 2% of the total execution time. To ensure a fair comparison and avoid inter-function interference, optimizations are applied exclusively to hot functions extracted using the `llvm-extract`.

5.1 Performance and Portability

DeCOS is designed to continuously improve its efficiency by learning from accumulated experience. To evaluate this capability, we train DeCOS on optimization tasks for `Splash-3` and `Parsec-3` and test its effectiveness on `SPEC CPU 2017`.

The results are compared with the state-of-the-art performance tuner, `Opentuner` [5], which combines multiple search-based optimization approaches. For a fair comparison, we limit the optimization option sequences to 16 optimizations chosen from the 60 commonly used options, with the possibility of re-applying the passes. Once DeCOS or `Opentuner` generates a complete optimization sequence, the optimized program is tested and profiled. The reported results are performance normalized to that of `-O0`.

As shown in Figure 4, the evaluations are conducted with the same amount of effort, measured by the number of tests and profiles performed on each program. Each benchmark is tested for up to 32 tests/profiles. Due to experimental environment constraints, for benchmarks with long test/profiling cycles, the number of tests/profiles is adjusted and aligned across all experiments to fit within a 24-hour testing cycle. DeCOS achieves better performance on 61.5% (40/65) of hot functions from `SPEC CPU 2017` compared to `Opentuner`, and achieves similar performance on 7.7% (5/65) of hot functions. The results demonstrate that DeCOS can effectively optimize `SPEC` benchmarks it has not encountered during training. On average, DeCOS outperforms `Opentuner` by 21.4% (geometric mean).

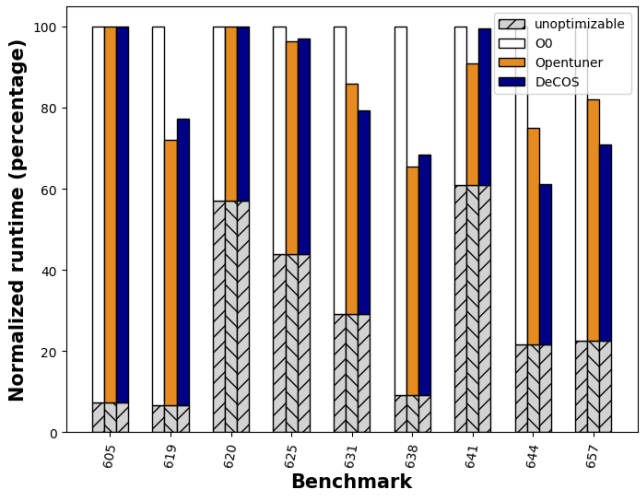


Figure 6: CPU-cycles of optimized `SPEC` benchmarks normalized to that of `-O0` (lower bars show better performance). For each benchmark, the diagonally striped components represent functions that are not identified as hot functions during evaluation and could not be optimized by either DeCOS or `Opentuner` in the evaluation workflow.

For overall program performance, we combine the best-optimized hot functions from each benchmark and profile the total runtime. The results are presented in Figure 6, DeCOS outperforms `Opentuner` on 605.mcf_s, 631.deepsjeng_s, 644.nab_s and 657.xz_s, aligned with the performance on the hot functions. However, for benchmark 641.leela_s, Although DeCOS achieves better function-level performance, it is outperformed by `Opentuner` at the program level. Synergistic approaches that consider both function-level and program-level performance can potentially mitigate this issue. Benchmarks 602.gcc_s and 623.xalancbmk_s are excluded from the program-level comparison (despite DeCOS performing better on both), because hot functions in these benchmarks account for less than 5% of the total execution time.

DeCOS can optimize target programs using performance counters in addition to code features. This capability is expected to enable DeCOS to adapt and perform effectively in execution environments that it is previously unexposed to. We apply DeCOS, trained on AMD EPYC 7763 CPUs, to a hardware environment equipped with an AMD Ryzen 9950X CPU. As shown in Figure 5, DeCOS maintains its efficiency on the new hardware platform. Compared to `Opentuner`, DeCOS achieves better performance on 66.2% (43/65) of the hot functions and similar performance on 16.9% (11/65) of the hot functions. The results indicate that DeCOS is able to effectively transfer its optimization capabilities to previously unseen programs, even on hardware platforms it has not been exposed to during training. On average, DeCOS outperforms `Opentuner` by 26.8%, demonstrating a clear advantage.

For `SPEC CPU` benchmarks, the `-O3` option can potentially perform transformations that incur significant performance degradation that is difficult to reverse. Thus, DeCOS chooses to explore the optimization space using `-O0` as the baseline. At the program level, DeCOS achieves a 5.1% performance improvement over `-O3`. A hybrid approach that applies the best-performing optimizations for each function, either `-O3` or DeCOS, achieves a 5.7% performance improvement compared to that of using `-O3` alone.

5.2 Data Efficiency

In this section, we evaluate the data efficiency of DeCOS with a case study examining its training process on `Splash-3`.

Figure 7 illustrates a clear trend of performance improvement during training, highlighting DeCOS's ability to learn and adapt as it accumulates experience. The X-axis represents epochs over the training process, where each epoch contains the experience of constructing 3 completed 16-option optimization sets. In these training epochs, the performance updates of the first 13 options are produced by the Cachegrind simulator, while the performance outcomes of the last 3 options are profiled with `perf` and refined with simulation results. While the Y-axis indicates performance normalized to the `-O0` optimization baseline. The `perf_epoch` line denotes the best performance achieved

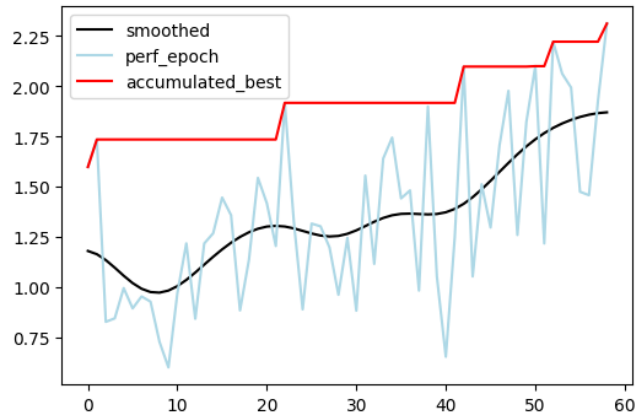


Figure 7: Performance achieved vs. training iteration for the `Transpose` function from `Splash-3 fft`: X-axis is the training epoch number; Y-axis is the performance normalized to that of `-O0`. The `perf_epoch` line represents the best performance achieved within each epoch, while the smoothed line is a Gaussian-smoothed version of `perf_epoch`. The `accumulated_best` line tracks the best performance achieved with prior epochs.

within each epoch, while the `accumulated_best` line reflects the best performance observed across all epochs up to that point. Within a relatively small number of epochs, the performance improvement achieved over time shown by the `perf_epoch` and `accumulated_best` lines demonstrates that DeCOS is capable of progressively updating and adapting its optimization strategy. To better visualize this overall trend, the `smoothed` line presents a Gaussian-smoothed version of the `perf_epoch` data, showing a consistent increase in performance during the search. Although both the `perf_epoch` and the `smoothed` lines trend upwards, the `perf_epoch` line shows significant fluctuation. This indicates that DeCOS is balancing its efforts between focusing on promising sub-spaces that are likely to yield better performance and exploring broader regions of the optimization space.

5.3 Overhead

DeCOS introduces two additional modules compared to traditional heuristic models: an RL-engine and an LLM query interface. As described in Section 4.1, the RL-engine is carefully designed to balance performance and efficiency, incurring negligible overhead (less than 0.1 seconds for each prediction). The overhead introduced by the LLM query interface is less straightforward to evaluate, as it relies on external API calls rather than local computation. We estimate its impact using wall-clock time. To quantify the overhead, we sample 32 queries during the evaluation and record the response times from OpenAI's server. On average, each query takes 13.7 seconds to complete. Since the LLM query interface is only used sparingly to accelerate RL-agent initialization, its contribution to the total wall-clock time is minimal. In our logs, these two modules incur less than 1% of wall-clock time overhead. Furthermore, both modules can run in parallel with the profiling and testing phases, further reducing their impact on overall runtime.

5.4 Ablation Studies

This section presents ablation studies focusing on the advanced code representation enhanced with hardware performance counters and the integration of LLMs. This evaluation includes the following configurations:

(1) A baseline DeCOS model utilizing advanced code representation but without LLM-integration.
(2) Baseline DeCOS model with LLM integrated.
(3) Baseline DeCOS model without hardware performance counter information in its code representation.

The comparison between configuration 1 and configuration 3 highlights the impact of incorporating hardware performance counters into the code representation. Similarly, the comparison between configuration 1 and configuration 2 demonstrates the impact of LLM-integration on optimization performance. It is worth pointing out that, comparison with configuration 3 also serves as a comparison against existing RL-based optimizers such as Autophase. Autophase solely relies on program features, optimization options, and overall performance to guide the RL-engine, and thus it is equivalent to a DeCOS configuration without LLM-integration or hardware performance counter feedback.

5.4.1 Advanced Code Representation. Compared to existing reinforcement learning-based optimization approaches, DeCOS incorporates hardware performance counters into its workflow. This integration offers valuable insights into the runtime behaviors of the target program, enabling more informed and effective optimization decisions. To demonstrate the effectiveness of the advanced code representation, we conduct a comparison using the same collected experience employed to train the baseline DeCOS. Specifically, we train a modified model, `DeCOS_withoutHPC`, in which the hardware performance counter observations are masked with zeros. This model is then applied to the same testing benchmarks where the baseline DeCOS is evaluated, with hardware counters masked. The evaluation results indicate that, compared to the baseline DeCOS, the `DeCOS_withoutHPC` model achieves better optimization results on only 32.3% of the functions, while performance decreases on 52.3% of the functions. The geometric mean of the optimized performance of the functions (normalized to `-O0`) also decreases

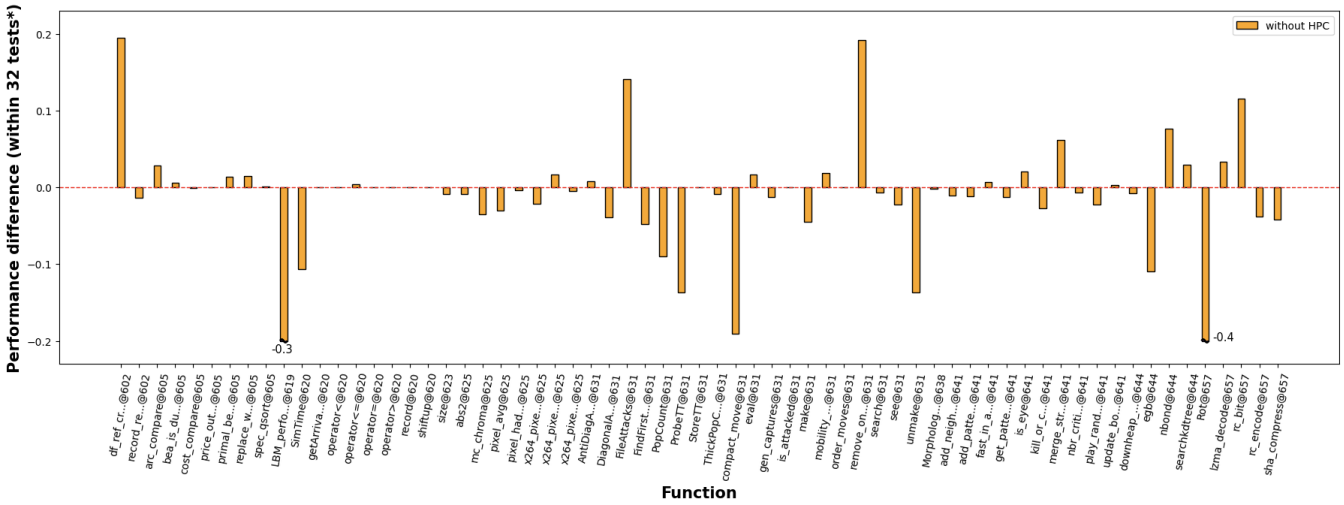


Figure 8: Performance Comparison: Differences in performance on the best-optimized `SPEC CPU 2017` functions achieved by DeCOS without hardware performance counters (`DeCOS_withoutHPC`), compared to the baseline DeCOS. Masking hardware performance counters has a negative impact on the performance of most functions.

by approximately 2%. As shown in Figure 8, masking hardware performance counters results in a negative impact on the performance. These findings highlight the impact of incorporating hardware performance counters into the code representation, enabling DeCOS to make more informed and optimal decisions.

In the evaluation phase, DeCOS must trade off the frequency of updating the hardware performance information and the associated performance overhead as minimizing overhead is important. Performing simulations after each optimization step incurs 16 simulations, introducing significant overhead. We limit the number of simulations to one during evaluation, updating the hardware performance information in the observation only after applying the first optimization option. This setup ensures minimal overhead, and all evaluation results presented in this paper are conducted under this setup. On the contrary, during the training phase, hardware performance counter information is updated after each optimization option is applied, providing detailed data to support efficient learning.

We evaluate a simplified version of DeCOS that eliminates all simulations during the optimization process, except for a single simulation of the unoptimized program at the starting point. With this configuration, the hardware performance information of the target function in the observation remains static, ensuring that each sequence of options formulated by DeCOS incurred exactly the same cost as `Opentuner` with no additional overhead. For `SPEC`, DeCOS's predictions remain largely unchanged in the initial epochs during the evaluation phase. This is because the `SPEC` benchmarks in this experiment run for relatively few epochs within a 24-hour window. However, benchmark behaviors diverge with additional epochs, indicating a trade-off between simulation overhead and the level of performance achieved.

LLM-Integration	Evaluating Phase	Training Phase
No. of Functions improved	35/65 (53.8%)	53/91 (58.2%)
No. of Functions degraded	19/65 (29.2%)	38/91 (41.8%)
Change in geometric mean	-0.005	+0.083

Table 1: Impact of LLM-integration: performance difference compared to the baseline DeCOS.

5.4.2 LLM-Integration. DeCOS integrates LLM into its RL infrastructure to speedup the start-up phase in training. To demonstrate the efficiency of this design, we integrate an LLM model into the baseline DeCOS, and evaluate the performance of DeCOS with an integrated LLM model. To isolate the impact of LLM-integration on DeCOS's training speed, the statistical results include only the optimization sets generated by DeCOS. Optimization sets suggested by the LLM are used only when updating DeCOS's strategy and are not factored into the presented outcomes. The performance results are summarized in Table 1.

Guided by the LLM, the trained DeCOS improves performance on 53.8% (35/65) of the functions during the evaluating phase, while it degrades performance on 29.2% (19/65) of the functions. Although LLM is effective on the majority of the functions evaluated, there is a slight decrease in

the geometric mean. This suggests a potential limitation of the LLM-integration. For a trained DeCOS, LLM-integration offers limited advantages, because LLM can potentially degrade performance on functions for which DeCOS is already effective.

As proposed in section 3, the primary purpose of LLM-integration is to accelerate the initial phase of model training, rather than to improve the performance of a trained model. To evaluate this effect, we conduct an additional evaluation during the training phase of DeCOS. Specifically, we compare the performance of an untrained DeCOS model with and without LLM-integrated on the training tasks (optimizing functions from `Splash-3` and `Parsec-3`). With the same amount of training time, DeCOS with LLM-integration outperforms the version without LLM. Compared to the configuration without LLM integration, 58.2% of functions achieve improved performance, while 41.8% experience performance degradation. LLM-integrated DeCOS also shows an improvement in geometric mean performance. Overall, our results suggest that integrating the LLM during the evaluation phase provides moderate benefits. However, incorporating it during the training phase has more significant benefits in enhancing training efficiency.

6 Conclusions

Machine learning techniques have been widely attempted for various program optimization tasks, however, their adoption is often hindered by the size and the complexity of the optimization space. In addition, the lack of inexpensive and accurate training data can slow the deployment of machine learning algorithms in optimization tasks.

This paper proposes DeCOS, a data-efficient reinforcement learning approach ignited by LLM, designed to automatically select compilation optimization passes for any given application. Benefiting from its RL-infrastructure, DeCOS is able to separate a complex mapping task of selecting a completed optimization sequence into multiple simpler sub-tasks of selecting a single optimization option. This decomposition significantly reduces the size of the machine learning problem and enables better data efficiency. Meanwhile, it also enables DeCOS to benefit from dynamically interacting with the optimization space and efficiently learn during this process with observed feedback.

DeCOS addresses many inherent challenges of applying RL-infrastructures to optimization tasks. Key innovations include an efficient hyper-parameter pre-selection process, which determines the optimal architecture of the RL model without significant cost; an enhanced representation combining static and dynamic information to effectively describe the state of the optimization task; a novel LLM-integration into RL-infrastructure, enabling the RL-model to leverage LLM guidance during its initial training phase, accelerating its early training; and a refined performance evaluation strategy, incorporating simulation results to mitigate profiling noise and reduce the cost of data collection, ensuring reliable training data.

In summary, DeCOS is able to generate efficient compilation optimization sequences that can outperform `Opentuner` on a significant number of `SPEC` benchmarks. Meanwhile, DeCOS has demonstrated the ability to port learned experience across benchmarks and across hardware platforms. Ablation studies further validate the efficiency and impact of DeCOS's key components.

Acknowledgments

This research was supported in part by NSF Grant CNS-2106771.

References

[1] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M.F.P. O'Boyle, J. Thomson, M. Toussaint, and C.K.I. Williams. 2006. Using machine learning to focus iterative optimization. In *International Symposium on Code Generation and Optimization (CGO'06)*. 11 pp.–305. doi:10.1109/CGO.2006.37

[2] Byung Hoon Ahn, Prannoy Pilligundla, Amir Yazdanbakhsh, and Hadi Esmaeilzadeh. 2020. Chameleon: Adaptive Code Optimization for Expedited Deep Neural Network Compilation. arXiv:2001.08743 [cs.LG]

[3] L. Almagor, Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven W. Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. 2004. Finding Effective Compilation Sequences. *SIGPLAN Not.* 39, 7 (jun 2004), 231–239. doi:10.1145/998300.997196

[4] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2018. code2vec: Learning Distributed Representations of Code. arXiv:1803.09473 [cs.LG]

[5] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. OpenTuner: An extensible framework for program autotuning. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*. 303–315. doi:10.1145/2628071.2628092

[6] Amir H. Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. 2018. A Survey on Compiler Autotuning Using Machine Learning. *ACM Comput. Surv.* 51, 5, Article 96 (sep 2018), 42 pages. doi:10.1145/3197978

[7] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. 2018. SPEC CPU2017: Next-Generation Compute Benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering* (Berlin, Germany) *(ICPE '18)*. Association for Computing Machinery, New York, NY, USA, 41–42. doi:10.1145/3185768.3185771

[8] John Cavazos, Grigori Fursin, Felix Agakov, Edwin Bonilla, Michael F.P. O'Boyle, and Olivier Temam. 2007. Rapidly Selecting Good Compiler Optimizations using Performance Counters. In *International Symposium on Code Generation and Optimization (CGO'07)*. 185–197. doi:10.1109/CGO.2007.32

[9] Keith D. Cooper, Devika Subramanian, and Linda Torczon. 2002. Adaptive Optimizing Compilers for the 21st Century. *J. Supercomput.* 23, 1 (aug 2002), 7–22. doi:10.1023/A:1015729001611

[10] Chris Cummins, Volker Seeker, Dejan Grubisic, Mostafa Elhoushi, Youwei Liang, Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Kim

Hazelwood, Gabriel Synnaeve, and Hugh Leather. 2023. Large Language Models for Compiler Optimization. arXiv:2309.07062 [cs.PL] https://arxiv.org/abs/2309.07062

[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. 2021. CompilerGym: Robust, Performant Compiler Optimization Environments for AI Research. arXiv:2109.08267 [cs.PL]

[12] Grigori Fursin, Cupertino Miranda, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Ayal Zaks, Bilha Mendelson, Edwin Bonilla, John Thomson, Hugh Leather, Chris Williams, Michael O'Boyle, Phil Barnard, Elton Ashton, Eric Courtois, and François Bodin. 2008. MILEPOST GCC: machine learning based research compiler. In *GCC Summit*. Ottawa, Canada.

[13] Shuzheng Gao, Cuiyun Gao, Wenchao Gu, and Michael Lyu. 2025. Search-Based LLMs for Code Optimization . In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, Los Alamitos, CA, USA, 254–266. doi:10.1109/ICSE55347.2025.00021

[14] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. 2001. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop (WWC '01)*. IEEE Computer Society, USA, 3–14.

[15] Ameer Haj Ali. 2021. *Machine Learning in Compiler Optimization*. Ph. D. Dissertation. EECS Department, University of California, Berkeley.

[16] Q. Huang, A., W. Moses, J. Xiang, I. Stoica, K. Asanovic, and J. Wawrzynek. 2020. AutoPhase: Juggling HLS Phase Orderings in Random Forests with Deep Reinforcement Learning. arXiv:2003.00671 [cs.DC]

[17] Qijing Huang, Ameer Haj-Ali, William Moses, John Xiang, Ion Stoica, Krste Asanovic, and John Wawrzynek. 2019. AutoPhase: Compiler Phase-Ordering for HLS with Deep Reinforcement Learning. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 308–308. doi:10.1109/FCCM.2019.00049

[18] Qijing Huang, Ruolong Lian, Andrew Canis, Jongsok Choi, Ryan Xi, Stephen Brown, and Jason Anderson. 2013. The Effect of Compiler Optimizations on High-Level Synthesis for FPGAs. In *2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*. 89–96. doi:10.1109/FCCM.2013.50

[19] Timo Kaufmann, Paul Weng, Viktor Bengs, and Eyke Hüllermeier. 2024. A Survey of Reinforcement Learning from Human Feedback. arXiv:2312.14925 [cs.LG] https://arxiv.org/abs/2312.14925

[20] Scott Robert Ladd. 2004. ACOVEA (Analysis of Compiler Options via Evolutionary Algorithm). https://github.com/Acovea/libacovea

[21] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. San Jose, CA, USA, 75–88.

[22] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. doi:10.48550/ARXIV.1301.3781

[23] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. 2012. *Foundations of Machine Learning*. The MIT Press.

[24] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. *SIGPLAN Not.* 42, 6 (jun 2007), 89–100. doi:10.1145/1273442.1250746

[25] OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mohammad Bavarian, Jeff Belgum, Irwan Bello, Jake Berdine, Gabriel Bernadett-Shapiro, Christopher Berner, Lenny Bogdonoff, Oleg Boiko, Madelaine Boyd, Anna-Luisa Brakman, Greg Brockman, Tim Brooks, Miles Brundage, Kevin Button, Trevor Cai, Rosie Campbell, Andrew Cann, Brittany Carey, Chelsea Carlson, Rory Carmichael, Brooke Chan, Che Chang, Fotis Chantzis, Derek Chen, Sully Chen, Ruby Chen, Jason Chen, Mark Chen, Ben Chess, Chester Cho, Casey Chu, Hyung Won Chung, Dave Cummings, Jeremiah Currier, Yunxing Dai, Cory Decareaux, Thomas Degry, Noah Deutsch, Damien Deville, Arka Dhar, David Dohan, Steve Dowling, Sheila Dunning, Adrien Ecoffet, Atty Eleti, Tyna Eloundou, David Farhi, Liam Fedus, Niko Felix, Simón Posada Fishman, Juston Forte, Isabella Fulford, Leo Gao, Elie Georges, Christian Gibson, Vik Goel, Tarun Gogineni, Gabriel Goh, Rapha Gontijo-Lopes, Jonathan Gordon, Morgan Grafstein, Scott Gray, Ryan Greene, Joshua Gross, Shixiang Shane Gu, Yufei Guo, Chris Hallacy, Jesse Han, Jeff Harris, Yuchen He, Mike Heaton, Johannes Heidecke, Chris Hesse, Alan Hickey, Wade Hickey, Peter Hoeschele, Brandon Houghton, Kenny Hsu, Shengli Hu, Xin Hu, Joost Huizinga, Shantanu Jain, Shawn Jain, Joanne Jang, Angela Jiang, Roger Jiang, Haozhun Jin, Denny Jin, Shino Jomoto, Billie Jonn, Heewoo Jun, Tomer Kaftan, Łukasz Kaiser, Ali Kamali, Ingmar Kanitscheider, Nitish Shirish Keskar, Tabarak Khan, Logan Kilpatrick, Jong Wook Kim, Christina Kim, Yongjik Kim, Jan Hendrik Kirchner, Jamie Kiros, Matt Knight, Daniel Kokotajlo, Łukasz Kondraciuk, Andrew Kondrich, Aris Konstantinidis, Kyle Kosic, Gretchen Krueger, Vishal Kuo, Michael Lampe, Ikai Lan, Teddy Lee, Jan Leike, Jade Leung, Daniel Levy, Chak Ming Li, Rachel Lim, Molly Lin, Stephanie Lin, Mateusz Litwin, Theresa Lopez, Ryan Lowe, Patricia Lue, Anna Makanju, Kim Malfacini, Sam Manning, Todor Markov, Yaniv Markovski, Bianca Martin, Katie Mayer, Andrew Mayne, Bob McGrew, Scott Mayer McKinney, Christine McLeavey, Paul McMillan, Jake McNeil, David Medina, Aalok Mehta, Jacob Menick, Luke Metz, Andrey Mishchenko, Pamela Mishkin, Vinnie Monaco, Evan Morikawa, Daniel Mossing, Tong Mu, Mira Murati, Oleg Murk, David Mély, Ashvin Nair, Reiichiro Nakano, Rajeev Nayak, Arvind Neelakantan, Richard Ngo, Hyeonwoo Noh, Long Ouyang, Cullen O'Keefe, Jakub Pachocki, Alex Paino, Joe Palermo, Ashley Pantuliano, Giambattista Parascandolo, Joel Parish, Emy Parparita, Alex Passos, Mikhail Pavlov, Andrew Peng, Adam Perelman, Filipe de Avila Belbute Peres, Michael Petrov, Henrique Ponde de Oliveira Pinto, Michael, Pokorny, Michelle Pokrass, Vitchyr H. Pong, Tolly Powell, Alethea Power, Boris Power, Elizabeth Proehl, Raul Puri, Alec Radford, Jack Rae, Aditya Ramesh, Cameron Raymond, Francis Real, Kendra Rimbach, Carl Ross, Bob Rotsted, Henri Roussez, Nick Ryder, Mario Saltarelli, Ted Sanders, Shibani Santurkar, Girish Sastry, Heather Schmidt, David Schnurr, John Schulman, Daniel Selsam, Kyla Sheppard, Toki Sherbakov, Jessica Shieh, Sarah Shoker, Pranav Shyam, Szymon Sidor, Eric Sigler, Maddie Simens, Jordan Sitkin, Katarina Slama, Ian Sohl, Benjamin Sokolowsky, Yang Song, Natalie Staudacher, Felipe Petroski Such, Natalie Summers, Ilya Sutskever, Jie Tang, Nikolas Tezak, Madeleine B. Thompson, Phil Tillet, Amin Tootoonchian, Elizabeth Tseng, Preston Tuggle, Nick Turley, Jerry Tworek, Juan Felipe Cerón Uribe, Andrea Vallone, Arun Vijayvergiya, Chelsea Voss, Carroll Wainwright, Justin Jay Wang, Alvin Wang, Ben Wang, Jonathan Ward, Jason Wei, CJ Weinmann, Akila Welihinda, Peter Welinder, Jiayi Weng, Lilian Weng, Matt Wiethoff, Dave Willner, Clemens Winter, Samuel Wolrich, Hannah Wong, Lauren Workman, Sherwin Wu, Jeff Wu, Michael Wu, Kai Xiao, Tao Xu, Sarah Yoo, Kevin Yu, Qiming Yuan, Wojciech Zaremba, Rowan Zellers, Chong Zhang, Marvin Zhang, Shengjia Zhao, Tianhao Zheng, Juntang Zhuang, William Zhuk, and Barret Zoph. 2024. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL] https://arxiv.org/abs/2303.08774

[26] Stuart Russell and Peter Norvig. 2009. *Artificial Intelligence: A Modern Approach* (3rd ed.). Prentice Hall Press, USA.

[27] Christos Sakalis, Carl Leonardsson, Stefanos Kaxiras, and Alberto Ros. 2016. Splash-3: A properly synchronized benchmark suite for contemporary research. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 101–111. doi:10.1109/ISPASS.2016.7482078

[28] M. Stephenson and S. Amarasinghe. 2005. Predicting unroll factors using supervised classification. In *International Symposium on Code Generation and Optimization*. 123–134. doi:10.1109/CGO.2005.29

[29] R.S. Sutton and A.G. Barto. 1998. Reinforcement Learning: An Introduction. *IEEE Transactions on Neural Networks* 9, 5 (1998), 1054–1054. doi:10.1109/TNN.1998.712192

[30] Burak Tağtekin, Berkan Höke, Mert Kutay Sezer, and Mahiye Uluyağmur Öztürk. 2021. FOGA: Flag Optimization with Genetic Algorithm. arXiv:2105.07202 [cs.NE]

[31] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D.I. August. 2003. Compiler optimization-space exploration. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.* 204–215. doi:10.1109/CGO.2003.1191546

[32] S. VenkataKeerthy, Rohit Aggarwal, Shalini Jain, Maunendra Sankar Desarkar, Ramakrishna Upadrasta, and Y. N. Srikant. 2020. IR2VEC: LLVM IR Based Scalable Program Embeddings. *ACM Trans. Archit. Code Optim.* 17, 4, Article 32 (dec 2020), 27 pages. doi:10.1145/3418463

[33] Zheng Wang and Michael O'Boyle. 2018. Machine Learning in Compiler Optimisation. arXiv:1805.03441 [cs.PL]

[34] Minjia Zhang, Menghao Li, Chi Wang, and Mingqin Li. 2021. DynaTune: Dynamic Tensor Program Optimization in Deep Neural Network Compilation. In *International Conference on Learning Representations*.