Reinforcement Learning in Code Optimization and Generation: A Survey

Anonymous ACL submission

Abstract

With the rapid evolution of machine learning, reinforcement learning (RL) has emerged as a pivotal technique for optimizing code in various domains. This paper presents a systematic survey of the application of RL in code optimization and generation, highlighting its role in enhancing compiler optimizations, resource allocation, and the development of frameworks and tools. Subsequent sections delve into the intricate processes of compiler optimization, where RL algorithms are leveraged to improve efficiency and resource utilization. The discussion progresses to the function of RL in resource allocation, emphasizing register allocation and system optimization. We also explore the burgeoning role of frameworks and tools in code generation, examining how RL can be integrated to bolster their capabilities. This survey aims to serve as a comprehensive resource for researchers and practitioners interested in harnessing the power of RL for advancing code optimization techniques.

1 Introduction

As software systems grow more complex and development timelines shorten, manual code optimisation and development become impractical. Code generation and optimisation from Natural Language (NL) have thus become essential for boosting software development efficiency (Zhu et al., 2022; Allamanis et al., 2018). Compiler optimisations play a key role in enhancing software performance and reducing resource consumption. Meanwhile, advances in Natural Language Processing (NLP), particularly in Large Language Models (LLMs), have opened new possibilities for code generation. Conventional compiler optimisation relies on techniques like autotuning(Basu et al., 2013), while deep learning approaches for optimised compiler sequences (Li et al., 2020) struggle with generalisation, and although large-scale language models (LLMs) improve code generation(Cummins et al., 2023), they often produce biased or inconsistent outputs(Wang et al., 2023a; Barke et al., 2023), with code-specific models like Code T5(Wang et al., 2021) and Code T5+(Wang et al., 2023b) requiring time-consuming pre-training.

To address the challenges, Reinforcement learning(RL) are gaining attention. As a method that can learn the optimal strategy in complex environments, provides a new idea for compiler optimisation(Trofin et al., 2021), code generation(Le et al., 2022) and code optimisation(Bendib et al., 2024). Reinforcement learning allows label-free input-output pairs and it leverages existing knowledge, and refining strategies through trial and error. The benefits alleviate the dependency of pre-training in the code base and make LLMs adapting to dynamic changes in the environment. Figure 2 in the appendix A illustrates related work on reinforcement learning applied to compiler optimisation and code generation.

In order to promote more researchers benefit from the advances in RL field, we provide in this paper a systematic survey of RL on compiler optimisation and code generation problems. While many works have mentioned ways to enhance LLMs for code generation and optimisation, this paper focuses on the idea of how RL can be successful when applied to the field and provides a broader overview of RL related reviews. The paper is organised as follows.

- A brief introduction to the Reinforcement Learning for code optimisation in Section 2
- Analyse compiler optimisation of code in Section 3
- The role of reinforcement learning in resource allocation in Section 4
- The development of frameworks and tools for

code optimisation and generation with reinforcement learning Section 5

- Code generation and optimisation and analyse specific strategies in Section 6

- Related applications of reinforcement learning in Section 7

2 Reinforcement Learning(RL) in Code Optimisation

2.1 Objectives of Reinforcement Learning

Reinforcement learning seeks to discover an optimal policy parameter θ that maximizes cumulative reward through continuous interaction with the environment (Sutton, 2018). The probability distribution $\pi_\theta(a|s)$ represents the selection of action a given state s. Since the environment is often unknown or partially observable, directly calculating cumulative rewards is infeasible, so algorithms like Q-learning, strategy gradient, and actor-critic estimate cumulative rewards or gradients and update θ accordingly. This optimization process involves exploring θ through trial and error, aiming to lock in the optimal parameter for maximizing cumulative reward.

2.2 Sequence-to-Sequence Program Synthesis

In sequence-to-sequence program synthesis, the input sequence D represents the problem description, and the output sequence $\hat{W} = (\hat{w_1}, ..., \hat{w_T})$ represents the program solution, where $\hat{w}_t \in V^4$. Each decoding step t computes an output based on the softmax function: $\hat{W}_t \sim$softmax(Linear(s_t)), with s_t storing the decoding context. Training aims to maximize the probability of the ground-truth program $W = (w_1, ..., w_T)$ by minimizing the cross-entropy loss:

$$\begin{aligned} L_{ce}(\theta) &= -\sum_t \log P_\theta(W|D) \\ &= -\sum_t \log[P_\theta(w_t|w_{t-1}, D)] \end{aligned} \tag{1}$$

where P_θ is parameterized by the softmax function. During testing, the model autoregressively samples $\hat{w}_t$ from $p_\theta(\hat{w}_{1:t-1}, D)$ to generate the program, evaluated through unit tests matching the generated output to the ground-truth (Husain et al., 2019).

2.3 Trends in Reinforcement Learning-Based Code Optimization

Combined with the process of reinforcement learning code optimization, as shown in Figure 4 in the appendix. First, tasks are classified based on user input and their published status, then submitted to the agent, which selects the appropriate RL model optimization method. The agent continuously learns during the process, submits results for feedback, and adjusts the optimization method based on the feedback until the desired outcome is achieved. The direction of reinforcement learning code optimization can be summarized as follows:

- Model architecture: Code optimization can be achieved by integrating deep reinforcement learning with pre-trained language models to enhance the code generation framework (Wang et al., 2022), automating an adaptive code optimization framework (Zan et al., 2022; Ahn et al., 2020), or optimizing the compiler's structure to improve problem description and program generation (Cummins et al., 2022; Haj-Ali et al., 2020; Sun, 2022; Kovac et al., 2022; Shahzad et al., 2022). Additionally, the compilation sequence can be ordered sequentially (Jain et al., 2022; Almakki et al., 2022; Liang et al., 2023; Huang et al., 2019; Mammadli et al., 2020).

- Hardware, parallelization, and frameworks: Hardware resource allocation is crucial for code optimization, and reinforcement learning can allocate resources based on real-time usage (Kim et al., 2022; Kao et al., 2020; Doerfert and Finkel, 2018). While task processing was originally linear, parallelization now enables optimized code generation. The framework can provide a set of templates for code generation, which can be used in subsequent optimization of the same type of code (Le et al., 2022; Sun, 2022).

- Code generation and evaluation and feedback strategies: Code generation is the foundation of this article, and specific generation optimizations can also be started (Jiang et al., 2024a,b; Shen et al., 2023; Shojaee et al., 2023). After code generation, code training and feedback can also be used to optimize the code (Sun et al., 2024; Chen et al., 2024).

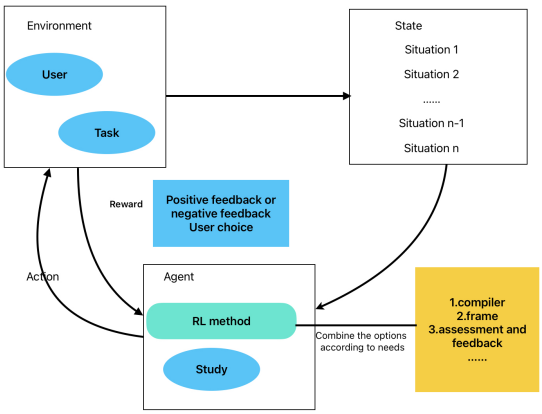

Figure 1: Reinforcement learning code is optimized for generation

3 Compiler Optimization Strategy

Compilers with advanced algorithms and intelligent mechanisms have transformed code optimization by enhancing program execution efficiency and resource utilization. The process involves parsing the source code, followed by data flow and control flow analysis, and applying techniques like loop optimization, inline unwinding, and dead code elimination. Recent advances include multi-stage optimizations, machine learning-assisted code optimization, and customized optimizations for specific hardware. Evaluating and benchmarking compiler optimization capabilities is also a key focus in this field. In the following sections, we will detail compiler code optimizations. Section 3.1 introduces data flow and control flow analysis techniques, essential for understanding and optimizing program behavior in RL code optimization. Section 3.2 discusses compiler flexibility and accuracy in code generation. Section 3.3 outlines the role of smart compilers in generating efficient machine code and applying various optimization techniques to enhance speed and resource utilization. Section 3.4 focuses on optimizing resource utilization through the sequencing of compiler compilation. The table below outlines the compiler's workflow.

3.1 Data Flow vs. Control Flow

In RL code optimization, data flow and control flow analysis play a crucial role. Originally from compiler optimization, these techniques are highly applicable to RL algorithm optimization, particularly in handling complex logic like policy updates and state transitions. Data flow analysis, which examines program data dependencies, can identify issues such as data competition, dead code, and constant folding in programs. In RL, data flow analysis is applied in several key areas: 1) State dependency analysis identifies redundant and key state variables, simplifying the state space and improving algorithm efficiency. 2) Reward function optimization detects defects, such as sparse or misleading rewards, for timely correction. 3) During policy gradient updates, data flow analysis helps identify errors and redundant calculations, enhancing the accuracy and efficiency of gradient updates. On the other hand, control flow analysis in RL offers several benefits: 1) It visualizes the policy update process, identifying potential iterative optimization by constructing control flow diagrams. 2) It enhances algorithm robustness by detecting anomalies and designing handling strategies. 3) Control flow also identifies parallelization opportunities, allowing for parallel computation of state value functions during policy evaluation and accelerating gradient calculations and parameter updates through parallel computing.

3.2 Compiler Usage and Code Generation

In code generation, compilers are crucial, prompting researchers to integrate reinforcement learning (RL) with compiler optimization. CompilerGym (Cummins et al., 2022), a tool specifically designed for AI research, makes compiler studies more accessible by providing an easy-to-use environment and benchmark sets, allowing researchers to experiment without needing to delve into the intricate details of the compiler internals. Through machine learning, CompilerGym optimizes performance by addressing key areas like code optimization and resource allocation, significantly improving compilation speed, code quality, and runtime efficiency. However, this approach depends heavily on specific compilers and machine learning libraries, which constrains its overall flexibility. To overcome these limitations, RL strategies have been introduced to optimize High-Level Synthesis (HLS) compilers (Huang et al., 2019), expanding existing learning strategies and evaluation metrics. These innovations provide greater flexibility in applying optimizations and ensure more reliable results by dynamically adapting to varying code structures and hardware configurations. The application of RL in HLS has allowed for the intelligent selection of optimization sequences and better handling of complex trade-offs in resource-constrained environments. Similarly, RL has been applied to

LLVM compiler optimization (Mammadli et al., 2020), where static analysis of source code is used to reduce instruction counts and execution times. RL enhances this by discovering optimal optimization sequences that traditional heuristics may miss. By leveraging parallelization techniques, this approach not only improves multi-core processor performance but also increases resource utilization and scalability. This combination of RL and compiler optimization represents a significant leap in software execution efficiency, system stability, and reliability, bringing substantial benefits to both the field of computer science and practical software engineering.

3.3 Intelligent Compiler

In the context of intelligent compilers, their role in managing massively parallel and heterogeneous computing environments is emphasized. It is proposed that intelligent compilers can automatically identify and optimize code, improving resource utilization and overcoming the limitations of manual tuning in traditional compilers (Kovac et al., 2022). This approach not only increases the versatility and flexibility of compilers but also reduces the technical barriers to professional compilation, potentially yielding significant economic and environmental benefits. Automated processes are expected to minimize manual intervention while maintaining code accuracy (Le et al., 2022). Notably, the CodeRL model demonstrates high-quality code generation across programming languages, enabling compilers to learn optimization patterns for generating more efficient code. This shows that deep learning can accelerate innovations in compiler optimization.

3.4 Compiler Sequence Sorting Optimization

Researchers have proposed innovative reinforcement learning-based methods for optimizing compiler sequence sorting, such as POSET-RL (Jain et al., 2022), which aims to find the optimal or near-optimal sorting through intelligent learning, optimizing both code size and execution time while demonstrating flexibility and scalability. In another approach, Autophase V2 (Almakki et al., 2022) focuses on function-level optimization, tailoring strategies for each function's features, reducing code length, and improving program efficiency and quality. By dynamically selecting optimal sequences, it overcomes the limitations of fixed optimization orders and enhances compiler adaptability. In addition, deep reinforcement learning-based compiler optimization methods (Sun, 2022) automate the determination of the order and parameters of compiler optimization phases, reducing the complexity of manual design. AutoPhase, for instance, uses deep reinforcement learning to solve the NP-hard problem of compiler stage ranking, achieving a 28% performance improvement over traditional methods with fewer samples (Huang et al., 2019), and its strong generalization makes it applicable across various programs, including real-world benchmarks. Furthermore, a deep reinforcement learning framework (Liang et al., 2023) has been proposed to automatically explore optimization sequences, improving both code size and execution time, particularly beneficial for resource-constrained environments such as embedded and mobile devices. The following figure classifies compiler code optimization techniques, and the accompanying table summarizes and compares each method.

4 The Role of Reinforcement Learning in Code Optimization in Resource Allocation

Register allocation assigns temporary variables to processor registers or memory, with registers offering faster access but limited availability. This limitation can lead to significant register pressure, especially in functions with many temporary variables. In extreme cases, some variables must overflow to memory, requiring expensive storage and loading instructions. The simplest method, spill-everywhere, generates spill code by inserting a store instruction after each definition of a temporary variable and a load instruction before each use. Figure 4 below illustrates this register allocation process. Register allocation employs a variety of program transformations to reduce the pressure on registers, minimize overflow, or reduce the overhead associated with overflow code when overflow is unavoidable. To this end, the researchers propose a redirect reinforcement learning (RL) strategy, which is tailored to the REAL problem (VenkataKeerthy et al., 2023), and the core lies in the multi-agent hierarchical optimization: 1) Finely divide and model register allocation sub tasks, covering coloring, range segmentation, overflow handling, etc.; 2) Cleverly encoded constraints to ensure that assignments are compatible with both program semantics and hardware specifications. The legitimacy and accuracy of the

assignment are maintained by precisely restricting the agent's behavior. Then, relying on the LLVM infrastructure, researchers developed the first end-to-end RL framework for register allocation. The LLVM Machine Intermediate Representation (MIR) is used to extract function interferograms, and the MIR2Vec embedding method provides application-independent characterizations of MIR entities. The LLVM-gRPC framework acts as a communication bridge between the RL model and the compiler, optimizing the training and deployment process to improve system portability. In this framework, the register allocation problem is treated as a single-player turn-based game, solved using the PBQP method with Deep-RL (Kim et al., 2022). By coloring the vertices of the PBQP graph, the system minimizes costs, using MCTS combined with DNN inference to intelligently select coloring schemes and collect training data to optimize the DNN. In practice, ConfuciuX performs a global coarse-grained search using the REINFORCE RL method, coupled with GA for fine-grained adjustments. It allocates DNN hardware resources effectively, applying the trained DNN directly to register allocation tasks (Kao et al., 2020).

5 Frameworks and Tools are Optimized for Code

In code optimization, selecting the right framework is crucial for improving efficiency and output performance. Framework optimization typically aims to reduce iteration steps while enhancing optimization efficiency and output. The Adam optimizer(Kingma and Ba, 2014), a first-order gradient-based algorithm, accelerates deep learning model training by combining the momentum method with the RMSprop optimizer to dynamically adjust each parameter's learning rate. It calculates the gradient using a first-order moment estimate (momentum) and a second-order moment estimate (uncentered variance), denoted as m_t and v_t. The first moment estimate m_t combines historical and current gradients through a weighted average, using the decay rate β_1.

$$m_t = \beta_1 \times m_{t-1} + (1 - \beta_1) \times \bigtriangledown \theta L(\theta_{t-1}) \quad (2)$$

The decay rate estimated by combining the second-order moments is β_2, and v_t is the weighted average of the squares of the gradient:

$$v_t = \beta_2 \times v_{t-1} + (1 - \beta_2) \times (\bigtriangledown \theta L(\theta_{t-1}))^2 \quad (3)$$

In order to compensate for the underestimation of bias in the initial stage of m_t and v_t, the Adam algorithm introduces a bias correction step, using the learning rate η, in order to prevent the error of dividing by 0, setting a minimum constant ϵ, obtained:

$$\theta_t = \theta_{t-1} - \eta \times \hat{m}_t / (\sqrt{\hat{v}_t + \epsilon}) \quad (4)$$

The corrected first-order moment estimate $\hat{m}_t$ and second-order moment estimate $\hat{v}_t$ enable adaptive learning rate adjustment. Researchers introduced CHAMELEON (Ahn et al., 2020), using reinforcement learning to adapt to new design spaces, converge in fewer iterations, and significantly improve optimization efficiency and output. Its adaptive sampling algorithm reduces costly hardware measurements and optimizes sample quality using domain knowledge, further accelerating optimization. In automatic program synthesis, methods based on large-scale pre-trained language models (LMs) show potential but are limited by the traditional supervised fine-tuning framework, which often overlooks key information like unit tests. To address this, the "CodeRL" framework (Le et al., 2022) integrates pre-trained LMs with deep reinforcement learning (RL). It treats the code-generation LM as an actor network and introduces a critic network to predict the functional accuracy of generated code, providing real-time feedback. Additionally, a key sampling-based iterative generation mechanism allows the model to optimize code using real-time feedback from unit tests and judges' scores during inference.

6 Code Generation and Optimization

6.1 Theoretical basis of code generation and optimization

Code generation involves the process of creating source code from a natural language description. Given a natural language description represented as a sequence $x = [x_1, \ldots, x_{|x|}]$, a Language Model (LM) p_{LM} predicts subsequent tokens in a sequential manner. At each step t, the LM determines the probability distribution of the next token based on the preceding tokens, denoted as $p_{LM}(x_t \mid x_{1:t-1})$. The probability of generating a program y with a token sequence $y = [x_{|x|+1}, \ldots, x_{|x|+|y|}]$ is derived from the product of these conditional probabilities:

$$P(y \mid x) = \prod_{t=|x|+1}^{|x|+|y|} p_{LM}(x_t \mid x_{1:t}) \quad (5)$$

In the context of few-shot learning with Large Language Models (LLMs), the generation process is often conditioned on a fixed set of m exemplars, denoted as $\{\langle x_i, y_i \rangle\}_{i \leq m}$. Consequently, the LLM-based code generation can be expressed as:

$$P_{LM}(y \mid x) = P\left(y \mid x, \{\langle x_i, y_i \rangle\}_{i \leq m}\right) \quad (6)$$

Code optimization involves the equivalent substitution of code to make it more time- and space-efficient. Local optimization targets areas of high time complexity to improve overall time performance, while global optimization addresses the entire code structure and execution process. With technological advancements, optimization occurs during the compilation process, covering both intermediate code optimization, which focuses on code structure, and object code optimization, which converts intermediate code into efficient machine code. Another method is dynamic optimization during program execution. Finally, algorithm and data structure optimization significantly reduces computational and spatial complexity.

6.2 Specific strategies for code generation and optimization

In the field of code generation, large language models have become more proficient, and through a comprehensive study of large model languages (LLMs), these models are dedicated to solving the task of generating code from natural language descriptions (NL2Code). Through an intuitive comparison method, all existing models were evaluated on the HumanEval(Chen et al., 2021) benchmark, and in-depth observation and analysis were used to conclude that the key factors for the success of the model were "large size, good quality data, expert tuning"(Zan et al., 2022).Regarding the cutting-edge progress of LLM code generation, we use a taxonomy to systematically classify and discuss the latest developments in the field of LLM code generation, covering data management, technical progress, performance evaluation, and practical applications. In addition, the evolution of LLM in code generation capabilities is discussed in depth, and empirical comparisons are made using widely recognized benchmarks such as HumanEval and MBPP(Austin et al., 2021) to visually demonstrate the gradual enhancement of LLM code generation capabilities(Jiang et al., 2024a). In the RL-based approach, a new RRTF framework optimization paradigm is pointed out for the development of LLMs, and then PanGu-Coder2 is introduced, which stands out among all code LLMs(Shen et al., 2023).

6.3 Evaluation and detection of optimized code

The core step to ensure the effectiveness of optimized code is a comprehensive evaluation and feedback system, covering multiple dimensions from execution-stage feedback to user experience. Runtime feedback from the compiler or interpreter verifies code correctness and accuracy, while automatically generated test cases ensure both syntax and functional logic correctness. Performance evaluation focuses on execution efficiency, analyzing time and spatial complexity to reveal optimization potential. Security and robustness are assessed by simulating threats to ensure stable operation. Automated code review tools enforce best practices, offering recommendations on quality and consistency. User feedback is crucial for evaluating code optimization, directly impacting user satisfaction. Benchmarking against industry standards provides an objective assessment of code performance and quality. Automated error detection tools instantly flag potential issues, offering fixes, while tools to improve code style enhance maintainability and team collaboration. For code generation models supporting multiple languages, cross-language performance and accuracy are critical, ensuring compatibility and flexible deployment options. By training datasets with SFT and RL, we address data pollution and evaluate the generalization ability and performance impact of SFT on RL. Moreover, during SFT checkpoint initialization, RL can significantly boost the model's generalization to target domains (Chen et al., 2024).

7 Related Applications of Reinforcement Learning

What is the essence of reinforcement learning and how it is applied to code generation and optimization? In this section, we show the application of reinforcement learning.

7.1 Direct application of reinforcement learning

Within the purview of reinforcement learning, two prominent approaches stand out for their direct applicability: value-based and strategy-based methods. Value-based methods focus on learning the

value of being in a particular state or taking a specific action. Q-learning, a model-free, off-policy algorithm, is a classic value-based approach, ideal for coding tasks with compact state and action spaces. It uses the Bellman equation to approximate the optimal action-value function, or Q-function, and iteratively updates the Q-table to converge on an effective policy (Carroll et al., 2001). Deep Q-Networks (DQN) extend Q-learning into deep learning, addressing high-dimensional state and continuous action spaces (Mnih et al., 2015). By using deep neural networks to approximate the Q-function, DQN enables efficient handling of complex problems that traditional Q-learning might struggle with (Ohnishi et al., 2019). Policy-based methods focus on directly learning the optimal policy by optimizing the policy function, making them ideal for continuous action spaces and differentiable code representations (Alibekov et al., 2018). Policy Gradient is one such method, enabling the algorithm to generate efficient, executable code by optimizing the policy (Papini et al., 2019). Proximal Policy Optimization (PPO), an extension of Policy Gradient, limits the size of policy updates, ensuring stability and robustness in coding tasks requiring high reliability. A key advantage of PPO is its ability to leverage the unique sequence-level characteristics of code. Most proposed methods extend PPO in two areas: a) designing reward strategies to obtain higher-granularity feedback from compilers (Gu et al., 2022), and b) addressing challenges like inefficiency, instability, high resource demands, and complex hyperparameter tuning in reinforcement learning (Zhu and Rosendo, 2021). PPOCoder integrates the pre-trained CodeT5 model with Proximal Policy Optimization (PPO), using execution feedback to assess syntactic and functional correctness. It also incorporates a reward function to evaluate syntactic and semantic consistency between Abstract Syntax Tree (AST) subtrees, Data Flow Graph (DFG) edges, and ground truths. RLTF further refines this by providing fine-grained feedback based on compiler errors and adaptive feedback scaled by test cases (Yang et al., 2021). To tackle the time complexity issues inherent in traditional RL methods like PPO, newer approaches such as DPO and RRHF have emerged. These methods shift the focus toward maximizing the likelihood between preferred and rejected responses, offering a more efficient alternative to PPO. By leveraging RRHF (Ranking Reinforcement from Human Feedback), PanGu-Coder 2 demonstrates strong potential in code generation tasks (Pinsler et al., 2018). This innovative use of RRHF not only optimizes the learning process but also enhances the consistency and quality of code generation across various models, positioning it as a promising solution for complex coding tasks.

7.2 Applications of Meta-learning

Meta-learning is a powerful machine learning approach designed to enable models to quickly adapt to new tasks with minimal gradient updates. In the field of code generation, this method is particularly useful, especially when dealing with domain-specific programming languages that may not have a large number of training samples (Chai et al., 2022). Recent research has proposed a new method called MetaCoder(Yang et al., 2024), which uses the Model-Agnostic Meta-Learning (MAML) algorithm to construct a transfer learning framework. This framework first learns general knowledge from large-scale source languages and then applies it to target languages in specific domains. The core idea of MetaCoder is to leverage MAML to extract general knowledge from large-scale source languages and quickly adapt to situations in specific domains, even when there are relatively few samples.

7.3 The Impact of Reinforcement Learning on End-to-end Software Development

Reinforcement learning (RL) is transforming software engineering, enhancing various stages from design to deployment (Zhuang et al., 2021). RL's ability to learn optimal actions through interactions with an environment is driving automation in areas like code suggestion and generation, speeding up development and reducing human error. In software testing, RL automates test case generation, execution order, and process optimization, improving test quality and coverage. In network control, RL enhances traffic management and control strategies, vital for distributed systems (Xiao et al., 2021). As RL matures, it is expected to play an increasingly integral role in end-to-end software development, from testing automation to resource management and code generation, becoming a key tool for software engineers.

7.4 Advanced Applications and Challenges of Reinforcement Learning

Reinforcement learning (RL) excels at maximizing rewards, making it a key tool in embodied intel-

ligence. In multi-objective reinforcement learning (MORL), RL addresses problems with multiple goals, like optimizing speed while minimizing fuel consumption in autonomous driving. MORL learns a set of policies to accommodate user preferences, selecting the best strategy based on these preferences during testing. In continual learning (CL), RL adapts to evolving objectives over time. For instance, Continual Multi-Objective Reinforcement Learning (CMORL) addresses shifting objectives through reward model rehearsal (Li et al., 2024). Despite RL's potential in software development, developers face challenges. A large-scale study of 927 DRL-related posts on Stack Overflow highlighted difficulties with comprehension, API usage, and design issues (Morovati et al., 2024). Parallel processing and DRL libraries/frameworks were the most challenging, and RL's high computational complexity complicates its application in low-latency systems.

8 Prospects for Future Development

Through the above discussion, with the continuous development of AI intelligence, there are many future directions for RL code optimization. In the register allocation part, reinforcement learning algorithms can be introduced to realize the intelligent allocation of register resources(Gao et al., 2021; Elthakeb et al., 2018; Paliwal et al., 2019; Ahn et al., 2019; Chen and Chung, 2022) to reduce register conflicts and overflows, and improve the utilization of CPU registers. Improve program execution efficiency by optimizing the lifetime management of variables and using reuse registers to reduce unnecessary loading and storage operations. In the direction of compiler optimization, the MDP-based polyhedral optimization environment(Brauckmann et al., 2021; Haj-Ali et al., 2019) can break through the dependence of traditional methods on specific cyclic shapes and innovatively point out shape-independent optimization formulas. On this basis, a medium-scale search space is constructed, which avoids the complexity of large spaces and transcends the limitations of small spaces, as a new path for efficient compiler optimization. In (Zhang et al., 2023), it was noted that existing benchmarks no longer fully capture real-world programming tasks, and future research must address more complex software engineering activities. The evaluation of large language models (LLMs) should extend beyond traditional code generation to encompass all phases of software engineering. Key research directions include acquiring high-quality data, integrating code features into LLMs, exploring non-imperative paradigms, and developing alternative architectures and training goals. As LLM capabilities grow, issues such as data pollution, content bias, and information leakage must also be addressed in software engineering. At the same time, (Ganai et al., 2023; Pecenin et al., 2019) noted that current research focuses on reducing the number of XLA operands to optimize performance, but future work should extend to goals such as execution speed and explore methods that incorporate hardware dependency, such as automatic tuning. Graph-based program representations, like ProGraML, could be used to capture valuable information, and larger training datasets are essential to avoid overfitting and improve model generalization. The introduction of domain-specific knowledge, such as HLO cost analysis, and continuous improvements to reward functions, value functions, and RL algorithms are expected to further enhance optimization. In the field of Advanced Synthesis (HLS) (Abo-eleneen et al., 2023), four new RL strategies were proposed and evaluated using indicators such as learning speed, performance potential, and fluctuation bands. The article emphasizes that adapting RL strategies to specific applications improves learning speed and performance. Future research will explore more strategies, evaluate them for various applications, simplify developer adjustments, and enhance compiler optimization, minimizing reliance on labeled data and facilitating cross-platform migration.

9 Conclusion

In this paper, we discuss early reinforcement learning (RL) approaches to code optimization and analyze various RL-based strategies in different optimization directions. We examine the commonalities and differences of these methods, arguing that RL holds significant promise for code optimization, potentially marking a major shift in the field. Our goal is to help researchers gain a comprehensive understanding of the possible directions for RL-based code optimization and the core challenges in each area, inspiring future advancements and progress in this evolving field.

10 Limitations

In this comprehensive survey, we have examined the application of reinforcement learning in code generation and optimization, analyzing a range of methods and techniques. However, due to space limitations, we have not provided a comprehensive analysis of all aspects under discussion. Firstly, we did not provide a detailed account of the datasets employed for model training, which are of paramount importance for the model's generalization and performance. Further research could examine the influence of disparate datasets on model performance and the construction of more diverse and representative datasets to improve generalization. Secondly, the scalability and generalization of reinforcement learning models in the context of large-scale codebases and across multiple projects were not discussed. In practical applications, models must be capable of handling codebases of varying scales and complexities, necessitating good scalability and adaptability. Further research could concentrate on improving the scalability of the models and the transfer of learning between disparate projects and programming languages. Lastly, a detailed comparison of the training and inference times of various algorithms was not provided. In the context of software development, the efficiency of the algorithm is of paramount importance. Consequently, future studies could assess the time complexity of different reinforcement learning algorithms during the training and inference phases, with the aim of optimizing these times.

References

Amr Abo-eleneen, Ahammed Palliyali, and Cagatay Catal. 2023. The role of reinforcement learning in software testing. *Information and Software Technology*, 164:107325.

Byung Hoon Ahn, Prannoy Pilligundla, and Hadi Esmaeilzadeh. 2019. Reinforcement learning and adaptive sampling for optimized dnn compilation. *arXiv preprint arXiv:1905.12799*.

Byung Hoon Ahn, Prannoy Pilligundla, Amir Yazdanbakhsh, and Hadi Esmaeilzadeh. 2020. Chameleon: Adaptive code optimization for expedited deep neural network compilation. *arXiv preprint arXiv:2001.08743*.

E. Alibekov, Jiří Kubalík, and Robert Babuška. 2018. Policy derivation methods for critic-only reinforcement learning in continuous spaces. *Eng. Appl. Artif. Intell.*, 69:178–187.

Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Comput. Surv.*, 51(4).

Mohammed Almakki, Ayman Izzeldin, Qijing Huang, Ameer Haj Ali, and Chris Cummins. 2022. Autophase v2: Towards function level phase ordering optimization. In *Machine Learning for Computer Architecture and Systems 2022*.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Shraddha Barke, Michael B. James, and Nadia Polikarpova. 2023. Grounded copilot: How programmers interact with code-generating models. *Proc. ACM Program. Lang.*, 7(OOPSLA1).

Protonu Basu, Mary Hall, Malik Khan, Suchit Maindola, Saurav Muralidharan, Shreyas Ramalingam, Axel Rivera, Manu Shantharam, and Anand Venkat. 2013. Towards making autotuning mainstream. *Int. J. High Perform. Comput. Appl.*, 27(4):379–393.

Nazim Bendib, Iheb Nassim Aouadj, and Riyadh Baghdadi. 2024. A reinforcement learning environment for automatic code optimization in the mlir compiler.

Alexander Brauckmann, Andrés Goens, and Jeronimo Castrillon. 2021. A reinforcement learning environment for polyhedral optimizations. *arXiv preprint arXiv:2104.13732*.

J. Carroll, T. Peterson, and N. Owens. 2001. Memory-guided exploration in reinforcement learning. *IJCNN'01. International Joint Conference on Neural Networks. Proceedings (Cat. No.01CH37222)*, 2:1002–1007 vol.2.

Yitian Chai, Hongyu Zhang, Beijun Shen, and Xiaodong Gu. 2022. Cross-domain deep code search with meta learning. *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pages 487–498.

Jie Chen, Xintian Han, Yu Ma, Xun Zhou, and Liang Xiang. 2024. Unlock the correlation between supervised fine-tuning and reinforcement learning in training code large language models. *arXiv preprint arXiv:2406.10305*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen

Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code.

Weibin Chen and Yeh-Ching Chung. 2022. Profile-guided optimization for function reordering: A reinforcement learning approach. In *2022 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 2326–2333. IEEE.

Chris Cummins, Volker Seeker, Dejan Grubisic, Mostafa Elhoushi, Youwei Liang, Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Kim M. Hazelwood, Gabriel Synnaeve, and Hugh Leather. 2023. Large language models for compiler optimization. *ArXiv*, abs/2309.07062.

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)*, pages 92–105. IEEE.

Johannes Doerfert and Hal Finkel. 2018. Compiler optimizations for parallel programs. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 112–119. Springer.

Ahmed T Elthakeb, Prannoy Pilligundla, FatemehSadat Mireshghallah, Amir Yazdanbakhsh, and Hadi Esmaeilzadeh. 2018. Releq: A reinforcement learning approach for deep quantization of neural networks. *arXiv preprint arXiv:1811.01704*.

Milan Ganai, Haichen Li, Theodore Enns, Yida Wang, and Randy Huang. 2023. Target-independent xla optimization using reinforcement learning. *arXiv preprint arXiv:2308.14364*.

Chao Gao, Tong Mo, Taylor Zowtuk, Tanvir Sajed, Laiyuan Gong, Hanxuan Chen, Shangling Jui, and Wei Lu. 2021. Bansor: Improving tensor program auto-scheduling with bandit based reinforcement learning. In *2021 IEEE 33rd International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 273–278. IEEE.

Yangyang Gu, Yuhu Cheng, C. L. P. Chen, and Xuesong Wang. 2022. Proximal policy optimization with policy feedback. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 52:4600–4610.

Ameer Haj-Ali, Nesreen K Ahmed, Ted Willke, Joseph Gonzalez, Krste Asanovic, and Ion Stoica. 2019. A view on deep reinforcement learning in system optimization. *arXiv preprint arXiv:1908.01275*.

Ameer Haj-Ali, Nesreen K Ahmed, Ted Willke, Yakun Sophia Shao, Krste Asanovic, and Ion Stoica. 2020. Neurovectorizer: End-to-end vectorization with deep reinforcement learning. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, pages 242–255.

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)*, pages 308–308. IEEE.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*.

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)*, pages 121–131. IEEE.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024a. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*.

Xue Jiang, Yihong Dong, Zhi Jin, and Ge Li. 2024b. Seed: Customize large language models with sample-efficient adaptation for code generation. *arXiv preprint arXiv:2403.00046*.

Sheng-Chun Kao, Geonhwa Jeong, and Tushar Krishna. 2020. Confuciux: Autonomous hardware resource assignment for dnn accelerators using reinforcement learning. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 622–636. IEEE.

Minsu Kim, Jeong-Keun Park, and Soo-Mook Moon. 2022. Solving pbqp-based register allocation using deep reinforcement learning. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–12. IEEE.

Diederik P. Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980.

Mihael Kovac, Mario Brcic, Agneza Krajna, and Dalibor Krleza. 2022. Towards intelligent compiler optimization. In *2022 45th Jubilee International Convention on Information, Communication and Electronic Technology (MIPRO)*, pages 948–953. IEEE.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. 2022. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems*, 35:21314–21328.

Lihe Li, Ruotong Chen, Ziqian Zhang, Zhichao Wu, Yi-Chen Li, Cong Guan, Yang Yu, and Lei Yuan. 2024. Continual multi-objective reinforcement learning via reward model rehearsal. In *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence (IJCAI-24)*, pages 4434–4442. National Key Laboratory for Novel Software Technology, Nanjing University.

Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, and Depei Qian. 2020. The deep learning compiler: A comprehensive survey. *IEEE Transactions on Parallel and Distributed Systems*, 32:708–727.

Youwei Liang, Kevin Stone, Ali Shameli, Chris Cummins, Mostafa Elhoushi, Jiadong Guo, Benoit Steiner, Xiaomeng Yang, Pengtao Xie, Hugh James Leather, et al. 2023. Learning compiler pass orders using coreset and normalized value prediction. In *International Conference on Machine Learning*, pages 20746–20762. PMLR.

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)*, pages 1–11. IEEE.

Volodymyr Mnih, K. Kavukcuoglu, David Silver, Andrei A. Rusu, J. Veness, Marc G. Bellemare, Alex Graves, Martin A. Riedmiller, A. Fidjeland, Georg Ostrovski, Stig Petersen, Charlie Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, D. Kumaran, Daan Wierstra, S. Legg, and D. Hassabis. 2015. Human-level control through deep reinforcement learning. *Nature*, 518:529–533.

Mohammad Mehdi Morovati, Florian Tambon, Mina Taraghi, Amin Nikanjam, and Foutse Khomh. 2024. Common challenges of deep reinforcement learning applications development: an empirical study. *Empirical Softw. Engg.*, 29(4).

Shotaro Ohnishi, E. Uchibe, Yotaro Yamaguchi, Kosuke Nakanishi, Y. Yasui, and S. Ishii. 2019. Constrained deep q-learning gradually approaching ordinary q-learning. *Frontiers in Neurorobotics*, 13.

Aditya Paliwal, Felix Gimeno, Vinod Nair, Yujia Li, Miles Lubin, Pushmeet Kohli, and Oriol Vinyals. 2019. Reinforced genetic algorithm learning for optimizing computation graphs. *arXiv preprint arXiv:1905.02494*.

M. Papini, Matteo Pirotta, and Marcello Restelli. 2019. Smoothing policies and safe policy gradients. *Machine Learning*, 111:4081 – 4137.

Marcelo Pecenin, André Murbach Maidl, and Daniel Weingaertner. 2019. Optimization of halide image processing schedules with reinforcement learning. In *Simpósio em Sistemas Computacionais de Alto Desempenho (SSCAD)*, pages 37–48. SBC.

Robert Pinsler, R. Akrour, Takayuki Osa, Jan Peters, and G. Neumann. 2018. Sample and feedback efficient hierarchical reinforcement learning from human preferences. *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 596–601.

Hafsah Shahzad, Ahmed Sanaullah, Sanjay Arora, Robert Munafo, Xiteng Yao, Ulrich Drepper, and Martin Herbordt. 2022. Reinforcement learning strategies for compiler optimization in high level synthesis. In *2022 IEEE/ACM Eighth Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*, pages 13–22. IEEE.

Bo Shen, Jiaxin Zhang, Taihong Chen, Daoguang Zan, Bing Geng, An Fu, Muhan Zeng, Ailun Yu, Jichuan Ji, Jingyang Zhao, et al. 2023. Pangu-coder2: Boosting large language models for code with ranking feedback. corr abs/2307.14936 (2023). *arXiv preprint arXiv:2307.14936*, 10.

Parshin Shojaee, Aneesh Jain, Sindhu Tipirneni, and Chandan K Reddy. 2023. Execution-based code generation using deep reinforcement learning. *arXiv preprint arXiv:2301.13816*.

Qiushi Sun, Zhirui Chen, Fangzhi Xu, Kanzhi Cheng, Chang Ma, Zhangyue Yin, Jianing Wang, Chengcheng Han, Renyu Zhu, Shuai Yuan, et al. 2024. A survey of neural code intelligence: Paradigms, advances and beyond. *arXiv preprint arXiv:2403.14734*.

Yilin Sun. 2022. Optimizing llvm pass list using reinforcement learning.

Richard S Sutton. 2018. Reinforcement learning: An introduction. *A Bradford Book*.

Mircea Trofin, Yundi Qian, Eugene Brevdo, Zinan Lin, Krzysztof Choromanski, and David Xinliang Li. 2021. Mlgo: a machine learning guided compiler optimizations framework. *ArXiv*, abs/2101.04808.

S VenkataKeerthy, Siddharth Jain, Anilava Kundu, Rohit Aggarwal, Albert Cohen, and Ramakrishna Upadrasta. 2023. Rl4real: Reinforcement learning for register allocation. In *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction*, pages 133–144.

Huanting Wang, Zhanyong Tang, Cheng Zhang, Jiaqi Zhao, Chris Cummins, Hugh Leather, and Zheng Wang. 2022. Automating reinforcement learning architecture design for code optimization. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*, pages 129–143.

Shiqi Wang, Zheng Li, Haifeng Qian, Chenghao Yang, Zijian Wang, Mingyue Shang, Varun Kumar, Samson Tan, Baishakhi Ray, Parminder Bhatia, Ramesh Nallapati, Murali Krishna Ramanathan, Dan Roth, and Bing Xiang. 2023a. ReCode: Robustness evaluation of code generation models. In *Proceedings of the 61st Annual Meeting of the Association for*

Computational Linguistics (Volume 1: Long Papers), pages 13818–13843, Toronto, Canada. Association for Computational Linguistics.

Yue Wang, Hung Le, Akhilesh Gotmare, Nghi Bui, Junnan Li, and Steven Hoi. 2023b. CodeT5+: Open code large language models for code understanding and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 1069–1088, Singapore. Association for Computational Linguistics.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Yang Xiao, Jun Liu, Jiawei Wu, and N. Ansari. 2021. Leveraging deep reinforcement learning for traffic engineering: A survey. *IEEE Communications Surveys & Tutorials*, 23:2064–2097.

Yang Yang, Chaoyue Pan, Zheng Li, and Ruilian Zhao. 2021. Adaptive reward computation in reinforcement learning-based continuous integration testing. *IEEE Access*, 9:36674–36688.

Zhen Yang, Jacky Wai Keung, Zeyu Sun, Yunfei Zhao, Ge Li, Zhi Jin, Shuo Liu, and Yishu Li. 2024. Improving domain-specific neural code generation with few-shot meta-learning. *Information and Software Technology*, 166:107365.

Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Yongji Wang, and Jian-Guang Lou. 2022. Large language models meet nl2code: A survey. *arXiv preprint arXiv:2212.09420*.

Ziyin Zhang, Chaoyu Chen, Bingchang Liu, Cong Liao, Zi Gong, Hang Yu, Jianguo Li, and Rui Wang. 2023. A survey on language models for code. *arXiv preprint arXiv:2311.07989*.

Qingfu Zhu, Xianzhen Luo, Fang Liu, Cuiyun Gao, and Wanxiang Che. 2022. A survey on natural language processing for programming. abs/2212.05773:1690–1704.

Wangshu Zhu and A. Rosendo. 2021. A functional clipping approach for policy optimization algorithms. *IEEE Access*, 9:96056–96063.

Peiwen Zhuang, Tao Xu, and Shan Wang. 2021. A reinforcement-learning-based deployment strategy for gpp components. *Proceedings of the 9th International Conference on Computer and Communications Management*.

A Appendix

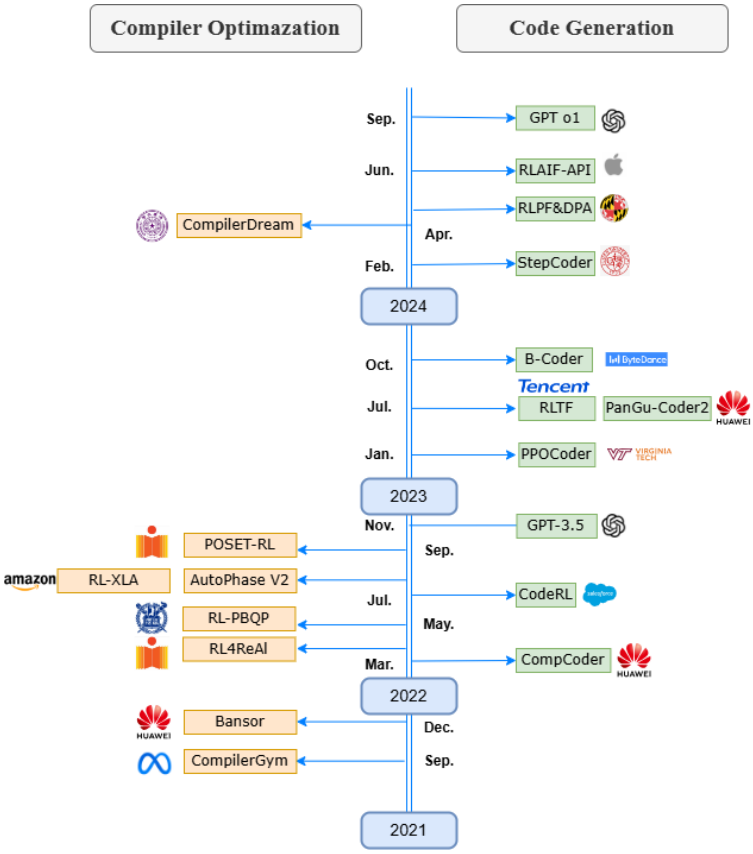

Figure 2: The diagram illustrates the development of code optimization and generation technologies from 2021 to 2024, covering key projects like CompilerDream, the GPT series, and contributions from various organizations.

Process steps	Concrete implementation	Examples of Methods (Functions)
Lexical analysis	The compiler breaks down the source code into a series of lexical units	Keywords, identifiers, operators, and so on. The lexical analyzer scans the source code, identifies these lexical units, and generates a flow of lexical units for subsequent steps.
Grammatical analysis	The compiler analyzes the flow of lexical units according to predefined syntax rules and constructs a syntax tree.	A syntax tree is a tree structure based on syntax rules, which represents the structure and semantics of the code. The parser checks if the code conforms to the grammar rules.
Semantic analysis	The compiler analyzes the syntax tree to check if the semantics are correct.	Check if the variables have been declared, if the function call is correct, and so on. Semantic analyzers are typically implemented using methods such as symbol tables, type checking, and so on.
Intermediate code generation	The compiler converts the syntax tree into intermediate code that can be used for subsequent code optimization and object code generation.	Intermediate code is a representation between source and object code that makes it easier for compilers to optimize code and compile across platforms.
Code optimization	The compiler optimizes the intermediate code to improve the execution efficiency and performance of the code.	Code optimizers are typically implemented using techniques such as constant folding, loop expansion, instruction reflow, and so on.
Object code generation	The compiler converts the optimized intermediate code into target machine code.	Object code is binary instructions that can be executed directly by a computer, and they are packaged into executable files or libraries in order to run a program on a computer.

Table 1: Compiler code processing

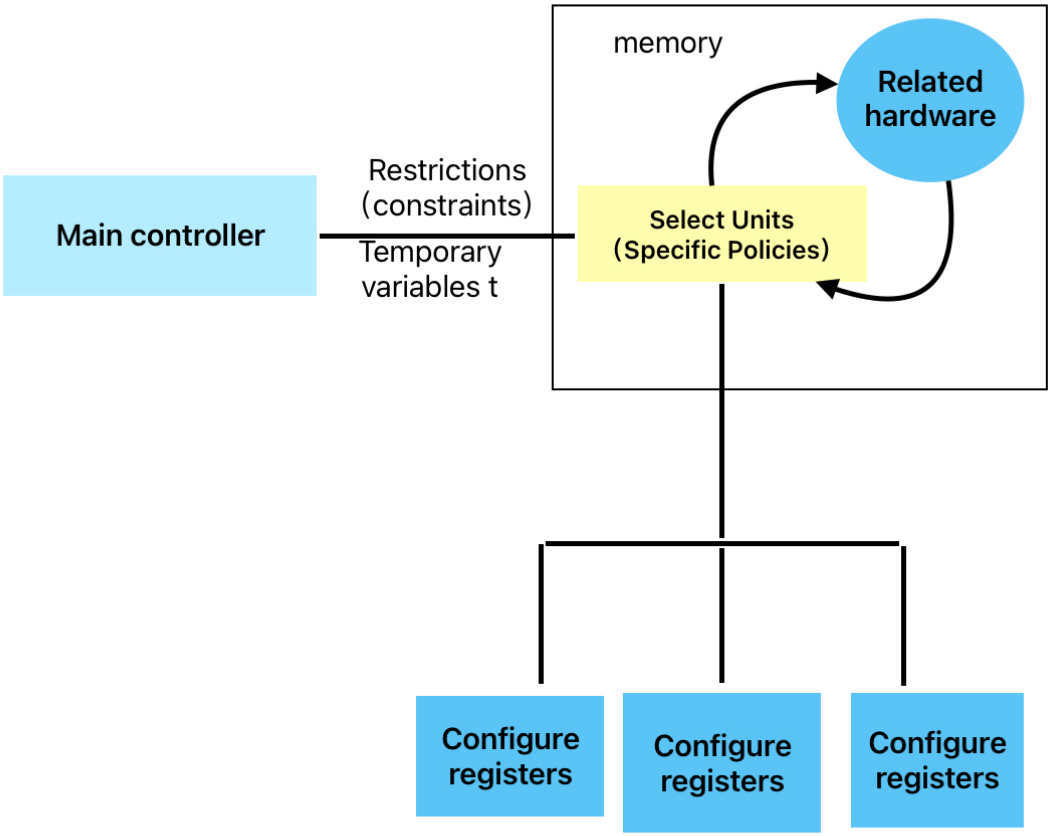

Figure 3: Register allocation method

Table 2: Performance of Base, Reinforcement Learning-based, and Multi Reinforcement Learning-based Models on **APPS** benchmark

Model	**Size**	**Pass@1**				**Pass@5**				**Pass@1000**			
		Intro	Inter	Comp	All	Intro	Inter	Comp	All	Intro	Inter	Comp	All
Base Models													
Codex	12B	4.14	0.14	0.02	0.92	9.65	0.51	0.09	2.25	25.02	3.70	3.23	7.87
AlphaCode	1B	-	-	-	-	-	-	-	-	17.67	5.24	7.06	8.09
GPT-3	175B	0.20	0.03	0.00	0.06	2.70	0.73	0.00	1.02	-	-	-	-
GPT-2	0.1B	1.00	0.33	0.00	0.40	3.60	1.03	0.00	1.34	-	-	-	-
GPT-2	1.5B	1.30	0.70	0.00	0.68	5.50	1.03	0.00	1.58	27.90	9.27	8.80	12.32
GPT-Neo	2.7B	3.90	0.57	0.00	1.12	5.50	0.80	0.00	1.58	27.90	9.83	11.40	13.76
GPT-J	6B	5.60	1.00	0.50	1.82	9.20	1.73	1.00	3.08	35.20	13.15	13.51	17.63
CodeT5†	60M	1.40	0.67	0.00	0.68	2.60	0.87	0.10	1.06	-	-	-	-
CodeT5†	220M	2.50	0.73	0.00	0.94	3.30	1.10	0.10	1.34	-	-	-	-
CodeT5†	770M	3.60	0.90	0.20	1.30	4.30	1.37	0.20	1.72	-	-	-	-
Reinforcement Learning-based Models													
CodeRL	770M	6.20	1.50	0.30	2.20	9.39	1.90	0.42	3.10	35.30	13.33	13.60	17.78
PPOCoder	770M	5.20	1.00	0.50	1.74	9.10	2.50	1.20	3.56	35.20	13.35	13.90	17.77
RLTF	770M	4.16	0.97	0.20	1.45	10.12	2.65	0.82	3.78	38.30	15.13	15.90	19.92
$\mathcal{B}$-Coder	$\leq$770M/stage[3]	6.70	1.50	0.30	2.30	10.40	2.63	0.70	3.80	37.00	13.67	12.60	18.12
Multi Reinforcement Learning- based Models													
CodeRL + CodeT5	770M	4.90	1.06	0.5	1.71	8.60	2.64	1.0	3.51	36.10	12.65	13.48	17.50
PPOCoder + CodeT5	770M	5.20	1.00	0.5	1.74	9.10	2.50	1.20	3.56	35.20	13.35	13.90	17.77

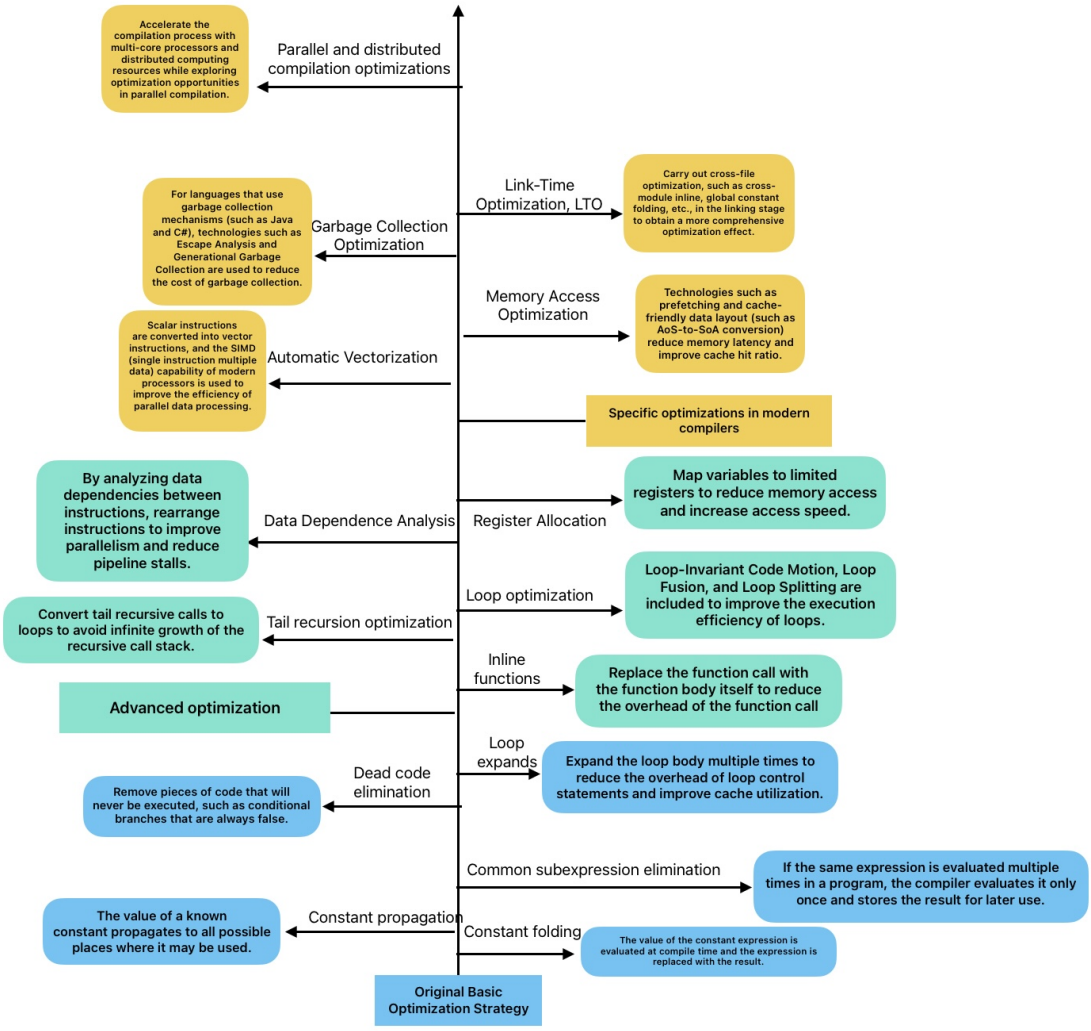

Figure 4: Compiler Code Optimization Technology Classification

method	Resource consumption	Generalization ability	Degree of automation
(Huang et al., 2019)	high	high	high
(Mammadli et al., 2020)	high	high	extremely high
(Sun, 2022)	high	high	high
(Kovac et al., 2022)	high	high	——
(Shahzad et al., 2022)	high	low	——
(Liang et al., 2023)	——	low	high
(Almakki et al., 2022)	high	high	——
(Haj-Ali et al., 2020)	high	high	——
(Cummins et al., 2022)	high	low	——
(Brauckmann et al., 2021)	high	——	——
(Jain et al., 2022)	high	high	high

Table 3: Compiler optimization methods are compared