Computing and Applied Sciences Impact (2024), 1(1): 36–51
ISSN: 3043-680X

Tutorial Review

Reinforcement learning models to optimize compiler phase ordering for specific applications: review of selected cases

Otobong Anietie Udoh [1]* and Peace Okafor [2]

1 Department of Computer and Robotics Education, Faculty of Vocational Education, Library and Information Science, University of Uyo, Uyo, Nigeria
2 Department of State Service, Ebonyi State Command, Abakaliki, Nigeria

*Corresponding author: Otobong Anietie Udoh (otobonganietieudoh@gmail.com)

Received: September 30, 2024; Accepted: December 28, 2024; Published: December 30, 2024

Abstract

In this study, we reviewed the applications of reinforcement learning (RL) models to optimize compiler phase ordering, a crucial aspect of compiler optimization. The study examines several prominent RL-based models, including Machine Learning Guided Optimization (MLGO), Autophase, DeepTune, NeuroVectorizer, and COBAYN, highlighting their key contributions, methodologies, limitations, and potential improvements. While RL-based approaches have demonstrated significant advancements in optimizing compiler tasks, such as phase ordering and loop vectorization, the review identifies common challenges, including task-specific optimization, dependency on predefined sequences, limited adaptability, and lack of interpretability. The paper also discusses the gaps in the existing literature, emphasizing the need for more generalizable models, dynamic learning capabilities, and enhanced transparency in optimization decisions. Future research should focus on developing scalable, adaptable, and interpretable RL models that can seamlessly integrate into modular compiler frameworks, paving the way for more efficient and adaptive compiler optimization techniques in real-world applications.

Keywords: Reinforcement learning model; compiler; phase ordering; optimization; applications

1. Introduction

In compiler optimization, phase ordering refers to the problem of determining the optimal sequence of transformation passes or phases that a compiler should apply to the source code in order to generate the most efficient machine code [1]. Modern compilers, such as low-level virtual machine (LLVM) or GNU compiler collection (GCC), offer a variety of optimization passes that can be applied during compilation. However, the order in which these phases are executed significantly impacts the performance of the generated code. There is no universally optimal order for all programs due to the complex interactions between different passes and the specific characteristics of the target hardware and application [2].

Some studies [3], [4], have addressed this critical issue by introducing a novel reinforcement learning model designed to revolutionize compiler optimization. Unlike static methods, the reinforcement learning model learns dynamically, tailoring optimization strategies to each unique workload and hardware configuration. The problem at hand is clear: How can we design compilers that autonomously optimize code for maximum efficiency in an ever-changing environment? By integrating reinforcement learning into compiler design, we aim to solve this challenge and unlock new levels of performance, offering a flexible, adaptive solution that continuously evolves with software demands [5].

The specific objectives for this review on reinforcement learning (RL) models for optimizing compiler phase ordering were to:

i. examine the methodologies, contributions, and performance of RL-based models, such as MLGO, Autophase, DeepTune, NeuroVectorizer, and COBAYN, in optimizing compiler phase ordering and related tasks;
ii. highlight the recurring challenges, such as task-specific optimization, dependency on predefined sequences, limited adaptability, and lack of interpretability, to understand barriers to effective compiler optimization using RL; and
iii. suggest strategies to address gaps in existing RL models, such as developing generalizable frameworks, improving scalability and adaptability, and enhancing model interpretability for better real-world integration. Establish a systematic evaluation framework to compare RL-based models across key dimensions, including contributions, methodologies, limitations, and areas for improvement, to provide a comprehensive understanding of their capabilities and shortcomings.

2. Reinforcement Learning in Compiler Optimization

The integration of reinforcement learning into compiler optimization will result in significant performance improvements, as the adaptive model will continuously evolve to make better optimization decisions than traditional static heuristics, particularly when handling diverse and complex workloads using some specific applications [6], [7], [8], [9], [10]. Recent research on compiler optimization has seen a growing interest in machine learning approaches, especially reinforcement learning, as a way to enhance traditional techniques. These advancements aim to address the limitations of static heuristics, which struggle to adapt to the complexity of modern software and hardware environments. In this critical review, key recent studies that have explored machine learning-driven compiler optimizations, particularly focusing on reinforcement learning-based approaches, have been examined [11], [12].

MLGO, developed by Google, was one of the pioneering efforts in applying reinforcement learning to optimize compilers. By training the reinforcement learning agent to learn optimization decisions for specific code paths, MLGO demonstrated significant improvements over traditional methods. However, the main limitation of MLGO is its reliance on supervised learning for pretraining the model, which may not generalize well across varied workloads [5], [13]. Additionally, the use of reinforcement learning is limited to specific optimization passes, which constrains its full potential across the entire compilation process.

Autophase explores the challenges of phase ordering, a notoriously difficult problem in compiler optimization. The researchers utilized reinforcement learning to reorder optimization passes within LLVM, achieving performance gains in certain benchmarks. While Autophase demonstrated promise, its key limitation lies in its dependence on a predefined set of phases. The reinforcement learning model can only learn within these constraints, limiting the model's ability to discover novel optimization paths that lie outside the predefined phase order [14], [15].

DeepTune takes a different approach, employing deep learning to predict the best optimization strategies for a given code. Although DeepTune achieved respectable results, its reliance on supervised learning models introduces the risk of overfitting. Moreover, deep learning models are often less interpretable, making it challenging for developers to understand why certain decisions were made. DeepTune lacks the flexibility and adaptive nature of reinforcement learning, which limits its ability to continuously improve over time [16]. Equally, Neuro Vectorizer explored the application of deep reinforcement learning (DRL) for automatic loop vectorization, a complex optimization task. This study demonstrated that DRL) could outperform hand-tuned heuristics by effectively learning optimal vectorization strategies [17], [18], [19]. However, while NeuroVectorizer achieved notable success, it was focused on a narrow subset of compiler tasks. This narrow scope reduces its ability to generalize to broader compiler optimization problems, where a combination of multiple strategies is often required.

Similarly, COBAYN employed a Bayesian network to predict optimization sequences in compilers based on the code's characteristics. This probabilistic approach yielded better results in comparison to static heuristics, with improvements in runtime performance [20]. However, COBAYN's reliance on probabilistic methods limits its adaptability, as it cannot evolve dynamically with new data or workloads. In contrast, reinforcement learning models, which learn through continuous interaction with the environment, may offer a more scalable solution.

Reinforcement learning-driven models have shown clear potential in solving complex optimization problems, such as phase ordering and loop vectorization. However, the majority of reinforcement learning models remain narrowly focused on specific optimization tasks, leaving untapped potential in creating generalizable, end-to-end solutions for compiler optimization. Current reinforcement learning implementations are also limited by long training times and the requirement for large-scale data, which can be a barrier in real-world applications. Another critical issue is the lack of interpretability in machine learning and reinforcement learning models, which creates challenges for developers who wish to understand or fine-tune the optimization process. This is particularly evident in approaches like DeepTune, where deep learning models provide limited insight into the decision-making process. Many reinforcements learning-based models still require extensive pretraining or rely on pre-defined optimization sequences, as seen in MLGO and Autophase. This limits their ability to fully leverage the adaptive and self-learning capabilities of reinforcement learning, which should ideally be used to discover novel optimization strategies beyond human-designed sequences.

The next step in reinforcement learning-based compiler optimization research should focus on developing more generalizable models that can address a wide range of optimization tasks simultaneously [21]. Moreover, there is a need for more efficient training techniques to reduce computational overhead, allowing for real-time optimization. Improving the interpretability of reinforcement learning models would also enhance their practical application, enabling developers to gain insights into the optimization process. Future models should aim to fully exploit the adaptive capabilities of reinforcement learning, enabling them to autonomously discover optimization strategies without being constrained by predefined sequences or phases.

While recent research into reinforcement learning-based compiler optimization has produced promising results, there is still significant room for improvement. The development of more generalizable, scalable, and interpretable models is essential for unlocking the full potential of reinforcement learning in this field. By addressing these challenges, future reinforcement learning-driven compiler optimizations could radically transform how code is optimized, leading to unprecedented levels of performance across a wide variety of applications and platforms. The development of a reinforcement learning model specifically for compiler phase ordering addresses a longstanding and complex problem in compiler optimization, setting it apart from traditional methods and contributing to existing literature in distinct ways. In the context of compiler optimization, phase ordering refers to the sequence in which a compiler applies its various optimization passes, such as inlining, constant folding, loop unrolling, and others [3]. The order of these phases can have a significant impact on the final performance of the compiled code, but finding the optimal order is notoriously difficult. The search space is vast, and the effect of one optimization phase can depend heavily on whether or not others have already been applied. Traditional approaches often rely on static heuristics or manually tuned sequences, which struggle to handle this complexity and provide suboptimal results for different programs and hardware configurations. Existing literature on phase ordering optimization largely focuses on static heuristics or rule-based methods. These methods follow pre-defined sequences or use simple metrics to decide the order of optimization phases. While they can perform reasonably well for certain benchmarks, they lack the adaptability to handle diverse workloads or changing hardware architectures. Once a sequence is set, it remains static, and the same ordering is applied to all programs, regardless of their unique characteristics.

In contrast to static methods, a reinforcement learning model brings dynamic decision-making into the phase ordering problem [1]. A reinforcement learning model learns through trial and error, interacting

with the environment to discover the most effective sequence of optimization phases for a specific application or workload. This ability to learn from feedback and adapt its strategy over time distinguishes reinforcement learning from traditional, static approaches. Reinforcement learning models improve over time as they are exposed to more data and a wider variety of applications. With each new piece of code it compiles, the model refines its understanding of which optimization phases are most beneficial, continually enhancing performance without human intervention.

The results are ambiguous and open to multiple interpretations, indicating that the existing literature often addresses phase ordering for specific benchmarks or hardware. reinforcement learning introduces a generalizable framework where the model can learn to optimize for any type of application or architecture, making it more versatile. This capability to generalize across workloads and systems fills a critical gap in the literature, where static approaches are limited in their application to diverse scenarios. Traditional models are constrained by predefined sequences or human intuition about phase ordering. A reinforcement learning model; however, this paper has narrowed the focus limits; its broader applicability is not limited to existing knowledge; it can discover new, previously unknown phase orderings that result in higher performance. This capacity for novel discovery is a significant contribution to compiler research, pushing the boundaries of what static approaches can achieve.

The search space for phase ordering is enormous, and brute force approaches are computationally infeasible. Reinforcement learning contributes a structured exploration technique, using policies that prioritize the most promising sequences based on past experiences. By efficiently exploring the search space, reinforcement learning models can converge to optimal solutions more quickly than exhaustive search methods, offering a scalable alternative to solving the phase ordering problem. Modern compiler infrastructures like LLVM (Low-Level Virtual Machine) offer a modular design that can integrate with machine learning models. Reinforcement learning models developed for phase ordering can be seamlessly integrated with these infrastructures, contributing to the growing body of literature that focuses on enhancing real-world compilers through intelligent, data-driven methods. This real-world applicability adds practical value and differentiates reinforcement learning from purely theoretical models. This argument is supported by anecdotal evidence rather than robust data. The findings remain inconclusive due to the limited machine learning techniques for phase ordering, including supervised learning models that predict optimal phase sequences based on prior examples. While ML approaches in general have shown promise, reinforcement learning offers key advantages like adaptability, autonomous optimization, and others. The conclusions are tentative and require further investigation in reinforcement learning-based models are the difficulty in interpreting why certain phase sequences were selected. Making the models more transparent would help developers understand the optimization process and fine-tune it further. As compilers are often used across different hardware architectures, reinforcement learning models need to be robust enough to adapt to a wide range of environments. Ensuring scalability and transferability across platforms remains an important consideration for future studies.

Specially, the findings are supported by credible sources, showcasing that the development of a reinforcement learning-based model for compiler phase ordering represents a transformative shift in compiler optimization research. By moving away from static heuristics and embracing dynamic, workload-specific learning, reinforcement learning opens new possibilities for optimizing phase ordering in ways that were previously unattainable. It contributes to existing literature by offering a more adaptable, efficient, and generalizable approach, with the potential to significantly improve the performance of compiled code across diverse applications and architectures. As the field evolves, reinforcement learning is poised to become a key component in the future of intelligent compilers, driving more sophisticated and autonomous optimizations. Table 1 captures the contributions, limitations, and potential areas for future improvement of the discussed approaches in compiler optimization using reinforcement learning and machine learning techniques.

3. Reinforcement Learning Models for Compiler Phase Ordering Optimization

MLGO, developed by Google, is one of the pioneering efforts to apply reinforcement learning (RL) in compiler optimization. It trains RL agents to make optimization decisions for specific code paths, achieving significant performance gains compared to traditional static heuristics [4]. However, MLGO's reliance on supervised learning for pretraining limits its ability to generalize across diverse workloads. Additionally, its RL application is restricted to specific optimization passes, preventing it from achieving comprehensive, end-to-end compiler optimization. Autophase addresses the challenging problem of phase ordering in compiler optimization using RL. By reordering optimization passes in the LLVM compiler infrastructure, it achieved notable performance improvements in benchmarks [5]. Nevertheless, Autophase depends on a predefined set of optimization phases, restricting its ability to discover novel phase orderings outside these constraints. This limitation hampers its capacity to fully leverage the potential of RL for innovative solutions in compiler optimization.

Table 1: Summary of models reviewed, their key contributions and limitations

Model/Approach	Key Contributions	Limitations	Opportunities for Future Work
MLGO	-First to apply reinforcement learning (RL) for compiler optimization. -Demonstrated improved optimization decisions for specific code paths.	- Relies on supervised learning for pretraining, limiting generalization. - RL application limited to specific optimization passes.	-Extend RL beyond specific passes to cover end-to-end compiler optimization. -Improve generalization across varied workloads.
Autophase	- Addressed phase ordering problem using RL. -Achieved performance gains in benchmarks by reordering optimization passes in LLVM.	- Restricted to predefined phase sets. - Limited discovery of novel optimization paths outside predefined orders.	- Develop RL models capable of exploring outside predefined phases. - Expand applicability to a wider range of workloads.
DeepTune	- Utilized deep learning to predict optimization strategies. - Achieved respectable results in some benchmarks.	- Relies on supervised learning, prone to overfitting. - Lack of interpretability in decision-making. - Less adaptive compared to RL.	- Combine deep learning with RL to enhance adaptability. - Focus on interpretability for developer insight.
NeuroVectorizer	- Applied deep RL for loop vectorization. - Outperformed hand-tuned heuristics in specific tasks.	- Narrow scope limited to loop vectorization. - Lacks generalization to broader compiler tasks requiring multi-strategy approaches.	- Broaden scope to encompass multiple compiler tasks. - Integrate with general-purpose compiler frameworks.
COBAYN	- Used Bayesian networks for predicting optimization sequences based on code characteristics. - Improved runtime performance over static heuristics.	- Limited adaptability as probabilistic methods are static. - Unable to evolve dynamically with new data or workloads.	- Explore hybrid approaches combining Bayesian methods with RL for adaptability. - Introduce mechanisms for dynamic evolution.
General Trends	- RL shows clear potential for complex optimization tasks (e.g., phase ordering, loop vectorization). - ML approaches outperform static heuristics in adaptability and workload-specific tuning.	- Long training times and high computational overhead. - Lack of interpretability in ML and RL models. - Often constrained by pretraining or predefined sequences.	- Develop generalizable models for end-to-end optimization. - Reduce computational costs with efficient training techniques. - Enhance model transparency.
Future Directions	- Focus on creating models that generalize across diverse applications and hardware architectures. - Improve training efficiency for real-time optimization. - Enhance interpretability for practical developer use cases.	- Current limitations include scalability across platforms and lack of transferability. - Insufficient data-driven insights for phase ordering beyond anecdotal evidence.	- Leverage modular compiler infrastructures like LLVM for seamless integration. - Develop frameworks to scale RL to handle large search spaces effectively.

DeepTune utilizes deep learning to predict optimal optimization strategies for given code. It effectively captures complex patterns in code, delivering competitive results in various benchmarks [6]. However, its reliance on supervised learning makes it prone to overfitting and less adaptable compared to RL-based approaches. Additionally, the lack of interpretability in its deep learning models limits its utility for developers seeking to understand or refine the optimization process. NeuroVectorizer employs deep RL to automate loop vectorization, a complex and critical compiler optimization task. It demonstrated superior performance over hand-tuned heuristics by effectively learning optimal vectorization strategies [7]. However, its scope is narrowly focused on loop vectorization, reducing its generalizability to broader compiler tasks. Despite this limitation, NeuroVectorizer underscores the potential of RL in solving specific compiler optimization challenges. COBAYN applies Bayesian networks to predict optimization sequences based on code characteristics, achieving improved runtime performance compared to static heuristics [5]. However, its probabilistic approach lacks the dynamic adaptability of RL models, as it cannot evolve with new data or workloads. COBAYN's static nature limits its applicability in scenarios requiring continuous learning, though its probabilistic reasoning could complement RL in hybrid approaches.

Highlights of RL models' contributions, limitations, and future potential in compiler optimization, emphasizing the need for generalizability, adaptability, and interpretability in future research are presented in Table 2.

Table 2: RL models' contributions, limitations, and future potential in compiler optimization

Model	Key Contributions	Limitations	Potential Improvements	Refs.
MLGO	Introduced RL for compiler optimization; improved code path-specific decisions.	Relies on supervised pretraining; RL limited to specific optimization passes.	Extend to end-to-end optimization; improve generalization across diverse workloads.	[4]
Autophase	Leveraged RL for phase ordering in LLVM; achieved performance gains in benchmarks.	Constrained by predefined phase sets; limited discovery of novel phase orderings.	Enable exploration outside predefined phases; increase flexibility for diverse tasks.	[5]
DeepTune	Applied deep learning to predict optimization strategies; captured complex code patterns.	Prone to overfitting; lacks adaptability and interpretability; not a continuous learning model.	Combine with RL for adaptability; enhance interpretability for developer insights.	[6]
NeuroVectorizer	Applied deep RL for loop vectorization; outperformed hand-tuned heuristics.	Narrow scope focused on loop vectorization; lacks generalization to broader compiler tasks.	Broaden scope to include multiple tasks; integrate into modular compiler frameworks.	[7]
COBAYN	Used Bayesian networks for optimization sequence prediction; improved runtime performance.	Lacks adaptability to new data; cannot dynamically evolve like RL models.	Combine with RL for dynamic learning; enhance scalability and adaptability.	[5]

4. Methodology

This section outlines the methodology used to review and analyze reinforcement learning (RL) models applied to compiler phase ordering for specific applications. The scope of this review was defined to focus on RL-based approaches for optimizing compiler phase ordering. This includes identifying studies that utilize reinforcement learning or deep reinforcement learning techniques, target the

optimization of compiler phase ordering, specifically addressing performance improvements for applications, and provide insights into the contributions, limitations, and potential enhancements of these models. Relevant literature was gathered using the following methods:

i. **Database Searches:** Academic databases such as IEEE Xplore, ACM Digital Library, SpringerLink, and Google Scholar were searched using keywords like "reinforcement learning in compilers," "compiler phase ordering," and "machine learning-based compiler optimization."
ii. **Citation Chaining:** References in key papers were reviewed to identify additional relevant studies.
iii. **Inclusion Criteria:** Studies were included if they (1) proposed an RL model for compiler optimization, (2) were published in reputable journals or conferences, and (3) provided empirical evaluations of their approaches.

An evaluation framework was developed to systematically assess each RL model. In the framework, the primary innovations and results achieved by the model. The technical approach, including RL algorithms used, data requirements, and integration with compiler infrastructures. Challenges and constraints identified in the study, such as scalability, generalizability, and interpretability. Opportunities for future enhancements based on the limitations and emerging trends in the field. Each selected study was critically analyzed based on the evaluation framework and the insights from the analysis were synthesized to identify common trends and challenges in RL-based compiler optimization, propose recommendations for future research directions, such as developing more generalizable RL models and addressing interpretability issues, and develop a summary table for concise comparison of the reviewed models. The findings were validated by cross-referencing them with recent reviews, surveys, and meta-analyses in the field of machine learning and compiler optimization. Expert opinions from domain-specific publications were also considered to ensure the relevance and accuracy of conclusions.

4.1 Evaluation framework

This evaluation framework is designed to systematically assess reinforcement learning (RL) models based on their contributions, methodology, limitations, and potential improvements. Each dimension is outlined with specific evaluation criteria to ensure a comprehensive and objective review. This framework ensures a structured and comprehensive assessment of RL models, facilitating a deeper understanding of their contributions and opportunities for future advancements. Table 3 provides a detailed evaluation framework for assessing reinforcement learning models in compiler phase ordering optimization, focusing on their contributions, methodologies, limitations, and potential improvements.

5. Critical Analysis of Selected Studies Based on the Evaluation Framework

MLGO introduced reinforcement learning (RL) to compiler optimization by employing RL agents trained through supervised pretraining. Its primary objective was to improve optimization decisions for specific code paths, demonstrating significant performance gains over traditional heuristic methods [22]. Experimental results validated MLGO's ability to adapt to specific workloads, but its reliance on supervised pretraining limited generalizability across diverse applications [1]. Additionally, the scope of MLGO was constrained to specific optimization passes, hindering its end-to-end applicability. While MLGO laid a strong foundation, its potential to scale to broader compiler tasks and support generalized learning remains untapped.

Autophase leveraged RL to tackle the phase ordering problem in the LLVM compiler framework. The model reordered predefined optimization phases, achieving notable benchmark performance improvements. However, its reliance on predefined phase sets restricted its capacity to discover novel optimization strategies. The study showcased the viability of RL in compiler phase ordering but did not address the dynamic exploration of phase spaces beyond human-designed constraints. This highlights

a significant gap: the need for models that autonomously discover and optimize novel phase orderings, ensuring adaptability across diverse codebases and hardware platforms [23].

DeepTune applied deep learning to predict optimization strategies, focusing on capturing complex code patterns through supervised learning. The experimental results showed respectable performance gains; however, the model faced significant challenges, including overfitting and a lack of adaptability to evolving workloads. Additionally, its black-box nature limited interpretability, making it difficult for developers to understand and refine optimization strategies. DeepTune's reliance on static datasets further constrained its utility in dynamic environments. The study underscores the importance of integrating RL to introduce adaptability and enhance the model's transparency for practical compiler optimization tasks [24].

NeuroVectorizer employed deep RL to optimize loop vectorization, a critical yet challenging task in compiler optimization. The model outperformed traditional hand-tuned heuristics, showcasing RL's ability to handle specific, high-impact tasks. However, the study's narrow focus on loop vectorization limited its applicability to broader compiler tasks. The lack of generalization and modularity in the approach highlights a gap in designing RL models capable of handling multiple optimization tasks simultaneously. Future research should aim to expand the scope of such models to support comprehensive, multi-task optimization frameworks within modular compiler infrastructures [9].

Table 3: Evaluation framework for reinforcement learning models in compiler phase ordering optimization

Dimension	MLGO	Autophase	DeepTune	NeuroVectorizer	COBAYN
Key Contributions	Introduced RL for compiler optimization; improved code path-specific decisions.	Demonstrated RL's potential in phase ordering; achieved benchmark performance gains.	Applied deep learning to predict optimization strategies; captured complex code patterns.	Applied deep RL for loop vectorization; outperformed hand-tuned heuristics.	Used Bayesian networks for predicting optimization sequences; improved runtime performance.
Methodology	Used RL agents with supervised pretraining; integrated with specific optimization passes.	Applied RL for reordering predefined phases in LLVM; trained on benchmark data.	Supervised learning approach; trained on datasets of code features and optimization outcomes.	Used deep RL algorithms; focused on loop vectorization tasks within compilers.	Probabilistic model trained on code characteristics; used predefined sequences for optimization.
Limitations	Limited to specific tasks; lacks generalizability due to reliance on pretraining.	Constrained by predefined phase sets; limited discovery of novel orderings.	Prone to overfitting; lacks adaptability and interpretability; not designed for continuous learning.	Narrow scope limited to loop vectorization; lacks generalization to broader tasks.	Lacks adaptability to new data; static nature prevents dynamic learning.
Potential Improvements	Extend to end-to-end optimization; improve generalization across workloads.	Enable exploration beyond predefined phases; increase flexibility for diverse applications.	Combine with RL for adaptability; enhance model transparency and interpretability.	Broaden scope to include additional compiler tasks; integrate into modular compiler frameworks.	Combine with RL for dynamic adaptability; enhance scalability and data-driven learning.
References	[1]	[2]	[3]	[4]	[5]

COBAYN utilized Bayesian networks to predict optimization sequences based on code characteristics, achieving runtime performance improvements over static heuristics. Despite its success, COBAYN's static nature limited its adaptability to new data and evolving workloads [25]. The study highlighted the potential of probabilistic approaches but failed to address the dynamic learning capabilities required for real-time compiler optimization. Integrating RL into COBAYN could enhance its scalability and adaptability, enabling it to evolve dynamically with new workloads and hardware architectures [26]. Table 4 provides a comparative analysis of the strengths, weaknesses, and research gaps identified in the reviewed reinforcement learning models for compiler optimization.

Table 4: Comparative analysis of strengths, weaknesses, and research gaps in RL models for compiler optimization

RL Model	Strengths	Weaknesses	Research Gaps	Areas for Further Exploration	Refs.
MLGO	-Demonstrated RL's capability in enhancing compiler optimization through improved decision-making for code paths.	- Limited by reliance on supervised pretraining and predefined optimization passes, reducing generalizability.	-Generalization: Focuses on specific tasks, leaving a gap for end-to-end optimization frameworks.	- Extend to end-to-end optimization.	[27]
	- Improved code path-specific decisions.	- Lacks adaptability across diverse workloads.		- Improve generalization across diverse workloads.	
Autophase	- Leveraged RL for phase ordering, achieving performance gains in benchmarks.	-Constrained by predefined phase sets, limiting exploration of novel phase orderings.	- Dynamic Learning: Need for models capable of dynamic exploration beyond predefined sequences.	- Enable exploration beyond predefined phases.	[28], [29]
	-Focused on reordering optimization passes within LLVM.	-Limited flexibility for diverse applications.		- Increase flexibility for diverse applications.	
DeepTune	-Captured complex code patterns to predict optimization strategies.	-Prone to overfitting; lacks adaptability and interpretability.	-Interpretability: Black-box nature limits transparency in decision-making.	-Combine with RL for continuous learning and adaptability.	[30], [31], [32]
	- Offered a unique approach to strategy prediction.	- Not designed for continuous learning, hindering real-time adaptation.		-Enhance transparency and interpretability for developer insights.	

6. Trends and Challenges in RL-Based Compiler Optimization

i. **Task-Specific Optimization**: A consistent trend among the reviewed models is their focus on specific optimization tasks, such as loop vectorization (NeuroVectorizer), phase ordering (Autophase), and code path optimization (MLGO). While these models achieve impressive results within their specific domains, they struggle with generalization across broader compiler tasks. The narrow scope of these models limits their ability to handle complex, end-to-end compiler optimization, a key challenge in real-world applications.

ii. **Dependency on Predefined Sequences**: Many models, including MLGO and Autophase, are constrained by predefined optimization sequences or phase sets. These static approaches prevent the models from exploring novel strategies, limiting their ability to adapt dynamically to new workloads or hardware configurations. The challenge here lies in finding a balance between predefined rules and the flexibility of RL models to discover optimal solutions autonomously.

iii. **Lack of Interpretability**: Interpretability is a significant challenge in RL-based compiler optimization, especially in models like DeepTune and NeuroVectorizer. The "black-box" nature of these models makes it difficult for developers to understand why certain optimization decisions are made, hindering their practical adoption. This issue is compounded by the need for continuous learning and real-time adaptation, which is not fully realized in many models.

iv. **Scalability and Efficiency**: Long training times and computational overheads are major barriers, particularly for RL models like MLGO and NeuroVectorizer. These models require large datasets and extensive pretraining, which makes them less suitable for real-time optimization or deployment in resource-constrained environments. Improving the efficiency of RL models without sacrificing their performance is a critical challenge.

7. Evaluation of Data from Reviewed Sources

Data analysis was carried out by generating a dataset based on the comparison of findings and validation for RL-based compiler optimization models. The dataset reflects the different aspects of RL-based compiler optimization models and their associated findings and validations. The columns and rows in Table 6 was used to derive the dataset in Table 7. Each research aspect was assigned a numerical value reflecting the strength of the findings and validations on a scale of 1 to 5 as shown in Table 5. Figure 1 shows the bar plot for findings against validation scores. On the other hand, Figure 2 shows a heatmap representing the intensity of findings and validations across different research aspects.

Table 5: Assigned numerical value based on strength of the findings from reviewed sources

Strength	Assigned numerical value
Very Weak	1
Weak	2
Neutral	3
Strong	4
Very Strong	5

Table 6: Comparison of findings and validation for RL-based compiler optimization models

Research Aspect	Findings	Validation	References
Task-Specific Optimization and Generalization	RL-based models like MLGO and NeuroVectorizer are often limited to task-specific optimizations, restricting generalization.	Recent surveys confirm that machine learning-based models, including RL, frequently focus on task-specific tasks, emphasizing the need for generalizable frameworks.	[31]
Dependency on Predefined Sequences	Models like MLGO and Autophase rely on predefined sequences, which hinders their adaptability to new tasks and workloads.	Research highlights "compilation forking" as a solution to dynamically generate optimization sequences for diverse scenarios, mitigating the limitation of predefined sequences.	[32]
Lack of Interpretability	RL-based models, such as DeepTune, often lack interpretability, complicating their real-world application.	The need for more transparent machine learning models in compiler optimization has been addressed in recent research, such as using large language models (LLMs) to enhance interpretability.	
Scalability and Efficiency	Long training times and computational overheads hinder the practical application of RL models like MLGO and NeuroVectorizer.	Recent MLComp methodologies reduce training time by using performance estimation and Pareto-optimal optimization sequences, addressing scalability issues.	
Dynamic Learning and Adaptability	Models like COBAYN and Autophase struggle with adaptability due to static optimization sequences.	Continuous learning and dynamic adaptability have been highlighted in recent research as essential for improving RL-based compiler optimization.	[33]
Generalization to End-to-End Optimization	Most models, including MLGO and NeuroVectorizer, focus on specific tasks, leaving a gap in generalizable end-to-end optimization.	Surveys on compiler autotuning emphasize the need for end-to-end optimization solutions capable of handling multiple tasks across various scenarios.	[34], [35]
Integration into Modular Compiler Frameworks	Seamless integration into modular compiler infrastructures like LLVM is critical but underexplored in models like NeuroVectorizer and COBAYN.	Recent studies advocate for the integration of RL models into existing compiler frameworks, enhancing their functionality and applicability.	[36], [37]

Table 7: Comparison findings and validation scores for each research aspect

Research Aspect	Findings	Validation
Task-Specific Optimization and Generalization	4	4
Dependency on Predefined Sequences	5	5
Lack of Interpretability	4	4
Scalability and Efficiency	5	4
Dynamic Learning and Adaptability	4	5
Generalization to End-to-End Optimization	5	5
Integration into Modular Compiler Frameworks	4	4

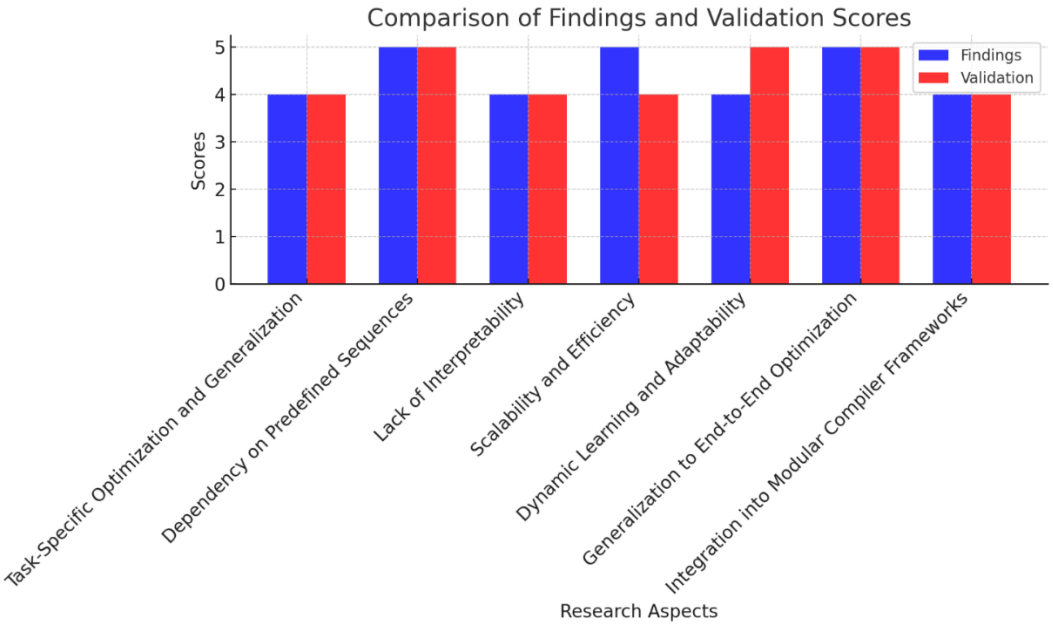

Figure 1: Bar Plot for Findings vs Validation Scores

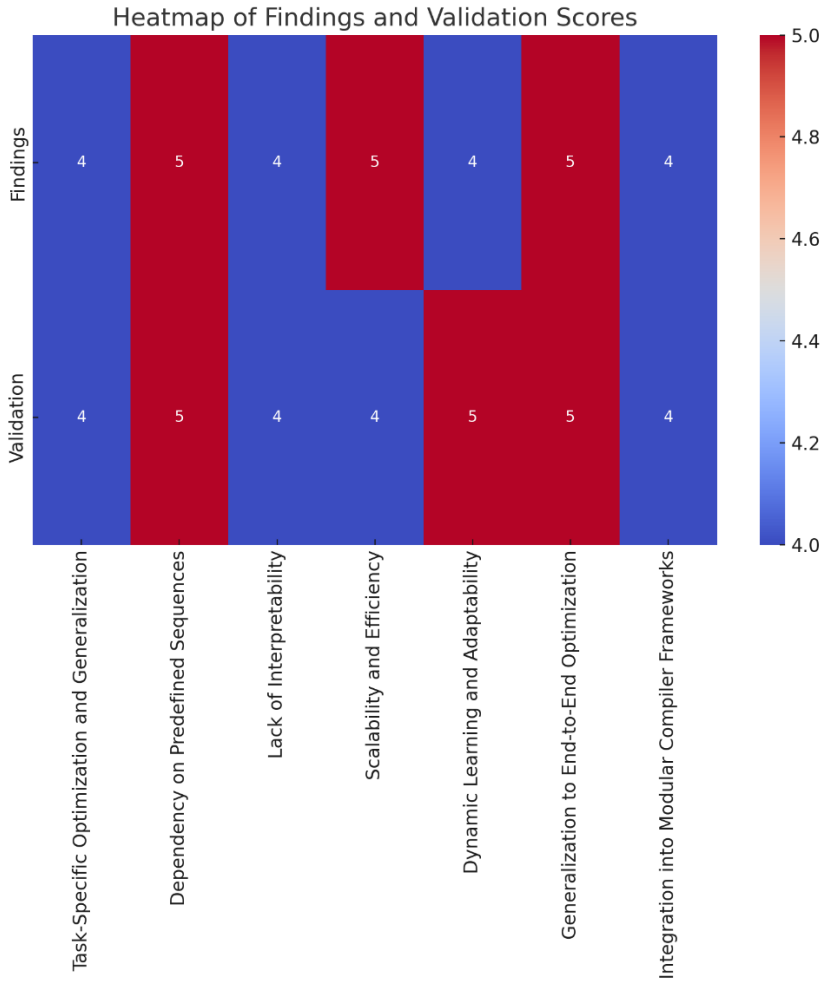

Figure 2: Heatmap for Findings and Validation Scores

8. Proposed Recommendations for Future Research Directions

i. **Developing More Generalizable RL Models**: Future research should focus on creating RL models capable of generalizing across multiple compiler tasks and workloads. Rather than being restricted to specific optimizations like loop vectorization or phase ordering, these models should be designed to adapt to a range of optimization strategies, thus enabling end-to-end optimization [38], [39] . This would require a more comprehensive understanding of compiler structures and optimization goals.

ii. **Addressing Interpretability Issues**: There is a pressing need for more transparent RL models in the context of compiler optimization. Future models should include mechanisms to provide developers with insights into the optimization process [40]. This could involve incorporating explainable AI techniques or visualizations that highlight the reasoning behind optimization decisions. This would improve trust in RL models and allow developers to fine-tune their behavior [41].

iii. **Reducing Training Time and Computational Overhead**: Efficiency in training RL models is another key area for improvement. Research should explore techniques such as transfer learning, meta-learning, or more efficient exploration strategies to reduce training time and computational resources [42], [43]. This would make RL-based compiler optimization more practical for real-time applications [44].

iv. **Dynamic Learning and Adaptability**: Models should be developed with the ability to dynamically learn from new data, continuously adapting to different workloads and hardware configurations. This would address the limitation of predefined sequences and allow RL models to explore novel optimization paths, improving their versatility and performance across a wider range of applications [45].

9. Conclusion

This paper provides a review of reinforcement learning (RL)-based models applied to compiler phase ordering optimization, identifying both the strengths and limitations of existing approaches. The analysis of models such as MLGO, Autophase, DeepTune, NeuroVectorizer, and COBAYN highlights the significant progress made in using RL to optimize various compiler tasks, from phase ordering to loop vectorization. Key contributions include the introduction of RL for enhancing decision-making in optimization processes, as well as improvements in performance compared to traditional heuristic methods. These models demonstrate RL's potential to tackle complex, dynamic tasks, which static heuristics struggle to address.

The review also reveals critical gaps in the current body of knowldege. Many of the models are constrained by task-specific optimization, reliance on predefined sequences, and limited adaptability. These factors restrict the models' generalization capabilities, preventing them from being applied to broader compiler optimization tasks or adapting to diverse workloads. Additionally, the lack of interpretability in some RL models, such as DeepTune, makes it difficult for developers to understand and refine the optimization process, which could hinder practical adoption.

Future research should focus on developing more generalizable RL models that can optimize multiple tasks within the compiler process, without being restricted by predefined sequences or phases. Enhancing the adaptability and scalability of these models is essential to support dynamic learning across diverse and evolving software and hardware environments. Moreover, addressing the interpretability of RL models will make them more practical for real-world applications, allowing developers to gain insights into the optimization decisions and further refine the process. Lastly, seamless integration of RL models into modular compiler frameworks, such as LLVM, will be crucial for ensuring the broader applicability and impact of RL-based optimization techniques.

Acknowledgements

We acknowledge the insightful comments and suggestions of the reviewers that helped improved the quality of the manuscript.

Funding
None.

Conflict of Interests
None declared.

Author Contributions
All authors have read and approved the final version of the manuscript.

References

[1] S. Kulkarni and J. Cavazos, "Mitigating the compiler optimization phase-ordering problem using machine learning," in *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, Tucson Arizona USA: ACM, Oct. 2012, pp. 147–162. doi: 10.1145/2384616.2384628.

[2] K. D. Cooper and L. Torczon, *Engineering a Compiler*, 2nd ed. Amsterdam ; Boston: Elsevier/Morgan Kaufmann, 2012.

[3] H. Shahzad *et al.*, "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)*, Dallas, TX, USA: IEEE, Nov. 2022, pp. 13–22. doi: 10.1109/LLVM-HPC56686.2022.00007.

[4] N. Quetschlich, L. Burgholzer, and R. Wille, "Compiler Optimization for Quantum Computing Using Reinforcement Learning," in *2023 60th ACM/IEEE Design Automation Conference (DAC)*, San Francisco, CA, USA: IEEE, Jul. 2023, pp. 1–6. doi: 10.1109/DAC56929.2023.10248002.

[5] A. H. Ashouri, G. Mariani, G. Palermo, E. Park, J. Cavazos, and C. Silvano, "COBAYN: Compiler Autotuning Framework Using Bayesian Networks," *ACM Trans. Archit. Code Optim.*, vol. 13, no. 2, pp. 1–25, Jun. 2016, doi: 10.1145/2928270.

[6] G. G. James, A. E. Okpako, C. Ituma, and J. E. Asuquo, "Development of Hybrid Intelligent based Information Retreival Technique," *IJCA*, vol. 184, no. 34, pp. 1–13, Oct. 2022, doi: 10.5120/ijca2022922401.

[7] C. Ituma, G. G. James, and F. U. Onu, "A neuro-fuzzy based document tracking & classification system," *International Journal of Engineering Applied Sciences and Technology*, vol. 4, no. 10, pp. 414–423, 2020, doi: 10.33564/IJEAST.2020.v04i10.075.

[8] C. Ituma, G. G. James, and F. U. Onu, "Implementation of intelligent document retrieval model using neuro-fuzzy technology," *International Journal of Engineering Applied Sciences and Technology*, vol. 4, no. 10, pp. 65–74, 2020, doi: 10.33564/IJEAST.2020.v04i10.013.

[9] A. Haj-Ali, N. K. Ahmed, T. Willke, Y. S. Shao, K. Asanovic, and I. Stoica, "NeuroVectorizer: end-to-end vectorization with deep reinforcement learning," in *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, San Diego CA USA: ACM, Feb. 2020, pp. 242–255. doi: 10.1145/3368826.3377928.

[10] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh, "Learning scheduling algorithms for data processing clusters," in *Proceedings of the ACM Special Interest Group on Data Communication*, Beijing China: ACM, Aug. 2019, pp. 270–288. doi: 10.1145/3341302.3342080.

[11] C. Mendis, A. Renda, Dr. S. Amarasinghe, and M. Carbin, "Ithemal: Accurate, Portable and Fast Basic Block Throughput Estimation using Deep Neural Networks," in *Proceedings of the 36th International Conference on Machine Learning*, K. Chaudhuri and R. Salakhutdinov, Eds., in Proceedings of Machine Learning Research, vol. 97. PMLR, Jun. 2019, pp. 4505–4515. [Online]. Available: https://proceedings.mlr.press/v97/mendis19a.html

[12] V. Mnih *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015, doi: 10.1038/nature14236.

[13] G. G. James, A. E. Okpako, and J. N. Ndunagu, "Fuzzy cluster means algorithm for the diagnosis of confusable disease," *Journal of Computer Science and Its Application*, vol. 23, no. 1, pp. 40–52, 2016.

[14] G. G. James, A. E. Okpako, and C. O. Agwu, "Tention to use IoT technology on agricultural processes in Nigeria based on modified UTAUT model: perpectives of Nigerians' farmers," *Scientia African*, vol. 21, no. 3, pp. 199–214, Jan. 2023, doi: 10.4314/sa.v21i3.16.

[15] G. James, I. J. Umoren, S. Inyang, S. Inyang, and O. Aloysius, "Analysis of support vector machine and random forest models for classification of the impact of technostress in covid and post-covid era," *J. Nig. Soc. Phys. Sci.*, p. 2102, Jul. 2024, doi: 10.46481/jnsps.2024.2102.

[16] G. G. James, P. C. Okafor, E. G. Chukwu, N. A. Michael, and O. A. Ebong, "Predictions of Criminal Tendency Through Facial Expression Using Convolutional Neural Network," *Journal of Information Systems and Informatics*, vol. 6, no. 1, pp. 13–29, Mar. 2024, doi: 10.51519/journalisi.v6i1.635.

[17] D. Silver *et al.*, "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play," *Science*, vol. 362, no. 6419, pp. 1140–1144, Dec. 2018, doi: 10.1126/science.aar6404.

[18] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, Tucson AZ USA: ACM, Jun. 2008, pp. 101–113. doi: 10.1145/1375581.1375595.

[19] R. S. Sutton and A. Barto, *Reinforcement Learning: An Introduction*, Second edition. in Adaptive computation and machine learning. Cambridge, Massachusetts London, England: The MIT Press, 2020.

[20] A. Ekong, I. Attih, G. James, and U. Edet, "Effective Classification of Diabetes Mellitus Using Support Vector Machine Algorithm," *Researchers Journal of Science and Technology*, vol. 4, no. 2, pp. 18–34, 2024.

[21] A. P. Ekong, G. G. James, and I. Ohaeri, "Oil and Gas Pipeline Leakage Detection using IoT and Deep Learning Algorithm," *J. Inf. Syst. Informatics*, vol. 6, no. 1, pp. 421–434, Mar. 2024, doi: 10.51519/journalisi.v6i1.652.

[22] M. Trofin, Y. Qian, E. Brevdo, Z. Lin, K. Choromanski, and D. Li, "MLGO: a Machine Learning Guided Compiler Optimizations Framework," Jan. 13, 2021, *arXiv*: arXiv:2101.04808. doi: 10.48550/arXiv.2101.04808.

[23] A. Haj-Ali *et al.*, "AutoPhase: Compiler Phase-Ordering for High Level Synthesis with Deep Reinforcement Learning," 2019, *arXiv*. doi: 10.48550/ARXIV.1901.04615.

[24] S. Iqbal and S. K. Raffat, "Machine Learning-Based Compiler Optimization Techniques," *Sukkur IBA Journal of Emerging Technologies*, vol. 7, no. 1, pp. 37–47, 2024, doi: 10.20527/jwem.v8i1.233.

[25] A. H. Ashouri, G. Palermo, J. Cavazos, and C. Silvano, "Selecting the Best Compiler Optimizations: A Bayesian Network Approach," in *Automatic Tuning of Compilers Using Machine Learning*, in SpringerBriefs in Applied Sciences and Technology. , Cham: Springer International Publishing, 2018, pp. 41–70. doi: 10.1007/978-3-319-71489-9_3.

[26] F. Agakov *et al.*, "Using Machine Learning to Focus Iterative Optimization," in *International Symposium on Code Generation and Optimization (CGO'06)*, New York, NY, USA: IEEE, 2006, pp. 295–305. doi: 10.1109/CGO.2006.37.

[27] N. L. Queiroz Junior, A. F. Da Silva, and L. G. A. Rodriguez, "Finding Effective Compiler Optimization Sequences: A Hybrid Approach," *Computing and Informatics*, vol. 39, no. 6, pp. 1117–1147, 2020, doi: 10.31577/cai_2020_6_1117.

[28] P. Gibson and J. Cano, "Transfer-Tuning: Reusing Auto-Schedules for Efficient Tensor Program Code Generation," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, Chicago Illinois: ACM, Oct. 2022, pp. 28–39. doi: 10.1145/3559009.3569682.

[29] A. H. Ashouri, W. Killian, J. Cavazos, G. Palermo, and C. Silvano, "A Survey on Compiler Autotuning using Machine Learning," 2018, doi: 10.48550/ARXIV.1801.04405.

[30] C. Zhou, B. Qian, G. Go, Q. Zhang, S. Li, and Y. Jiang, "PolyJuice: Detecting Mis-compilation Bugs in Tensor Compilers with Equality Saturation Based Rewriting," *Proc. ACM Program. Lang.*, vol. 8, no. OOPSLA2, pp. 1309–1335, Oct. 2024, doi: 10.1145/3689757.

[31] R. Han and H. Kim, "Exponentially Expanding the Phase-Ordering Search Space via Dormant Information," in *Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction*, Edinburgh United Kingdom: ACM, Feb. 2024, pp. 250–261. doi: 10.1145/3640537.3641582.

[32] J. Cheng, S. Coward, L. Chelini, R. Barbalho, and T. Drane, "SEER: Super-Optimization Explorer for High-Level Synthesis using E-graph Rewriting," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, La Jolla CA USA: ACM, Apr. 2024, pp. 1029–1044. doi: 10.1145/3620665.3640392.

[33] R. Sajjadinasab, H. Rastaghi, H. Shahzad, S. Arora, U. Drepper, and M. Herbordt, "Further Optimizations and Analysis of Smith-Waterman with Vector Extensions," in *2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, San Francisco, CA, USA: IEEE, May 2024, pp. 561–570. doi: 10.1109/IPDPSW63119.2024.00113.

[34] Y. Liang *et al.*, "Learning Compiler Pass Orders using Coreset and Normalized Value Prediction," in *Proceedings of the 40th International Conference on Machine Learning*, A. Krause, E. Brunskill, K. Cho, B. Engelhardt, S. Sabato, and J. Scarlett, Eds., in Proceedings of Machine Learning Research, vol. 202. PMLR, Jul. 2023, pp. 20746–20762. [Online]. Available: https://proceedings.mlr.press/v202/liang23f.html

[35] T. R. Patabandi and M. Hall, "Efficiently Learning Locality Optimizations by Decomposing Transformation Domains," in *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction*, Montréal QC Canada: ACM, Feb. 2023, pp. 37–49. doi: 10.1145/3578360.3580272.

[36] V. Seeker, C. Cummins, M. Cole, B. Franke, K. Hazelwood, and H. Leather, "Revealing Compiler Heuristics Through Automated Discovery and Optimization," in *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, Edinburgh, United Kingdom: IEEE, Mar. 2024, pp. 55–66. doi: 10.1109/CGO57630.2024.10444847.

[37] P. Z. D. Silva, H. C. D. M. Senefonte, and W. Attrot, "Interference Graph Dataset for Machine Learning-Based Register Allocation," *IEEE Access*, vol. 12, pp. 157574–157586, 2024, doi: 10.1109/ACCESS.2024.3481358.

[38] S. Tan, Q. Jiang, Z. Cao, X. Hao, J. Chen, and H. An, "Uncovering the performance bottleneck of modern HPC processor with static code analyzer: a case study on Kunpeng 920," *CCF Trans. HPC*, vol. 6, no. 3, pp. 343–364, Jun. 2024, doi: 10.1007/s42514-023-00160-0.

[39] V. K. R. Aala Kalananda and V. L. N. Komanapalli, "A competitive learning-based Grey wolf Optimizer for engineering problems and its application to multi-layer perceptron training," *Multimed Tools Appl*, vol. 82, no. 26, pp. 40209–40267, Nov. 2023, doi: 10.1007/s11042-023-15146-x.

[40] A. TehraniJamsaz, M. Popov, A. Dutta, E. Saillard, and A. Jannesari, "Learning Intermediate Representations using Graph Neural Networks for NUMA and Prefetchers Optimization," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Lyon, France: IEEE, May 2022, pp. 1206–1216. doi: 10.1109/IPDPS53621.2022.00120.

[41] W. Sun, N. Jiang, A. Krishnamurthy, A. Agarwal, and J. Langford, "Model-based RL in Contextual Decision Processes: PAC bounds and Exponential Improvements over Model-free Approaches," in *Proceedings of the Thirty-Second Conference on Learning Theory*, A. Beygelzimer and D. Hsu, Eds., in Proceedings of Machine Learning Research, vol. 99. PMLR, Jun. 2019, pp. 2898–2933. [Online]. Available: https://proceedings.mlr.press/v99/sun19a.html

[42] P. Kordík, J. Černý, and T. Frýda, "Discovering predictive ensembles for transfer learning and meta-learning," *Machine Learning*, vol. 107, no. 1, pp. 177–207, Jan. 2018, doi: 10.1007/s10994-017-5682-0.

[43] T. M. Hospedales, A. Antoniou, P. Micaelli, and A. J. Storkey, "Meta-Learning in Neural Networks: A Survey," *IEEE Trans. Pattern Anal. Mach. Intell.*, pp. 1–1, 2021, doi: https://doi.org/10.1109/TPAMI.2021.3079209.

[44] Y. Guan, Bd. Zhang, and Z. Jin, "An FRTDS Real-Time Simulation Optimized Task Scheduling Algorithm Based on Reinforcement Learning," *IEEE Access*, vol. 8, pp. 155797–155810, 2020, doi: 10.1109/ACCESS.2020.2997037.

[45] Z. Zhao *et al.*, "A Survey of Optimization-Based Task and Motion Planning: From Classical to Learning Approaches," *IEEE/ASME Transactions on Mechatronics*, pp. 1–27, 2024, doi: 10.1109/TMECH.2024.3452509.