The Theoretical Application of Computational Complexity Theory in Compiler Optimization

Wuwei Shao*

Zhejiang Gongshang University, Hangzhou, 310018, China
**Corresponding Author: ShaoWuwei331@outlook.com*

Abstract: *Computational complexity theory, as an important branch of theoretical computer science, systematically characterizes the consumption of time and space resources in computational tasks, providing a solid foundation for understanding the inherent difficulty of various computational problems. Compiler optimization, as a key aspect of improving program design and execution efficiency, involves numerous complex algorithm designs and resource scheduling issues. The computational complexity characteristics of these tasks directly constrain the feasibility and performance of optimization strategies. Based on the classification system of complexity theory, reduction techniques, and complexity boundary analysis, this paper explores in-depth the complexity modeling and theoretical characterization of key tasks in compiler optimization, such as register allocation and instruction scheduling. It discusses scheduling methods for polynomial-time solvable problems, the complexity control mechanisms of approximation algorithms and heuristic strategies, as well as the structural insights that complexity theory provides for the optimization of compiler backends. The research shows that complexity theory not only offers theoretical guidance for the design of compiler optimization algorithms but also promotes the deep integration of theory and engineering practice, driving theoretical innovation and application expansion for efficient compilation technologies.*

Keywords: *Computational Complexity Theory; Compiler Optimization; Complexity Classification; Register Allocation; Instruction Scheduling; Approximation Algorithms; Complexity Boundaries*

Introduction

With the continuous growth in the scale and complexity of software, compiler optimization has become increasingly important in improving program execution efficiency and resource utilization. However, numerous key optimization problems, such as register allocation and instruction scheduling, face significant algorithmic design challenges due to their inherent computational complexity. Computational complexity theory provides a systematic theoretical framework for these problems. Through the classification of complexity classes and reduction techniques, it reveals the intrinsic difficulty and computational resource requirements of optimization tasks, offering scientific guidance for algorithm design. In-depth analysis of complexity boundaries and modeling expressions not only promotes a better understanding of the solvability and approximability of optimization problems but also provides theoretical support for the enhancement of static optimization strategies and constraint-solving techniques. This paper focuses on the theoretical application of complexity theory in compiler optimization, aiming to explore its guiding significance for optimization algorithm design and its potential for improving the performance of practical compilation systems, promoting the deep integration of theoretical computer science and compiler technology, and meeting the demands of modern high-performance computing.

1. The Fundamental Structure and Core Propositions of Computational Complexity Theory

1.1 The Classification System and Model Construction of Computational Complexity

Computational complexity theory, as an important branch of theoretical computer science, aims to systematically characterize the resource consumption characteristics in the process of solving computational problems, primarily involving two dimensions: time complexity and space complexity. This theory, based on the Turing machine as a formal computational model, has developed a rich system of complexity classes. The polynomial time complexity class (P) is widely regarded as the

representative of "efficiently solvable" problems, encompassing decision problems that can be computed in polynomial time on a deterministic Turing machine. In contrast, the nondeterministic polynomial time class (NP) includes decision problems whose solutions can be verified in polynomial time on a nondeterministic Turing machine. The subtlety of this definition lies in the distinction between the "verification" process and the "solving" process, revealing the deeper structure of computational complexity [1].

Moreover, space complexity classes (such as PSPACE) extend the consideration of computational resources, encompassing problems solvable within polynomial space limitations, reflecting the diversity and complexity of the computational resource dimension. Theoretically, the PSPACE class contains both P and NP classes, reflecting the hierarchical progression of resource requirements in computational problems. The reduction techniques within the complexity classification system, especially polynomial-time reductions and logarithmic space reductions, provide the core tools for defining the relationships between different complexity classes, allowing the complexity mappings between problems to be revealed through mutual transformations. This system not only supports the rigor of theoretical computation but also lays a solid theoretical foundation for the assessment of computational difficulty in practical application domains.

1.2 The Relationships and Hierarchical Structure of Complexity Classes

The hierarchical structure of complexity classes is the theoretical cornerstone for understanding the differences in computational difficulty. The inclusion relationships between these classes reflect the fundamental differences in resource consumption for computational tasks. The class P, as the core set of low complexity, represents traditionally "efficiently computable" problems, while the NP class extends this scope by incorporating verifiability into complexity analysis. The relationship between P and NP remains one of the most challenging core problems in computational complexity, and it has yet to be fully resolved. Its importance lies in its direct impact on understanding and classifying the solvability of a series of key computational tasks. The PSPACE class further broadens the computational perspective, encompassing all solvable problems within polynomial space constraints, forming a natural hierarchy in complexity theory that reveals the decisive role of spatial resources in computational complexity.

Reduction theory plays a decisive role in this hierarchical system. Polynomial-time reductions not only define NP-completeness but also provide standardized tools for analyzing the inclusion and equivalence relationships between complexity classes. Through reductions, the difficulty of a problem can be proven, allowing the theoretical community to clearly define the intrinsic limits of resource consumption for that problem, which then guides the theoretical positioning of algorithm design in related fields. The hierarchical relationships of complexity classes help researchers grasp the difficulty level of specific optimization tasks from a theoretical perspective in fields like compiler optimization, providing scientific foundations for constructing reasonable algorithm frameworks and decision strategies. Additionally, this understanding has spurred innovative developments in complexity theory within emerging computational scenarios [2].

1.3 The Abstraction of Decision Problems and Function Problems in the Compiler Context

The core tasks in compiler optimization can essentially be abstracted as decision problems and function problems. These two problem paradigms provide clear entry points for the theoretical analysis of computational complexity. Decision problems typically involve determining whether a certain optimization condition is satisfied, such as in register allocation, where the task is to determine whether a feasible allocation scheme exists under given resource constraints, or in instruction scheduling, where the goal is to check whether specific dependency relationships allow for a certain instruction ordering. The theoretical description of these problems makes it easier to map optimization tasks into complexity classes, evaluating their solvability and computational resource requirements.

Function problems further extend this paradigm by focusing on how to generate a specific optimal or approximate solution given the input, reflecting the algorithmic design challenges faced by real-world compilers in generating high-quality code.

This abstraction facilitates the deep integration of compiler optimization with complexity theory, ensuring that complexity analysis not only remains at the theoretical level but also directly informs the algorithm selection and optimization strategies in compiler design. By formalizing compiler tasks into typical problems within the complexity framework, researchers can apply a rich array of theoretical

tools to analyze their solution boundaries, identifying the feasibility and efficiency bottlenecks of algorithms. This methodology not only helps clarify the inherent difficulty of optimization problems but also lays a solid foundation for future theoretical breakthroughs aimed at more efficient compilation technologies, while simultaneously driving the innovative transformation of computational complexity theory into practical engineering applications.

2. The Complexity Modeling and Theoretical Characterization of Compiler Optimization Problems

2.1 Complexity Boundary Analysis of Key Optimization Tasks in Compilation

Core tasks in compiler optimization, such as register allocation, instruction scheduling, and basic block ordering, involve a deep trade-off between program execution efficiency and resource utilization. The computational complexity characteristics of these tasks directly influence the design and implementation of optimization algorithms. Register allocation problems are typically classified as NP-complete problems, with difficulty arising from the need to achieve optimal coverage of variable lifetimes under limited register resources. Instruction scheduling, another typical optimization task, involves handling dependencies and execution order, and its computational complexity also has significant theoretical boundaries, often proven to be NP-hard problems. The clear characterization of the complexity boundaries of these tasks reveals the inevitable computational bottlenecks in optimization algorithm design, providing starting points for approximation algorithms and heuristic strategies under theoretical guidance [3].

Research into complexity boundaries goes beyond simply classifying problems as easy or difficult. It delves deeper into the polynomial-time solvability and unsolvability conditions of specific problem subclasses, laying the foundation for fine-grained algorithmic complexity analysis. For compiler optimization tasks, precise analysis of complexity boundaries helps identify the feasible algorithm design space and its limitations, while also providing theoretical support for future optimization of algorithm complexity and resource scheduling strategies. Such boundary characterization promotes the intersection of complexity theory and compiler optimization, advancing the knowledge transfer between theory and engineering practice.

2.2 The Theoretical Guiding Role of Computational Complexity in Static Optimization Strategies

Static optimization, as a key process in the compilation of program code analysis and transformation, exhibits its complexity characteristics through the structured understanding and processing of control flow and data flow. Complexity theory provides a rigorous theoretical framework for static analysis tasks, particularly in the design of optimization strategies such as control flow graph construction, data dependency analysis, and unreachable code elimination. Complexity metrics become an important criterion for selecting algorithmic paths. By identifying the complexity class of the analysis task, the computational resource requirements of static optimization strategies can be scientifically defined, which in turn guides the feasibility evaluation and performance improvement of algorithms.

The static optimization guided by complexity theory focuses not only on theoretical solvability but also promotes hierarchical optimization designs targeting complexity bottlenecks. This includes the development of polynomial-time algorithms under specific constraints and the decomposition and modularization of tasks with higher complexity. This theoretical perspective deepens the understanding of resource limitations in static analysis, facilitating the shift in compiler optimization from being experience-driven to being guided by theory, thereby improving the systematic and scalable nature of static optimization strategies. The introduction of complexity-guided mechanisms has provided innovative momentum for enhancing compiler analysis precision and efficiency [4].

2.3 Constraint Optimization Models and Complexity Compression Mechanisms in the Compilation Process

Constraint expression and solving in the compilation optimization process are crucial for achieving efficient code generation, involving the formalization of optimization problems into models such as Constraint Satisfaction Problems (CSP), Boolean Satisfiability Problems (SAT), or Integer Linear Programming (ILP). Complexity theory provides key guidance for the construction and simplification

of these constraint models, particularly showing significant value in constraint compression and complexity reduction. By identifying redundant and irreducible structures within constraint sets, complexity compression mechanisms effectively reduce the size of the solution space, enhancing the computational efficiency of optimization algorithms.

The theoretical foundation of complexity compression mechanisms relies primarily on reducibility theory, graph compression techniques, and parameterized complexity analysis. By analyzing the path magnitudes, node degree distributions, and dependency core structures of constraint graph structures, it is possible to identify the least impactful weak dependency paths or redundant conditions, allowing for the stripping and restructuring of the constraint set. Furthermore, this mechanism demonstrates significant advantages in hierarchical modeling strategies: low-complexity subproblems can be solved first to reduce the size of the main problem, while high-complexity regions are locally processed through boundary compression or parameter reduction, thus lowering the overall complexity level. This strategy not only enhances the compiler's optimization capabilities for large-scale programs but also strengthens the modularity and maintainability of the algorithm structure, laying a solid foundation for constructing optimization-solving systems with theoretical explainability and computational scalability.

3. Computability-Oriented Optimization Strategies and Complexity-Guided Algorithm Design

3.1 Theoretical Scheduling Methods for Polynomial-Time Solvable Optimization Tasks

Although some tasks in compiler optimization fall within the category of NP-hard problems, under specific constraints or limited input sizes, they can be transformed into subproblems that are solvable in polynomial time. These problems typically exhibit clear structural features, such as linear dependency graphs, tree-like data dependencies, or deterministic resource allocation patterns. By utilizing these structural features, theoretical scheduling methods leverage dynamic programming, greedy strategies, and graph algorithms to achieve efficient scheduling of control flow and data flow during the compilation process. Especially in optimization tasks such as loop unrolling, basic block ordering, and instruction pipeline construction, the time complexity of the algorithm can be effectively controlled, ensuring the collaborative improvement of both compilation efficiency and generated code quality [5].

Furthermore, the core value of complexity theory in scheduling algorithm design lies in identifying the "solvability window" of a task through fine-grained classification of complexity classes and reduction analysis. For example, under the premise that the task scheduling graph is a Directed Acyclic Graph (DAG), the scheduling problem can be reduced to a priority scheduling model, enabling the construction of a near-optimal solution within polynomial time. Additionally, complexity analysis reveals the complexity transition points in specific scheduling scenarios, such as the exponential increase in complexity when transitioning from a single-processor environment to a multi-core architecture. These theoretical results provide clear guidance for the applicable boundaries of scheduling methods.

The optimization framework based on theoretical scheduling methods not only provides reusable algorithm modules for existing compiler architectures but also presets universal scheduling interfaces for the compiler design of new programming languages and architectures. This scheduling mechanism, centered on complexity classification, strengthens the systematic and scalable nature of algorithm design, allowing compiler optimization to evolve away from empirical rule-based constructions toward formal modeling and complexity control. This strategy holds significant forward-looking implications for building theory-driven next-generation compiler systems.

3.2 Complexity Control Mechanisms of Approximation Algorithms and Heuristic Methods

When the computational complexity of an optimization problem has been proven to be NP-complete or falls into higher complexity classes, pursuing an exact solution inevitably leads to explosive growth in computational resource requirements. In this context, approximation algorithms and heuristic methods provide practical and efficient alternative paths. Complexity theory supports the design and evaluation of these algorithms by precisely defining approximation ratios, performance bounds, and worst-case error margins. For example, problems related to compiler resource allocation, such as minimum dominating set and graph coloring, can be solved using approximation algorithms that provide high-quality solutions within acceptable error bounds, thus balancing real-time performance and computational efficiency during the compilation process.

The core of the complexity control mechanism lies in the predictability and systematization of algorithm performance. By introducing polynomial-time approximation schemes (PTAS), greedy approximation structures, and LP relaxation techniques, designers can effectively control the growth of complexity without sacrificing the overall structural stability. In compiler design, tasks such as code layout optimization and cache-sensitive scheduling can adopt multi-stage heuristic methods combined with local search and cost-function-driven approaches, thereby avoiding the uncontrollable complexity found in global search strategies. Furthermore, heuristic strategies can also integrate machine learning prediction models, dynamically adjusting the search paths based on historical samples or feedback from the objective function, further enhancing adaptability and solution quality [6].

Compared to traditionally experience-driven heuristic algorithms, approximation and heuristic methods based on complexity theory are more systematic and verifiable. This shift not only improves the generalizability and transferability of the solutions but also strengthens the mapping between optimization strategies and problem structures, laying the algorithmic foundation for subsequent automatic compiler optimization strategy generation. By combining complexity compression techniques and fuzzy decision mechanisms, optimization systems with progressive adaptive capabilities can be built, achieving optimal performance harmonization across different program scales and runtime environments.

3.3 Structural Insights from Complexity Theory for Compiler Backend Optimization

The compiler backend is a critical phase that transforms intermediate representations into target code, involving high-complexity tasks such as instruction selection, instruction scheduling, register allocation, and machine code generation. The nature of these tasks typically manifests as constraint satisfaction problems, resource allocation problems, and combinatorial optimization problems, with their computational complexity significantly higher than that of frontend syntax analysis and semantic transformations. Complexity theory, through systematic analysis of problem structures and resource boundaries, provides an extensible theoretical toolkit to reveal the inherent irreducible complexity in the optimization process, thereby guiding algorithm design toward the direction of "controllable complexity."

From a structural perspective, complexity analysis reveals the coupling mechanisms and hierarchical dependencies between backend optimization tasks, such as the interdependent relationship between register allocation and instruction scheduling. Based on this theoretical insight, modular reconstruction methods can be employed to decompose high-complexity tasks into multiple sub-tasks of lower complexity, using an abstraction layer of complexity (such as SSA form) in intermediate representations to decouple and optimize. This strategy not only reduces the peak complexity of the overall compilation path but also makes backend optimization more adaptable to the development trends of heterogeneous computing platforms and parallel architectures.

Complexity theory has also driven a redefinition of the "compilation optimality" boundary. Beyond traditional objective functions such as minimizing execution time, modern backend optimization increasingly incorporates multi-dimensional objectives such as power consumption control, cache friendliness, and security. Complexity theory, through multi-objective optimization modeling and Pareto boundary analysis, provides a framework that offers both theoretical completeness and strategic flexibility for compiler construction. In the future, as program scales and hardware architecture complexities continue to grow, complexity theory will play a deeper structural guiding role in automating optimization parameter selection, algorithm combination searches, and collaborative scheduling of compilation tasks.

Conclusion

This paper systematically explores the theoretical application of computational complexity theory in compiler optimization, clarifying the core role of complexity classification and reduction techniques in characterizing the difficulty of key optimization tasks. The paper analyzes scheduling methods for polynomial-time solvable tasks and the complexity control mechanisms of approximation and heuristic strategies for NP-hard problems. At the same time, complexity theory’s structural insights into the compiler backend optimization process have facilitated the modular design and collaborative optimization of algorithms. Future research could further deepen the integration of complexity theory with cutting-edge technologies such as machine learning and automated reasoning, exploring adaptive compiler optimization frameworks for large-scale heterogeneous computing environments. Additionally,

the application prospects of complexity theory in dynamic optimization and real-time compilation are vast, with the potential to drive compiler technology toward higher levels of intelligence and efficiency, contributing to the development of next-generation high-performance software systems.

References

[1] Fang, Jianbin, et al. "Systematic Ability-Oriented Parallel Compiler Optimization Course Teaching Reform." Computer Education 05 (2025): 96-99.

[2] Xiong, Kang, et al. "Airborne Deep Computation Compiler Optimization for UAV Collaborative Positioning." Computer Science and Exploration 19.01 (2025): 141-157.

[3] Zhang, Yanshuo, et al. "Case-Based Design of Computational Complexity Theory Courses Based on Comparative Analysis." Journal of Beijing University of Electronic Science and Technology 32.02 (2024): 87-98.

[4] Pei, Xue, et al. "Compilation Optimization and Implementation of High-Order Cryptographic Operators on FPGA." Computer Science 51.S2 (2024): 785-795.

[5] Zhang, Hongbin, et al. "AutoConfig: An Automatic Configuration Mechanism for Deep Learning Compilation Optimization." Journal of Software 35.06 (2024): 2668-2686.

[6] Gao, Xiangyu. "Research on Several Basic Issues of Big Data Computational Complexity Theory." 2024. Harbin Institute of Technology, PhD dissertation.