



北京大学

本科生毕业论文

题目：基于文档策略提取的代码自动优化方法的
设计与实现

Design and Implementation of an
Automatic Code Optimization Method
Based on Document Strategy Extraction

姓 名：	肖千与
学 号：	2200012932
院 系：	信息科学技术学院
专 业：	信息与计算科学
导师姓名：	熊英飞

二〇二六年五月

北京大学本科毕业论文导师评阅表

学生姓名	肖千与	本科院系	信息科学技术学院	论文成绩	
学生学号	2200012932	本科专业	信息与计算科学	(等级制)	
导师姓名	熊英飞	导师单位/ 所在学院	计算机学院	导师职称	长聘副教授
论文题目	中文	基于文档策略提取的代码自动优化方法的设计与实现			
	英文	Design and Implementation of an Automatic Code Optimization Method Based on Document Strategy Extraction			
导师评语 (包含对论文的性质、难度、分量、综合训练等是否符合培养目标的目的等评价)					
导师签名: 年 月 日					

版权声明

任何收存和保管本论文各种版本的单位和个人，未经本论文作者同意，不得将本论文转借他人，亦不得随意复制、抄录、拍照或以任何方式传播。否则，引起有碍作者著作权之问题，将可能承担法律责任。

摘要

代码性能优化是软件工程中的重要问题。低效代码不仅会增加程序运行时间和计算资源消耗，还可能影响系统的稳定性和用户体验。近年来，大语言模型在代码理解、代码生成和程序修复等任务中表现出较强能力，也逐渐被应用于自动化代码优化场景。然而，直接要求大语言模型对代码进行优化往往存在目标不明确、优化策略选择困难以及结果不稳定等问题。**SemOpt** 工具针对这一问题，将大语言模型与静态程序分析相结合，通过构建代码优化策略库、生成 **Semgrep** 静态分析规则，并利用规则定位潜在优化位置，从而引导大语言模型完成代码优化。

本文在 **SemOpt** 原有框架的基础上，围绕其优化策略获取流程进行了扩展。原有流程主要依赖从代码提交记录中提取和归纳优化策略，而本文设计并实现了一种面向书籍和文档的优化策略提取方法。该方法利用大语言模型从自然语言技术资料中识别与代码性能优化相关的知识，抽取策略描述、适用条件、优化目标和示例信息，并将其转换为 **SemOpt** 后续规则生成与代码优化模块可以使用的结构化策略表示。通过该扩展，**SemOpt** 的策略来源不再局限于 **GitHub commit**，而可以从更广泛的文本知识源中获取优化经验。

本文的主要工作包括：分析 **SemOpt** 原有基于代码提交的策略获取流程；设计面向文档资料的策略抽取流程与结构化策略格式；实现从书籍和技术文档中抽取、解析、筛选并加载优化策略的功能；最后通过实例验证该扩展能够支持从非代码提交资料中获取策略并应用于代码优化任务。本文工作提高了 **SemOpt** 策略库构建的开放性和可扩展性，使其能够利用更丰富的软件工程知识来源，为后续构建面向多源知识的自动化代码优化系统提供了基础。

关键词：大语言模型；代码优化；静态分析

ABSTRACT

Code performance optimization is a critical issue in software engineering. Inefficient code not only increases program execution time and computational resource consumption, but may also affect system stability and user experience. In recent years, large language models (LLMs) have demonstrated strong capabilities in tasks such as code understanding, code generation, and program repair, and have gradually been applied to automated code optimization scenarios. However, directly prompting LLMs to optimize code often leads to problems such as unclear optimization goals, difficulty in selecting appropriate optimization strategies, and unstable results. To address these issues, the SemOpt tool combines LLMs with static program analysis by constructing a library of code optimization strategies, generating Semgrep static analysis rules, and using these rules to locate potential optimization sites, thereby guiding LLMs to perform code optimization.

In this thesis, we extend the original SemOpt framework by focusing on its strategy acquisition pipeline. The original pipeline primarily relies on extracting and summarizing optimization strategies from code commit histories. We design and implement a method for extracting optimization strategies from books and technical documents. This method uses LLMs to identify knowledge related to code performance optimization from natural language technical materials, extract strategy descriptions, applicable conditions, optimization goals, and example code, and then transform them into structured strategy representations that can be used by SemOpt's subsequent rule generation and code optimization modules. With this extension, the source of optimization strategies for SemOpt is no longer limited to GitHub commits, but can draw upon a broader range of textual knowledge sources.

The main contributions of this thesis include: analyzing SemOpt's original commit-based strategy acquisition process; designing a document-oriented strategy extraction pipeline and a structured strategy format; implementing the functions of extracting, parsing, filtering, and loading optimization strategies from books and technical documents; and demonstrating through case studies that the extension supports obtaining strategies from non-commit sources and applying them to code optimization tasks. This work improves the openness and scalability of SemOpt's strategy library construction, enabling it to leverage richer sources of software engineering knowledge and laying the foundation for building a multi-source knowledge-driven automated code optimization system in the future.

KEY WORDS: Large Language Models; Code Optimization; Static Analysis

目 录

1. 引言	1
2. 代码优化概述	4
2.1 基于规则的代码优化方法.....	4
2.2 数据流分析与经典编译器优化	5
2.3 循环优化与局部性优化.....	6
2.4 超优化与搜索式优化.....	8
2.5 等式饱和与 E-graph 优化	9
2.6 机器学习驱动的编译器优化.....	9
2.7 大语言模型驱动的代码优化.....	10
2.8 研究空白与本文动机.....	13
3. SemOpt 方法介绍.....	14
3.1 整体流程	14
3.2 策略库构建	14
3.3 静态分析规则生成.....	14
3.4 代码优化	15
3.5 方法特点	15
4. 面向文档的优化策略提取方法	16
4.1 整体流程	16
4.2 文档预处理与文本分块.....	17
4.3 基于大语言模型的策略抽取.....	17
4.4 多次生成与一致性筛选.....	19
4.5 代码示例规范化	19
4.6 Semgrep 规则生成与验证	20
4.7 与原始 SemOpt 的关系	23
5. 实验与结果对比	25

5.1 实验设置 25

5.1.1 实验环境与基准 25

5.1.2 评估指标 25

5.1.3 优化过程 25

5.2 实验结果 26

5.2.1 Redis 优化结果 26

5.2.2 RocksDB 优化结果 26

5.2.3 结果分析 27

6. 相关工作 28

6.1 基于大语言模型的代码优化..... 28

6.2 基于静态分析的程序优化..... 28

7. 结论 29

参考文献 30

致谢 33

北京大学学位论文原创性声明和使用授权说明..... 34

1. 引言

随着软件系统规模和复杂度的不断提升，程序性能已经成为衡量软件质量的重要因素之一。在服务器端系统、数据库、编译器、图形处理、嵌入式系统和高性能计算等场景中，代码中的低效实现可能导致运行时间增加、资源利用率下降，甚至影响系统的稳定性和用户体验。因此，如何发现并修复性能低效代码，一直是软件工程和程序分析领域的重要问题。

传统代码性能优化通常依赖开发者经验、性能分析工具和人工代码审查。开发者需要先通过 **profiling** 工具定位热点代码，再结合程序语义判断是否存在优化机会，最后选择合适的优化方式。例如，在循环中减少重复计算、替换低效数据结构、调整条件判断顺序、避免不必要的内存分配、使用更合适的 **API**，或根据语言和编译器特性改写代码。这一过程通常需要较强的专业知识和工程经验。对于大型项目而言，人工持续发现和修复性能问题的成本较高，因此自动化代码优化具有重要研究价值。

近年来，大语言模型（**Large Language Models, LLMs**）在软件工程领域得到了广泛关注。大语言模型已经被用于代码生成、程序修复、测试生成、代码摘要、缺陷检测和软件维护等多类软件工程任务，并有潜力优化软件工程过程与结果。这一趋势说明，大语言模型已经逐渐从通用自然语言处理工具发展为软件开发过程中的重要智能化辅助技术。

在代码优化任务中，大语言模型的优势主要体现在代码理解和代码生成能力上。相比传统规则方法，大语言模型可以根据上下文理解代码意图，并生成形式更灵活的优化结果。例如，模型可能识别出某段代码中存在冗余计算，或根据 **API** 语义建议使用更高效的替代写法。然而，直接要求大语言模型对代码进行优化并不稳定。一方面，代码优化方向具有多样性，同一段代码可能存在多种潜在优化方式；另一方面，模型生成的代码可能改变原有语义，或者只进行表面改写而没有带来实际性能收益。因此，仅依赖直接提示的大语言模型很难稳定完成代码优化任务。

为提高大语言模型代码优化的可靠性，已有研究开始引入优化示例或检索增强方法。一类常见方法是从开源项目的历史优化提交中收集代码修改示例，然后在优化新代码时检索相似示例，并将其作为上下文提供给大语言模型，如现有的 **RAPGen** 工具^[1]。这种方法能够为模型提供一定的优化参考，但仍存在明显局限。由于相同的优化策略可能表现为语法结构差异很大的代码修改，基于文本相似度或代码相似度的检索方法不一定能够找到真正相关的优化示例。同时，检索到的示例并不一定明确指出优化策略的适用条件，也难以精确定位目标代码中应该被优化的位置。

SemOpt^[2]针对上述问题提出了一种结合大语言模型与静态程序分析的代码优化框架。与直接提示或单纯检索相似代码示例的方法不同，**SemOpt** 首先从真实代码优化提交中

提取优化策略，并利用大语言模型对策略进行总结和聚类；随后，为每类优化策略生成 **Semgrep** 静态分析规则，用于匹配该策略适用的代码位置；最后，系统将待优化代码、匹配到的代码位置 and 对应优化策略共同输入大语言模型，由模型生成优化后的代码。通过这种方式，**SemOpt** 将大语言模型的语义理解能力与静态分析工具的模式匹配能力结合起来，使优化过程不再完全依赖模型的自由生成，而是受到明确优化策略和代码位置信息的引导。

相比已有方法，**SemOpt** 的优势主要体现在三个方面。第一，**SemOpt** 不只是检索语法相似的历史代码片段，而是通过静态分析规则刻画优化策略的适用条件，因此能够更准确地将优化策略应用到目标代码中。第二，**SemOpt** 同时向大语言模型提供优化位置和优化策略，降低了模型在整段代码中盲目寻找优化点的难度，从而提高生成结果的针对性。第三，**SemOpt** 的策略库可以覆盖多种优化模式，而不局限于单一类型的 API 替换或局部规则。原始研究表明，**SemOpt** 在多个大语言模型上均优于 **Direct Prompting**、**RAG** 和 **RAPGen** 等基线方法，并能够在真实大型 C/C++ 项目中取得性能提升^[2]。

然而，**SemOpt** 原有策略获取方式仍存在一个重要限制：其优化策略主要来自 **GitHub** 等开源仓库中的历史优化提交。这种方式具有较强的现实基础，因为真实提交能够反映开发者实际进行过的性能优化修改。但与此同时，它也会限制策略库的知识来源。许多有价值的优化知识并不一定会以 **commit** 的形式出现。例如，编程语言书籍可能总结了某类语言特性的性能影响；官方 API 文档可能说明某个接口在特定场景下更高效；性能优化手册可能给出缓存、内存分配、循环结构、字符串处理或容器选择方面的通用建议；技术文档和开发者指南中也可能包含大量可迁移的优化经验。如果 **SemOpt** 只能从历史代码提交中获取策略，就难以充分利用这些文本资料中蕴含的优化知识。

基于此，本文在 **SemOpt** 原有框架基础上，对其策略获取模块进行扩展，提出一种从书籍和技术文档中提取代码优化策略的方法。该方法的核心思想是：将自然语言形式的优化知识转化为 **SemOpt** 可以使用的结构化策略。具体而言，系统首先读取书籍、技术文档或其他文本资料中的相关内容；随后利用大语言模型识别其中与代码性能优化相关的片段，并抽取优化策略名称、策略描述、适用条件、优化目标和示例信息；之后对抽取得到的策略进行格式化和筛选，使其能够进入 **SemOpt** 后续的规则生成流程；最后，结合 **Semgrep** 规则和大语言模型，将这些来自文档的策略应用到目标代码优化任务中。

与原始 **SemOpt** 相比，本文扩展后的方法具有更开放的策略来源。原始 **SemOpt** 主要从历史代码修改中学习优化策略，而本文方法进一步引入书籍和文档中的显式优化知识，使策略库可以同时吸收实践经验和文本知识。前者来自真实项目中的开发者修改，具有较强的工程真实性；后者来自系统化整理的文档资料，往往具有更清晰的解释和适用条件。

二者结合后，可以提高策略库的覆盖范围，也有助于发现那些尚未在当前代码库历史提交中出现、但在理论或文档中已经被总结的优化方法。

本文工作的意义主要体现在以下几个方面。首先，该工作突破了 **SemOpt** 原有策略来源对 **GitHub commit** 的依赖，使系统能够从更广泛的软件工程知识源中获取优化策略。其次，文档来源的策略通常具有较强的可解释性，便于研究者理解策略含义、检查适用条件并进行人工修正。再次，文档策略可以作为可控输入参与实验，便于比较不同资料来源、不同策略集合对代码优化效果的影响。最后，该扩展使 **SemOpt** 更接近一个多源知识驱动的代码优化框架，为后续融合开源提交、官方文档、编程书籍和专家经验提供了基础。

综上所述，本文的主要工作是在 **SemOpt** 原有基于代码提交的策略获取流程之外，设计并实现一种面向书籍和技术文档的优化策略提取机制。本文首先分析 **SemOpt** 原有策略库构建和代码优化流程，然后设计文档策略抽取与结构化表示方法，接着实现从文本资料中抽取、筛选和加载优化策略的功能，并将其接入后续规则生成和优化流程，最后验证从文档中提取到策略的有效性。通过该工作，**SemOpt** 的策略库构建方式得到扩展，从而提高了自动化代码优化系统的知识覆盖范围、可扩展性和可维护性。

2. 代码优化概述

代码优化是编译器和软件工程领域中的核心问题之一，其目标是在保持程序语义不变的前提下，提升程序的执行效率、降低内存开销、减少代码体积，或改善程序在特定硬件平台上的运行表现。一般而言，给定原始程序 P 和优化后程序 P' ，一次正确的代码优化应满足如下语义保持条件：

$$\forall x, \llbracket P \rrbracket(x) = \llbracket P' \rrbracket(x), \quad (1)$$

其中， $\llbracket \cdot \rrbracket$ 表示程序语义， x 表示程序输入。也就是说，对于所有合法输入，优化前后的程序应产生一致的可观察行为。

从搜索的角度看，代码优化可以被形式化为在语义等价程序空间中寻找代价最小的程序：

$$P^* = \arg \min_{P' \in \mathcal{E}(P)} C(P'), \quad (2)$$

其中， $\mathcal{E}(P)$ 表示与程序 P 语义等价的程序集合， $C(P')$ 表示代价函数，可以是执行时间、指令数、内存访问次数、缓存缺失率或能耗等指标。

早期代码优化研究主要集中在编译器内部的程序分析与变换。**Allen** 和 **Cocke**^[3]对优化变换进行了系统整理，提出了许多后来成为编译器标准优化过程的技术，例如公共子表达式消除、代码移动、强度削弱和循环优化等。**Kildall**^[4]提出的数据流分析框架则为全局程序优化提供了统一的理论基础。由此可见，代码优化并不是简单的文本替换，而是依赖程序分析、语义保持和代价建模的系统性问题。

2.1 基于规则的代码优化方法

基于规则的优化方法是传统编译器中最经典、最重要的一类优化方法。其基本思想是由编译器专家手工总结程序中的低效模式，并将其编码为一组可匹配、可执行的变换规则。一个典型规则可以表示为：

$$pattern_{old} \Rightarrow pattern_{new}, \quad (3)$$

并满足如下条件：

$$\llbracket pattern_{old} \rrbracket = \llbracket pattern_{new} \rrbracket \quad \wedge \quad C(pattern_{new}) < C(pattern_{old}). \quad (4)$$

也就是说，规则左侧和右侧必须语义等价，同时右侧代码在某种代价模型下更加高效。

例如，代数化简是一类最常见的规则：

```
x = y + 0;
```

可以被改写为：

```
x = y;
```

类似地，乘以 2 在某些整数语义下可以被改写为左移操作：

```
x = y * 2;
```

可以被改写为：

```
x = y << 1;
```

然而，即使是这种看似简单的规则，也需要严格的适用条件。例如，对于有符号整数，左移可能涉及溢出和未定义行为；对于不同硬件平台，乘法和移位的实际开销也可能不同。因此，基于规则的优化不仅需要描述“如何变换”，还需要描述“什么时候可以变换”和“什么时候值得变换”。

基于规则的方法具有明显优点。首先，规则具有较强的可解释性，开发者可以理解每条规则的优化意图。其次，规则的行为相对可控，便于调试和验证。再次，规则一旦被写入编译器，可以长期稳定地应用于大量程序。然而，这类方法也存在明显局限。手写规则需要大量专家经验，开发成本较高；规则通常只能覆盖特定语法模式，难以识别语义相同但形式差异较大的优化机会；此外，多条优化规则之间还可能存在着复杂交互，导致优化顺序影响最终结果。

2.2 数据流分析与经典编译器优化

许多经典编译器优化依赖数据流分析。数据流分析通过在控制流图上计算程序属性，判断某个变量、表达式或语句在程序某一点是否具有特定性质。对于一个基本块 B ，前向数据流分析的一般形式为：

$$IN[B] = \bigwedge_{P \in pred(B)} OUT[P], \quad (5)$$

$$OUT[B] = f_B(IN[B]), \quad (6)$$

其中， $pred(B)$ 表示基本块 B 的前驱集合， f_B 表示基本块 B 的传递函数， $\wedge$ 表示格上的合并操作。

以到达定值分析为例，其传递函数通常可以写作：

$$OUT[B] = GEN[B] \cup (IN[B] - KILL[B]). \quad (7)$$

其中, $GEN[B]$ 表示基本块 B 中生成的定值, $KILL[B]$ 表示被该基本块覆盖或杀死的定值。

数据流分析支撑了许多经典优化。例如, 可用表达式分析可以用于公共子表达式消除; 活跃变量分析可以用于死代码消除和寄存器分配; 到达定值分析可以用于常量传播和复制传播。

例如:

```
int x = 4;
int y = x + 3;
```

如果编译器可以证明在第二条语句处 x 的值一定为 4, 则可以将代码优化为:

```
int x = 4;
int y = 7;
```

进一步地, 如果变量 x 在后续程序中不再被使用, 那么第一条赋值语句还可以被死代码消除。这个例子说明, 编译器优化往往不是孤立发生的。一次优化可能创造新的优化机会, 而最终结果取决于多个优化阶段的排列顺序。这一问题通常被称为优化阶段排序问题。

2.3 循环优化与局部性优化

循环结构通常是程序执行时间最集中的部分, 因此循环优化在代码优化中占据重要位置。常见循环优化包括循环不变式外提、循环展开、循环融合、循环分裂、循环交换和循环分块等^{[5][6]}。

循环不变式外提的基本思想是将循环内部每次迭代结果都相同的表达式移动到循环外部。例如:

```
for (int i = 0; i < n; i++) {
    a[i] = b[i] + x * y;
}
```

如果 x 和 y 在循环中不发生变化, 则可以改写为:

```
int t = x * y;
for (int i = 0; i < n; i++) {
    a[i] = b[i] + t;
}
```

该优化的核心规则可以概括为:

$$expr \in LoopInvariant(L) \Rightarrow move(expr, preheader(L)). \quad (8)$$

其中, $LoopInvariant(L)$ 表示循环 L 中的循环不变表达式, $preheader(L)$ 表示循环进入前的前置基本块。

循环分块则主要用于改善缓存局部性。例如, 对于矩阵遍历:

```
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        C[i][j] += A[i][j];
    }
}
```

可以改写为分块形式:

```
for (int ii = 0; ii < N; ii += B) {
    for (int jj = 0; jj < N; jj += B) {
        for (int i = ii; i < min(ii + B, N); i++) {
            for (int j = jj; j < min(jj + B, N); j++) {
                C[i][j] += A[i][j];
            }
        }
    }
}
```

分块后的程序在一个较小的数据块上连续计算, 从而提高缓存复用率, 减少缓存缺失。

多面体模型为循环优化提供了更加形式化的数学框架^{[7][8]}。对于仿射循环嵌套, 其迭代空间可以表示为整数多面体:

$$D = \{i \in \mathbb{Z}^d \mid Ai + b \geq 0\}. \quad (9)$$

调度函数将每个迭代点映射到执行时间:

$$\theta_S(i) = T_S i + t_S. \quad (10)$$

为了保证变换合法, 调度必须保持程序依赖关系。如果语句实例 $S(i)$ 必须在 $T(j)$ 之前执行, 则需要满足:

$$\theta_T(j) - \theta_S(i) \geq 0. \quad (11)$$

多面体优化相比局部窥孔优化具有更强的全局性, 能够对整个循环嵌套进行统一建模

和变换。但是，它通常要求程序具有较规则的控制流和内存访问模式，因此对一般程序的适用范围仍然有限。

2.4 超优化与搜索式优化

超优化（**superoptimization**）试图自动寻找实现某一计算功能的最优代码序列^[9]。与传统基于规则的方法不同，超优化并不完全依赖人工编写的规则，而是在候选程序空间中搜索语义等价且代价更低的程序。其基本形式可以表示为：

$$Q^* = \arg \min_{Q \in \mathcal{L}} C(Q), \quad (12)$$

并满足：

$$\forall x, \quad \llbracket Q \rrbracket(x) = \llbracket P \rrbracket(x). \quad (13)$$

其中， $\mathcal{L}$ 表示候选程序空间， Q 表示候选程序， P 表示原始程序。

超优化的优势在于能够发现人类专家未必会手写的非显然优化。例如，某些分支判断可以被替换为无分支算术表达式：

```
if (a < b) {
    c++;
}
```

在某些语义和平台条件下，可以改写为：

```
c += (a < b);
```

这种变换可能减少分支预测失败带来的开销。但是，它是否真正有利仍然依赖具体架构和运行时数据分布。

STOKE 等系统进一步将超优化建模为随机搜索问题^[10]，其代价函数可以写作：

$$Cost(R; T) = Eq(R; T) + \lambda \cdot Perf(R; T), \quad (14)$$

其中， T 表示目标程序， R 表示候选重写结果， $Eq(R; T)$ 衡量语义不一致程度， $Perf(R; T)$ 衡量性能代价， λ 是权重参数。

超优化方法可以发现高质量的底层代码优化，但其主要问题是搜索空间增长极快。随着程序长度增加，候选程序数量呈指数级增长。因此，超优化通常更适合小规模代码片段或指令级优化。

2.5 等式饱和与 E-graph 优化

传统规则优化通常采用破坏式重写方式，即一旦某条规则被应用，原始表达式就会被替换掉，其他可能的等价形式可能因此丢失。等式饱和（equality saturation）试图解决这一问题。它使用 E-graph 同时表示多个语义等价表达式，并通过不断应用重写规则扩展等价类，最后再通过代价模型选择最优表达式。

例如，给定如下规则：

$$x + 0 \Rightarrow x, \quad (15)$$

$$x \times 2 \Rightarrow x + x, \quad (16)$$

$$x \times 2 \Rightarrow x \ll 1, \quad (17)$$

E-graph 可以同时表示如下等价关系：

$$x \times 2 \equiv x + x \equiv x \ll 1. \quad (18)$$

在等式饱和完成后，系统再根据代价函数抽取最优表达式：

$$best(e) = \min_{n \in e} \left(cost(n) + \sum_{c \in children(n)} best(c) \right). \quad (19)$$

这种方法的优点是避免过早选择某一种变换路径，能够保留更多优化可能性。近年来，egg 等工具使等式饱和在程序优化和领域特定语言优化中变得更加实用^[11]。

2.6 机器学习驱动的编译器优化

在大语言模型被广泛应用之前，机器学习已经被用于编译器优化。其核心动机是：传统编译器中的启发式规则往往依赖人工经验，而不同程序、不同输入和不同硬件平台上的最佳优化决策可能并不相同。

例如，优化阶段排序可以形式化为：

$$\pi^* = \arg \min_{\pi \in \Pi} C(\text{compile}(P, \pi)), \quad (20)$$

其中， π 表示一组优化阶段或编译选项的排列， Π 表示所有可能的优化配置空间。

MILEPOST GCC^[12]是较早将机器学习用于编译器优化的代表系统之一。它通过提取程序特征，学习不同程序在不同优化选项下的表现，从而预测更合适的编译配置。后续研究还将机器学习用于函数内联、循环展开、向量化、寄存器分配和优化阶段排序等任务。

机器学习方法相比手工启发式具有更强的适应性，但也存在限制。首先，它们通常依赖人工设计的程序特征。其次，训练数据收集成本较高。再次，这类方法往往只优化编译器内部的某个决策，而不能像人类开发者一样直接对源代码进行较大范围的算法级改写。

2.7 大语言模型驱动的代码优化

近年来，大语言模型为自动化代码优化提供了新的研究方向。与传统基于规则的方法相比，大语言模型不再局限于固定的语法模式。它们能够理解变量命名、注释、库函数用法、算法结构以及程序的高层意图。因此，大语言模型不仅可以进行局部代码改写，也有可能完成更接近人类开发者的算法级优化。

首先，大语言模型具有更强的上下文理解能力。传统窥孔优化通常只能观察很小的代码窗口，而大语言模型可以同时分析整个函数甚至多个相关函数。这使其有可能发现非局部优化机会。例如，一个低效的双重循环查找可以被改写为基于哈希集合的查找：

```
def count_common(a, b):
    ans = 0
    for x in a:
        for y in b:
            if x == y:
                ans += 1
    return ans
```

可以被优化为：

```
def count_common(a, b):
    b_set = set(b)
    ans = 0
    for x in a:
        if x in b_set:
            ans += 1
    return ans
```

该变换将时间复杂度从接近 $O(nm)$ 降低为接近 $O(n + m)$ 。这种优化并不是简单的局部指令替换，而涉及数据结构选择和算法结构调整。传统编译器通常很难自动完成这类源代码级重构，而大语言模型在这方面具有天然优势。

其次，大语言模型能够利用自然语言中的优化知识。许多性能优化经验并不直接存在于编译器规则中，而是分散在编程教材、性能优化手册、开发者文档、代码审查记录和技

术博客中。例如，“避免在循环中重复分配内存”“将重复线性查找改为哈希查找”“减少不必要的数​​据拷贝”等经验往往以自然语言形式存在。大语言模型经过大规模语料训练，能够在一定程度上吸收并复用这些知识。

再次，大语言模型可以生成可解释的优化结果。传统编译器通常只输出优化后的代码或机器指令，而不会向开发者解释优化依据。相比之下，大语言模型可以同时生成优化后的代码和文字解释，说明为什么该变换可能提升性能，以及需要注意哪些语义条件。这对于面向开发者的代码优化工具具有重要价值。

一般而言，大语言模型驱动的代码优化过程可以表示为：

$$P_{t+1} = \text{LLM}(P_t, I_t, F_t), \quad (21)$$

其中， P_t 表示第 t 轮的程序版本， I_t 表示提示词或优化指令， F_t 表示反馈信息，例如编译错误、单元测试结果、运行时间、性能剖析结果等。

早期直接使用大语言模型进行代码优化的方法通常将该任务视为一次性序列生成问题，即给定原始代码后直接要求模型输出优化代码。然而，这种方式存在明显不足。复杂优化往往需要多步推理和多轮修正，一次性生成很难达到最优；同时，模型可能生成看似合理但实际错误的代码，或者生成通过简单测试但无法覆盖边界情况的代码。

因此，近期研究开始将反馈、搜索和检索机制引入大语言模型优化过程。**LangProp**^[13]是其中一种方法，它将大语言模型生成的代码放入执行环境中，根据输入输出样例、异常信息和性能指标进行评估，再将评估结果反馈给大语言模型，使模型能够迭代改进代码。其优化过程可以表示为：

$$F_t = \text{Eval}(P_t, \mathcal{D}), \quad (22)$$

$$P_{t+1} = \text{LLM}(P_t, F_t), \quad (23)$$

其中， $\mathcal{D}$ 表示测试数据或任务环境， F_t 表示评估反馈。该方法说明，大语言模型不应只被视为一次性代码生成器，而可以被放入一个带有外部反馈的优化循环中。

SBLLM 进一步从搜索的角度建模代码优化问题。它认为，复杂代码优化不应只依赖大语言模型的一次性输出，而应通过搜索不断发现和组合更好的优化方法。**SBLLM** 将大语言模型与进化搜索结合，主要包括三个组成部分：基于执行结果的代表样本选择、自适应优化模式检索，以及受遗传算子启发的链式思维提示。其过程可以抽象为：

$$\mathcal{P}_{t+1} = \text{Select}(\mathcal{P}_t \cup \text{Generate}_{\text{LLM}}(\mathcal{P}_t, R_t)), \quad (24)$$

其中, $\mathcal{P}_t$ 表示第 t 轮候选优化程序集合, R_t 表示检索得到的优化模式。通过这种方式, SBLLM 不再只依赖单个模型输出, 而是维护多个候选程序, 并通过执行反馈选择更有潜力的优化方向。

总体来看, 大语言模型相较于传统规则优化、超优化和传统机器学习优化具有明显优势。它能够直接处理源代码, 能够利用自然语言优化知识, 能够生成较大范围的程序重构, 并且能够通过反馈进行迭代改进。然而, 大语言模型也存在不可忽视的问题。首先, 模型可能产生幻觉, 生成不存在的 API 或错误的程序逻辑。其次, 模型可能错误地应用优化模式, 在不满足前置条件的情况下进行改写。再次, 模型通常缺乏精确的硬件代价模型, 无法可靠判断某个改写在真实平台上是否一定更快。最后, 仅依赖测试无法完全保证语义正确性, 因为测试往往无法覆盖所有输入情况。

因此, 一个更可行的方向是将大语言模型与传统程序分析技术结合。大语言模型负责理解代码和生成灵活改写, 而静态分析、规则匹配、测试和验证机制负责约束模型行为, 降低错误优化风险。在这一背景下, SemOpt 提出了大语言模型与基于规则的静态分析相结合的代码优化框架。

SemOpt 的核心思想并不是直接提示大语言模型“优化这段代码”, 而是先从真实世界优化提交中构建优化策略库, 再生成对应的 Semgrep 静态分析规则, 用这些规则在目标程序中定位可优化代码片段, 最后调用大语言模型根据对应策略生成优化结果。其流程可以概括为:

$$\text{Commit} \rightarrow \text{Strategy} \rightarrow \text{Rule} \rightarrow \text{Location} \rightarrow \text{Optimized Code}. \quad (25)$$

更形式化地说, 一个优化策略可以表示为:

$$S = (\text{Precondition}, \text{Transformation}, \text{Effect}), \quad (26)$$

其中, *Precondition* 表示该策略适用的前置条件, *Transformation* 表示具体代码变换方式, *Effect* 表示预期性能收益。静态分析规则用于判断目标程序中哪些位置满足该策略的前置条件:

$$\text{Match}(P, \text{Rule}_S) = \{l_1, l_2, \dots, l_k\}, \quad (27)$$

其中, l_i 表示程序 P 中被识别出的可优化位置。随后, 大语言模型在策略 S 的指导下

对相应位置进行改写：

$$P' = \text{LLM}(P, l_i, S). \quad (28)$$

与直接使用大语言模型提示相比，SemOpt 的优势在于优化过程更加可控，因为模型不是自由发挥，而是受到显式优化策略和静态分析规则的约束。与传统基于规则的方法相比，SemOpt 又具有更强的灵活性，因为最终代码改写由大语言模型完成，可以适配不同上下文中的代码结构。因此，SemOpt 可以被视为传统编译器优化思想与现代大语言模型代码智能之间的桥梁。

2.8 研究空白与本文动机

综上所述，代码优化方法经历了从人工规则、数据流分析，到搜索式优化、等式饱和、机器学习优化，再到大语言模型优化的发展过程。传统规则方法高效、稳定且可解释，但依赖专家经验，覆盖范围有限。超优化和等式饱和能够探索更大的等价程序空间，但面临搜索空间和可扩展性问题。传统机器学习方法能够改进编译器启发式决策，但通常依赖人工特征，并且难以直接完成源代码级重构。

大语言模型为代码优化带来了新的可能性。它们能够理解代码语义、利用自然语言知识，并生成较大范围的代码改写。然而，单纯依赖大语言模型仍然存在正确性不足、优化不可控和性能收益不稳定等问题。因此，当前关键问题不是简单地证明大语言模型能否优化代码，而是如何使大语言模型驱动和优化过程更加可靠、可控和可复用。本文正是在 SemOpt 的基础上对其进行了扩展，尝试进一步提升大模型结合静态分析的优化效果。

3. SemOpt 方法介绍

SemOpt 是一种结合大语言模型与静态程序分析的自动化代码优化方法。其目标不是直接让大语言模型自由改写代码，而是先为模型提供明确的优化策略和代码定位信息，再由模型生成优化结果。根据原论文，SemOpt 主要由三个部分组成：策略库构建器、规则生成器和代码优化器。^[2]

3.1 整体流程

SemOpt 的整体流程可以概括为“策略获取—规则生成—代码优化”三个阶段。首先，系统从真实代码优化提交中收集优化示例，并利用大语言模型总结其中的优化策略。随后，系统对语义相近的策略进行聚类，形成优化策略库。其次，SemOpt 针对每类优化策略生成对应的 Semgrep 静态分析规则，用于描述该策略适用的代码结构。最后，在处理待优化项目时，系统运行这些 Semgrep 规则，定位潜在可优化代码片段，并将原始代码、匹配位置和优化策略共同输入大语言模型，生成优化后的代码。

这一流程的核心思想是将大语言模型的代码理解与生成能力和静态分析工具的模式匹配能力结合起来。Semgrep 本身支持通过自定义规则描述代码匹配模式，并以 YAML 规则形式组织多个 pattern、message 和 metadata 等信息，因此可以作为策略适用条件的表达工具。

3.2 策略库构建

在第一阶段，SemOpt 从开源项目的历史代码提交中收集优化相关修改。对于每个优化提交，系统保留修改前代码、修改后代码、提交信息和 diff 文件等内容。随后，大语言模型根据这些信息总结该提交所体现的优化策略，例如调整条件判断顺序、减少重复计算、替换低效 API 或改变数据结构使用方式。

为了避免策略库中存在大量重复策略，SemOpt 会进一步对策略描述进行向量化表示，并按照语义相似度进行聚类。聚类后的每一组策略可以被看作一种相对独立的优化模式。这样做可以减少策略库规模，也能降低后续规则生成和代码匹配的开销。SemOpt 原文将这一部分称为 strategy library builder，即策略库构建器。

3.3 静态分析规则生成

在第二阶段，SemOpt 为策略库中的每类优化策略生成 Semgrep 静态分析规则。该规则的作用是描述某一优化策略可以应用到哪些代码结构中。例如，如果某个策略是“在短路求值中优先判断更简单的条件”，那么对应规则就需要匹配具有特定布尔表达式结构的

条件语句。

SemOpt 使用大语言模型辅助生成这些 **Semgrep** 规则。生成过程并不是一次性完成的，而是包含规则生成、执行验证和反馈修改等步骤。如果生成的规则存在语法错误或无法正确运行，系统会将错误信息反馈给大语言模型，并要求其修正规则。通过这一过程，**SemOpt** 将自然语言形式的优化策略转化为可以自动扫描代码的静态分析规则。

3.4 代码优化

在第三阶段，**SemOpt** 将生成好的 **Semgrep** 规则应用到待优化项目中。系统运行规则后，可以得到潜在可优化代码片段的位置以及对应的优化策略。随后，**SemOpt** 构造提示词，将完整代码、代码片段位置和优化策略一同输入大语言模型，要求模型在保持原有语义的前提下生成优化后的代码。

与直接提示大语言模型优化代码相比，**SemOpt** 的优化过程更加明确。模型不需要在整段代码中盲目寻找优化点，也不需要自行判断应该采用哪种优化方式。静态分析规则已经提前给出了候选位置和策略信息，模型主要负责根据上下文完成具体代码改写。因此，**SemOpt** 能够在一定程度上降低大语言模型生成结果的不稳定性。

3.5 方法特点

SemOpt 的主要特点是将“策略选择”和“代码生成”分开处理。传统 RAG 方法通常依赖文本相似度检索历史优化示例，但语法相似的代码不一定适用相同优化策略，语法不同的代码也可能共享同一优化思想。**SemOpt** 则通过静态分析规则刻画策略的适用条件，从而更准确地发现目标代码中的优化机会。原论文实验显示，**SemOpt** 在 151 个代码优化任务上，相比 Direct Prompting、RAG 和 RAPGen 等基线方法取得了更高的成功优化数量；在真实大型 C/C++ 项目中，也能带来一定性能提升。

总体来看，**SemOpt** 提供了一种较清晰的自动化代码优化流程：先从已有优化经验中获得策略，再将策略转化为静态分析规则，最后用规则定位代码并引导大语言模型完成优化。这一流程为本文后续从书籍和技术文档中提取优化策略提供了基础。本文的扩展工作正是在 **SemOpt** 原有 commit-based 策略获取方式之外，引入 document-based 策略提取机制，使优化策略库能够吸收更广泛的软件工程知识。

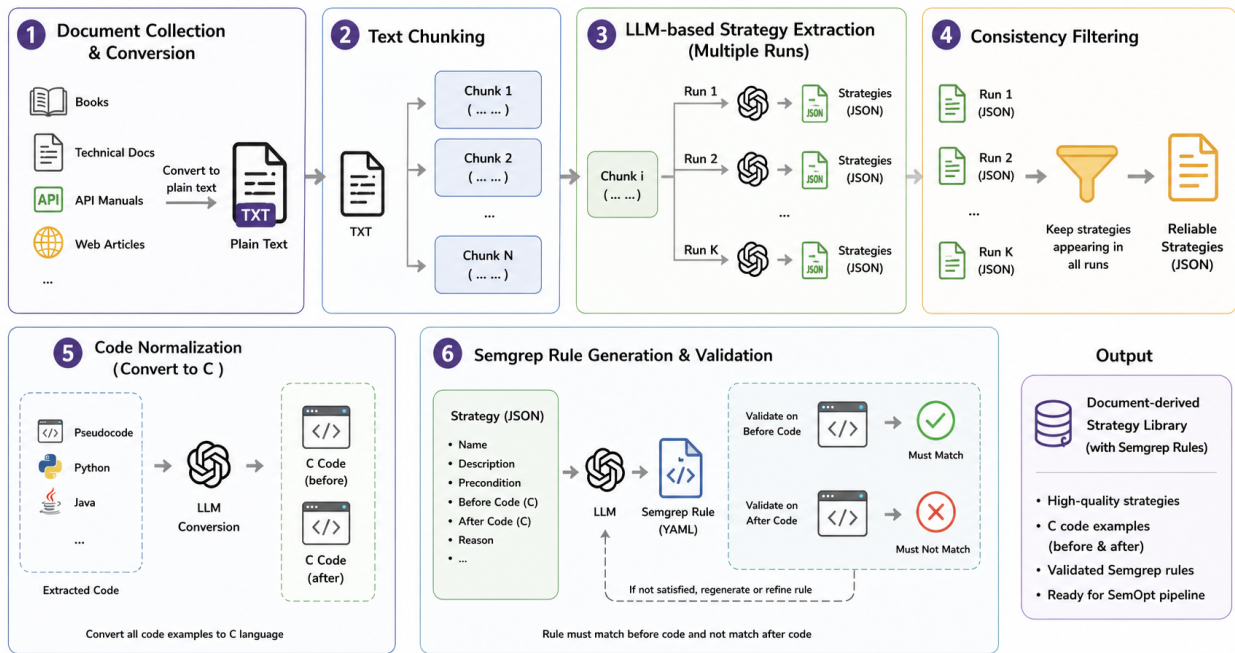


图 4.1 面向文档的优化策略提取流程

4. 面向文档的优化策略提取方法

原始 SemOpt 主要从 GitHub 等开源项目的优化提交中提取优化策略，再将策略转化为 Semgrep 静态分析规则，并用于后续代码优化。该方法能够从真实开发实践中学习优化经验，但策略来源仍然依赖历史代码提交。实际上，许多具有价值的优化知识并不一定以 commit 的形式出现，而是存在于编程书籍、官方文档、API 使用说明和性能优化手册中。为扩展 SemOpt 的策略来源，本文设计了一种面向文档的优化策略提取方法，使系统能够从文本资料中抽取优化策略，并将其接入 SemOpt 原有的规则生成和代码优化流程。

4.1 整体流程

本文方法的整体流程如图 4.1 所示，主要包括文档预处理、文本分块、策略抽取、策略筛选、代码规范化和 Semgrep 规则生成六个步骤。首先，将原始书籍或技术文档转换为纯文本格式；随后根据文档结构和文本长度将其划分为若干文本块；接着将每个文本块输入大语言模型，要求模型从中抽取代码优化策略，并按照统一 JSON 格式输出；然后对同一文本块进行多次策略抽取，只保留多次生成中反复出现的策略，以降低大语言模型幻觉的影响；之后，将策略中的伪代码或非 C 语言代码统一转换为 C 语言代码；最后，基于抽取得到的策略生成对应的 Semgrep 规则，并通过优化前后代码对规则进行验证。

4.2 文档预处理与文本分块

由于不同资料的文件格式和组织方式不同，本文首先将原始资料统一转换为文本格式。对于 EPUB 格式的书籍，文本通常具有较清晰的章节结构，因此可以按照章节或小节进行切分。对于 PDF 格式的文档，由于其文本结构可能不稳定，本文主要根据文本长度、段落边界和语义完整性进行切块，尽量避免在一句话或一个代码示例中间截断。

文本分块的目的是控制每次输入大语言模型的上下文长度，同时保留足够的语义信息。过长的文本块可能导致模型忽略细节，过短的文本块则可能缺少策略所需的上下文。因此，本文在切分时优先保持段落和代码示例的完整性。如果某个文本块末尾存在明显的语义截断，例如策略描述尚未结束或代码示例不完整，则在提示词中允许大语言模型判断是否需要后续文本块的信息，再决定是否进行策略抽取。

4.3 基于大语言模型的策略抽取

完成文本分块后，本文将每个文本块输入大语言模型，要求模型识别其中与代码性能优化相关的内容，并输出结构化策略。与从 commit 中提取策略不同，文档中的优化知识通常以自然语言形式出现，可能包含原理解释、适用场景、代码示例和注意事项。因此，本文要求模型不仅提取策略名称，还需要提取策略描述、适用条件、优化前代码、优化后代码和性能收益原因等信息。

本文采用 JSON 作为策略存储格式。每条策略包含如下字段：

- **name**: 优化策略名称;
- **description**: 策略的自然语言描述;
- **precondition**: 策略适用条件;
- **before_code**: 优化前代码或代码模式;
- **after_code**: 优化后代码或代码模式;
- **optimization_summary**: 优化策略总结;
- **doc**: 来源文档;

该结构化格式能够将文档中的自然语言知识转化为后续程序可以处理的策略数据。为了使大语言模型能够从文档片段中稳定提取结构化优化策略，本文设计了统一的策略抽取 Prompt。该 Prompt 要求模型仅提取具有源代码级示例的优化策略，并以 JSON 数组形式输出策略标题、描述、类别、适用条件、优化前后代码以及优化要点。通过固定输出格式，可以降低后续解析成本，并便于将抽取得到的策略接入 SemOpt 的规则生成流程。具体 Prompt 如 Listing 1 所示（来源文档无需通过大模型，由代码自动添加）。

You are a professional code optimization expert. Please extract all optimization strategies from the following document content.

For each strategy, extract the following information and return it in JSON format:

1. title: Strategy title
2. description: Detailed description of the strategy
3. category: Strategy category (e.g., CPU optimization, memory optimization, I/O optimization, etc.)
4. precondition: Conditions must be satisfied before applying this strategy.
5. code_before: List of code examples before optimization, each containing language and code
6. code_after: List of code examples after optimization, each containing language and code
8. optimization_summary: One-sentence summary of the optimization

Requirements:

- The optimization strategy must take effect at the source code level.
- Preserve the sample code provided in the document without modification.
- Only extract the strategy in which the sample code is given in the document.
- If the strategy involves assembly code, ignore it unless the example uses inline assembly within C code with `__asm__` blocks.
- The document must provide source-code-level example code for each strategy.
- The strategy should be able to apply to C code.

Return only a JSON array in the following format:

```
[
  {
    "title": "Strategy title",
    "description": "Strategy description",
    "category": "Category",
    "precondition": "precondition",
```

```
"code_before": [  
    {"language": "c", "code": "// code before optimization"}  
],  
"code_after": [  
    {"language": "c", "code": "// code after optimization"}  
],  
"optimization_summary_final": "One sentence summary"  
}  
]  
  
Document content:  
{text}
```

Listing 1: 用于文档优化策略抽取的 Prompt

4.4 多次生成与一致性筛选

由于大语言模型在信息抽取过程中可能产生幻觉，本文对同一文本块进行多次独立策略抽取。每次抽取使用相同文本内容和相同输出格式要求，但保留模型生成过程中的一定差异。随后，系统对多次抽取得到的策略进行对齐和筛选，只保留在多次结果中反复出现的策略。

具体而言，本文根据策略名称、策略描述、适用条件和代码模式判断两条策略是否表示同一优化思想。如果某条策略只在一次生成结果中出现，而其他生成结果中没有对应内容，则认为该策略可信度较低，并将其过滤。相反，如果某条策略在多次生成中均被提取出来，则说明该策略更可能来自原始文档，而不是模型临时生成的无依据内容。通过这种一致性筛选，可以在一定程度上降低大语言模型幻觉对策略库质量的影响。

4.5 代码示例规范化

从书籍和文档中提取的代码示例并不一定能够直接用于 SemOpt。部分资料中的代码可能是伪代码，也可能使用 Python、Java 或其他语言表示。由于本文后续规则生成和代码优化主要面向 C 语言代码，因此需要对这些代码示例进行规范化处理。

对于伪代码，本文使用大语言模型将其转换为语义等价的 C 语言代码；对于其他编程语言的代码，则要求模型保留原有优化思想，并生成对应的 C 语言版本。在转换过程中，模型需要同时输出优化前和优化后的 C 代码，以便后续生成 Semgrep 规则。该步骤并不要求代码一定能够作为完整程序独立运行，但需要尽量保持语法合理、变量关系清晰，并能

够体现优化前后代码结构的差异。具体 Prompt 如 Listing 2 所示。

```
You are a professional code conversion expert. Please convert the
    following SOURCE_LANG_PLACEHOLDER code into equivalent C language code.

Requirements:
1. Keep the logic and functionality of the original code unchanged
2. Use standard C language (C99 or later)
3. Keep the code style clear and readable
4. If the original code uses language-specific features, implement
    equivalent functionality in C

Original SOURCE_LANG_PLACEHOLDER code:
```SOURCE_LANG_PLACEHOLDER
CODE_PLACEHOLDER
```

Please return only the C language code, without any explanation:
```

Listing 2: 用于代码转换的 Prompt

例如，如果文档中给出的策略是“避免在循环中重复计算不变表达式”，即使原文使用伪代码描述，系统也会将其转换为 C 语言中的循环代码形式，使优化前代码包含循环内重复计算，优化后代码将不变表达式移动到循环外。这样生成的代码示例可以作为后续 Semgrep 规则生成和验证的输入。

4.6 Semgrep 规则生成与验证

在得到结构化策略和 C 语言代码示例后，本文进一步生成对应的 Semgrep 静态分析规则。Semgrep 规则的作用是检测目标代码中是否存在与优化前代码相似的低效模式。与普通模式匹配不同，本文要求生成的规则同时满足两个条件：第一，规则应能够检测出优化前代码；第二，规则不应检测出优化后的代码。这样可以避免规则过于宽泛，导致优化后的代码仍然被误判为需要优化。整体流程如图 4.2 所示。

规则生成过程同样由大语言模型完成。模型根据策略描述、适用条件、优化前代码和优化后代码生成初始 Semgrep 规则。随后，系统分别在优化前代码和优化后代码上运行该规则。如果规则无法检测出优化前代码，则说明规则过于严格或模式错误，需要重新生成或修改；如果规则仍然检测出优化后的代码，则说明规则不能区分低效模式和优化后模式，也需要进一步修正。只有同时满足“匹配优化前代码”和“不匹配优化后代码”的规则，才会被保留。同时，如果生成的规则无法编译或者无法正确匹配到代码，会尝试将错误信

息告知大模型，让其尝试根据错误信息再次生成 **Semgrep** 规则。多次重复上述流程直到生成的规则符合要求，或者尝试足够多次数后仍无法生成出对应规则就将此策略舍弃。这部分 prompt 如 Listing 3, 4 所示。

```
You are a professional semgrep rule generation expert. Please generate a
    semgrep detection rule based on the following optimization strategy.
```

```
Strategy Information:
```

- Title: TITLE_PLACEHOLDER
- Description: DESCRIPTION_PLACEHOLDER
- Optimization Points: SUMMARY_PLACEHOLDER

```
code to detect (the "before" code pattern):
```

```
```c
```

```
CODE_BEFORE_PLACEHOLDER
```

```
```
```

```
optimized code (the "after" code pattern):
```

```
```c
```

```
CODE_AFTER_PLACEHOLDER
```

```
```
```

```
Please generate a semgrep rule to detect locations in the code where the "
    before" pattern is used.
```

```
Rule Requirements:
```

1. Use YAML format
2. Must include id, languages, message, severity, pattern fields
3. pattern should be generic enough to match similar code patterns
4. Can use semgrep's metavariable syntax like \$VAR, \$FUNC etc.
5. Can use ... to match arbitrary code
6. languages must be ["c"] or ["cpp"]
7. the rule should not detect the "after" code pattern (to avoid false positives)

```
Please return the YAML format rule:
```

```
```yaml
```

```
rules:
 - id: opt_strategy_X
 languages: [c]
 message: "Detection of optimizable code patterns"
 severity: WARNING
 pattern: |
 // Your matching pattern
 ...
```

Listing 3: 用于 Semgrep 生成的 Prompt

You are a professional semgrep rule generation expert. The previously generated rule has errors. Please fix it based on the error messages.

Strategy Information:

- Title: TITLE\_PLACEHOLDER
- Description: DESCRIPTION\_PLACEHOLDER
- Optimization Points: SUMMARY\_PLACEHOLDER

code to detect (the "before" code pattern):

```
```c
CODE_BEFORE_PLACEHOLDER
```
```

optimized code (the "after" code pattern):

```
```c
CODE_AFTER_PLACEHOLDER
```
```

previous rule:

```
```yaml
PREVIOUS_RULE_PLACEHOLDER
```
```

error messages:

ERROR\_MESSAGES\_PLACEHOLDER

Please fix the above errors and regenerate the correct semgrep rule.

Note:

1. Carefully check the YAML syntax (indentation, colons, hyphens, etc.)
2. Ensure the pattern is a valid semgrep pattern that can match the "before" code and does not match the "after" code.
3. Must return a complete YAML format rule.

Please return the fixed YAML:

Listing 4: 带错误反馈的 Semgrep 生成的 Prompt

这一验证过程提高了规则的可用性。由于文档来源策略通常没有真实 commit 中的完整上下文，直接生成规则容易出现匹配范围过宽或过窄的问题。通过优化前后代码对规则进行双向验证，可以使生成的 Semgrep 规则更接近策略本身的适用条件，也能减少后续代码优化阶段的误报。

## 4.7 与原始 SemOpt 的关系

本文方法并不改变 SemOpt 的整体优化框架，而是扩展其策略获取来源。原始 SemOpt 的策略主要来自真实代码优化提交，属于 commit-based strategy mining；本文则引入 document-based strategy extraction，使系统能够从书籍和技术文档中获取优化策略。两类策略来源具有互补关系：代码提交能够反映真实项目中的优化实践，而书籍和文档通常包含更系统化、更明确的优化知识。将文档策略接入 SemOpt 后，系统可以利用更丰富的知识来源构建策略库，并继续使用原有的 Semgrep 规则生成和大语言模型优化流程。

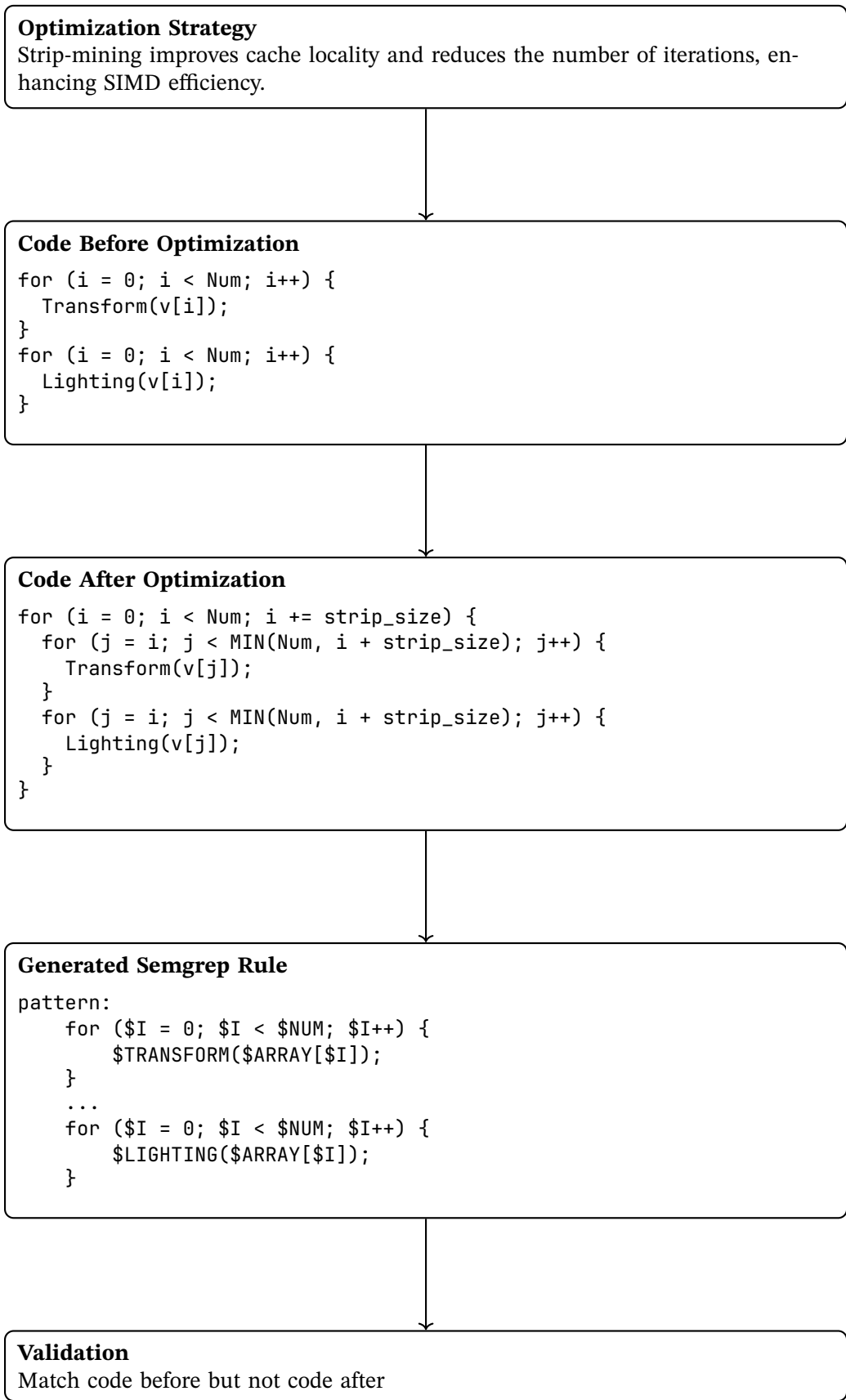


图 4.2 从优化策略到生成 Semgrep 规则的流程示意图

## 5. 实验与结果对比

为验证面向文档的优化策略提取方法在实际项目中的优化效果，本节在 SemOpt 原有框架基础上，将文档来源策略补充到策略库中，并在两个真实 C/C++ 项目上进行了性能优化实验。

### 5.1 实验设置

#### 5.1.1 实验环境与基准

实验在两个广泛使用的开源 C/C++ 项目上进行：Redis（版本: v7.0.15）与 RocksDB（版本: v10.2.1）。两者均提供了较为完善的性能测试套件，可以定量衡量优化前后的性能变化。

测试的策略库来自于 Intel® 64 and IA-32 Architectures Optimization Reference Manual<sup>[14]</sup>和《软件系统优化》<sup>[15]</sup>。

对于每个项目，首先克隆其最新稳定版本，并从项目提供的性能测试套件中提取所有可运行的性能测试用例。Redis 项目包含 21 个性能测试用例，RocksDB 项目包含 2 个性能测试用例，每个用例通常对应于项目中的一项核心功能或典型工作负载。

#### 5.1.2 评估指标

与 SemOpt 原论文一致，本文采用以下评估指标：

**性能提升比例 (Perf Improvement)**：对于每个性能测试用例，分别在优化前版本和优化后版本上各运行 6 次，丢弃第一次运行结果以避免冷启动影响，取剩余 5 次结果的平均值。根据测试用例的报告方式，将性能提升统一为速度提升比例：

- 若测试报告数值越大表示性能越好（例如每秒处理的操作数），则提升比例为  $\frac{y-x}{x} \times 100\%$ ；
- 若测试报告数值越小表示性能越好（例如执行耗时），则提升比例为  $\frac{x-y}{y} \times 100\%$ ，

其中  $x$  为优化前性能结果， $y$  为优化后性能结果。

**最大提升 (Max Improvement)**：所有性能测试用例中的最大提升比例。**平均提升 (Avg Improvement)**：所有性能测试用例提升比例的算术平均。**显著提升用例数 (#Test Cases where  $\uparrow \geq n\%$ )**：提升比例超过 5% 和 10% 的测试用例数量。

#### 5.1.3 优化过程

实验沿用了 SemOpt 原论文的优化流程，并将文档来源策略纳入策略库。

首先，通过性能剖析工具 `perf` 和覆盖率工具 `gcov` 对每个性能测试用例的执行过程进行分析，识别其中执行时间占比超过 0.1% 的热点函数。这些函数构成后续优化的候选目标。

其次，将 SemOpt 原有的 `commit-based` 策略库与本文从书籍和技术文档中抽取的优化策略合并，生成统一的 Semgrep 静态分析规则集合。

然后，对每个热点函数，运行所有 Semgrep 规则以定位潜在可优化代码位置。对于每个被匹配的位置，将原函数代码、位置信息和对应的优化策略共同输入大语言模型（实验中使用 DeepSeek-V3），生成优化后的代码。

最后，对生成的优化结果进行评估。对所有热点函数分别选择最优有效优化变体，组合生成最终优化版本的项目。

## 5.2 实验结果

两个项目的性能测试结果汇总于表 5.1。

表 5.1 面向文档策略扩展后的真实项目优化结果

| 项目      | Max Improvement | Avg Improvement | #Cases $\uparrow \geq 5\%$ | #Cases $\uparrow \geq 10\%$ |
|---------|-----------------|-----------------|----------------------------|-----------------------------|
| Redis   | 29.00%          | 6.75%           | 8/21                       | 4/21                        |
| RocksDB | 7.19%           | 5.84%           | 1/2                        | 0/2                         |

### 5.2.1 Redis 优化结果

在 Redis 上，优化后版本在 21 个性能测试用例中取得了最高 29.00% 的性能提升，平均提升为 6.75%。其中，有 8 个测试用例的优化提升超过 5%，4 个测试用例的优化提升超过 10%。这些提升主要来自循环不变式外提、条件短路求值顺序调整、无分支化改写以及容器操作替换等优化策略。部分策略直接来源于本文从编程书籍和性能优化手册中抽取的文档知识，表明文档来源策略能够在实际项目中产生可观测的性能收益。

与 SemOpt 原论文中基于单一 `commit-based` 策略库在 Redis 上取得的结果（最大提升 6.14%，平均提升 1.68%）相比，本文扩展策略库后的优化效果有明显提高，最大提升从 6.14% 提升至 29.00%，平均提升从 1.68% 提升至 6.75%。这在一定程度上说明了引入文档来源策略对优化效果的增强作用。<sup>[2]</sup>

### 5.2.2 RocksDB 优化结果

在 RocksDB 上，优化后版本在 2 个性能测试用例中取得了最高 7.19% 的性能提升，平均提升为 5.84%。其中，有 1 个测试用例的优化提升超过 5%，没有测试用例的优化提升超过 10%。

与 SemOpt 原论文中 RocksDB 的结果（最大提升 5.04%，平均提升 2.41%）相比，本文扩展后的优化效果也有一定提升，最大提升从 5.04% 提高到 7.19%，平均提升从 2.41% 提高到 5.84%。<sup>[2]</sup>

### 5.2.3 结果分析

两组实验结果共同表明，将文档来源优化策略引入 SemOpt 策略库后，系统可以捕获更多样化的优化机会，并在两个不同领域、不同规模的真实项目上产生可测量的性能改进。相比原论文中仅依赖 commit-based 策略库的 SemOpt 版本，扩展后的系统在最大提升和平均提升两个维度上均有改善。

此外，文档来源策略的可解释性较高，便于对策略的适用条件进行人工审查和修正，这也是在实际工程中应用此类策略的优势之一。

综上所述，本实验验证了面向文档的优化策略提取方法能够切实增强自动化代码优化系统的优化能力，并与原有 commit-based 策略形成有效互补。

## 6. 相关工作

代码优化是软件工程和编译器领域的经典研究课题。本文的工作涉及基于大语言模型的代码优化、基于规则的程序变换以及从非代码来源获取优化知识三个方面。

### 6.1 基于大语言模型的代码优化

近年来，大语言模型在代码理解和代码生成任务上展现了强大能力，也逐步被应用于自动化代码优化。Shypula 等人<sup>[16]</sup> 构建了 PIE 基准用于评估代码优化方法，并尝试了检索增强生成（RAG）等多种提示技术。Garg 等人提出的 RAPGen<sup>[1]</sup> 聚焦于 API 替换类的性能优化，通过检索历史优化提交来指导模型生成更高效的代码。Gao 等人提出的 SBLLM<sup>[17]</sup> 将搜索与大语言模型结合，通过执行反馈和进化搜索发现更优的优化组合。EffiCoder<sup>[18]</sup> 和 PerfCodeGen<sup>[19]</sup> 则利用性能剖析数据和执行反馈来引导大语言模型进行迭代优化。

上述方法主要从代码或执行反馈中学习优化经验，而本文关注的是从书籍、技术文档等自然语言资料中提取优化策略。这些文档资源通常包含更系统化、更具可解释性的优化知识，是对代码提交和执行反馈的重要补充。与此同时，本文的方法可以与上述基于执行反馈或搜索的优化方法相结合，进一步丰富策略来源。

### 6.2 基于静态分析的程序优化

静态程序分析在编译器优化中已经得到广泛应用。经典的数据流分析框架<sup>[4]</sup> 为公共子表达式消除、常量传播、死代码消除等优化提供了理论基础。基于规则的优化方法在编译器中被长期使用，如窥孔优化<sup>[20]</sup> 和代数化简等，这些方法通过人工编写的变换规则将低效代码模式替换为高效等价形式。

Semgrep<sup>[21]</sup> 是一个可定制的静态分析工具，允许用户以 YAML 格式编写模式匹配规则，已被广泛用于代码质量检查和安全漏洞检测。SemOpt 将 Semgrep 与大语言模型结合，利用大语言模型自动生成分析规则，再用这些规则定位可优化代码，实现了从“人工编写规则”到“自动生成规则”的转变。本文在此基础上进一步扩展策略来源，使系统能够从文档中获取优化知识并生成相应的静态分析规则，提高了规则生成的知识覆盖率。

## 7. 结论

本文在 **SemOpt** 原有框架的基础上，提出了一种面向书籍和技术文档的优化策略提取方法，扩展了自动化代码优化系统的策略来源。该方法利用大语言模型从自然语言文档中识别与代码性能优化相关的内容，抽取结构化优化策略，并通过一致性筛选和代码规范化将其转化为可供 **Semgrep** 静态分析规则生成使用的格式。实验结果表明，将文档来源策略补充到策略库后，在 **Redis** 和 **RocksDB** 两个真实 C/C++ 项目上，优化后的版本均取得了可观测的性能提升，且相比原论文中仅依赖 **commit-based** 策略库的结果有明显改善。这验证了本文方法在实际项目中的有效性，也表明从文档中系统提取优化知识是增强自动化代码优化系统的可行途径。

本文的工作为构建多源知识驱动的代码优化系统提供了基础。未来可以进一步探索将更多类型的资料纳入策略来源，并研究不同来源策略的质量评估与自动融合机制。

## 参考文献

- [1] GARG S, MOGHADDAM R Z, SUNDARESAN N. RAPGen: An Approach for Fixing Code Inefficiencies in Zero-Shot[C]//Proceedings of the 47th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP). IEEE, 2025.
- [2] ZHAO Y, XIAO Y, XIAO Q, et al. SemOpt: LLM-Driven Code Optimization via Rule-Based Analysis[J]. arXiv preprint arXiv:2510.16384, 2025.
- [3] ALLEN F E, COCKE J. A Catalogue of Optimizing Transformations[C]//RUSTIN R. Design and Optimization of Compilers. Prentice-Hall, 1972: 1-30.
- [4] KILDALL G A. A Unified Approach to Global Program Optimization[C/OL]//Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. ACM, 1973: 194-206. DOI: 10.1145/512927.512945.
- [5] WOLF M E, LAM M S. A Data Locality Optimizing Algorithm[C/OL]//Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation (PLDI). ACM, 1991: 30-44. DOI: 10.1145/113445.113449.
- [6] MCKINLEY K S, CARR S, TSENG C W. Improving Data Locality with Loop Transformations[J/OL]. ACM Transactions on Programming Languages and Systems, 1996, 18(4): 424-453. DOI: 10.1145/233561.233564.
- [7] FEAUTRIER P. Dataflow Analysis of Array and Scalar References[J/OL]. International Journal of Parallel Programming, 1991, 20(1): 23-53. DOI: 10.1007/BF01407931.
- [8] FEAUTRIER P. Some Efficient Solutions to the Affine Scheduling Problem, Part I: One-Dimensional Time[J/OL]. International Journal of Parallel Programming, 1992, 21(5): 313-348. DOI: 10.1007/BF01379407.
- [9] MASSALIN H. Superoptimizer: A Look at the Smallest Program[C/OL]//Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). ACM, 1987: 122-126. DOI: 10.1145/36206.36194.
- [10] SCHKUFZA E, SHARMA R, AIKEN A. Stochastic Superoptimization[C/OL]//Proceedings of the 18th International Conference on Architectural Support for Pro-

- gramming Languages and Operating Systems (ASPLOS). ACM, 2013: 305-316. DOI: 10.1145/2451116.2451150.
- [11] WILLSEY M, NANDI C, WANG Y R, et al. egg: Fast and Extensible Equality Saturation [C/OL]// Proceedings of the ACM on Programming Languages: vol. 5: POPL. 2021: 1-29. DOI: 10.1145/3434304.
- [12] FURSIN G, KASHNIKOV Y, MEMON A W, et al. Milepost GCC: Machine Learning Enabled Self-tuning Compiler[J/OL]. International Journal of Parallel Programming, 2011, 39(3): 296-327. DOI: 10.1007/s10766-010-0161-2.
- [13] FEDOSEEV G, CORRADO G, et al. LangProp: A Code Optimization Framework Using Large Language Models Applied to Driving[J]. arXiv preprint arXiv:2401.10314, 2024.
- [14] Intel® 64 and IA-32 Architectures Optimization Reference Manual: 355989-050US [A/OL]. Intel Corporation. 2025. <https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html>.
- [15] 郭健美, 黄波, 刘通宇, 等. 软件系统优化[M]. 北京: 机械工业出版社, 2025.
- [16] SHYPULA A, MADAAN A, ZENG Y, et al. Learning Performance-Improving Code Edits[C]// Proceedings of the 12th International Conference on Learning Representations (ICLR). 2024.
- [17] GAO S, GAO C, GU W, et al. Search-Based LLMs for Code Optimization[C]// Proceedings of the 47th IEEE/ACM International Conference on Software Engineering (ICSE). IEEE, 2025: 578-590.
- [18] HUANG D, ZENG G, DAI J, et al. SWIFTCODER: Enhancing Code Generation in Large Language Models through Efficiency-Aware Fine-tuning[C]// Proceedings of Machine Learning Research: Proceedings of the 42nd International Conference on Machine Learning (ICML): vol. 267. PMLR, 2025: 26058-26076.
- [19] PENG Y, GOTMARE A D, LYU M R, et al. PerfCodeGen: Improving Performance of LLM Generated Code with Execution Feedback[C]// Proceedings of the 2nd International Conference on AI Foundation Models and Software Engineering (FORGE). IEEE, 2025: 1-13.
- [20] MCKEEMAN W M. Peephole Optimization[J/OL]. Communications of the ACM, 1965, 8(7): 443-444. DOI: 10.1145/364995.365000.

- [21] Semgrep Inc. Semgrep[EB/OL]. 2020. <https://github.com/semgrep/semgrep>.

## 致谢

# 北京大学学位论文原创性声明和使用授权说明

## 原创性声明

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不含任何其他个人或集体已经发表或撰写过的作品或成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本声明的法律结果由本人承担。

论文作者签名：

日期： 年 月 日

## 学位论文使用授权说明

本人完全了解北京大学关于收集、保存、使用学位论文的规定，即：

- 按照学校要求提交学位论文的印刷本和电子版本；
- 学校有权保存学位论文的印刷本和电子版，并提供目录检索与阅览服务，在校园网上提供服务；
- 学校可以采用影印、缩印、数字化或其它复制手段保存论文；

论文作者签名：

导师签名：

日期： 年 月 日