软件学报 ISSN 1000-9825, CODEN RUXUEW
[doi: 10.13328/j.cnki.jos.007644] [CSTR: 32375.14.jos.007644]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

GCC 编译优化选项关系的分析应用*

高秉玉[1,2], 姚梦雨[1,2], 王子铭[1,2], 李 锭[1,2], 陈向群[1,2], 郭 耀[1,2]

[1](高可信软件技术教育部重点实验室 (北京大学), 北京 100871)

[2](北京大学 计算机学院 软件研究所, 北京 100871)

通信作者: 郭耀, E-mail: yaoguo@pku.edu.cn

摘 要: 编译优化选项作为编译器与开发者之间的核心交互接口, 其合理配置直接影响软件质量与开发效率. 然而, 编译选项的数量庞大且关系复杂, 优化效果同时受到程序特征和机器架构的动态影响, 官方文档难以满足实际应用场景的配置指导需求. 针对上述挑战, 结合静态和动态分析技术, 基于 GCC 编译器源码深入探讨影响编译器选项效果的两类关系: 内部约束关系和外部环境依赖关系. 首先基于静态分析识别出选项间 68 对内部约束关系 (涉及 84 个选项) 以及 103 对外部环境依赖关系 (包括 51 个机器架构敏感选项与 52 个程序特征敏感选项), 并进一步基于动态分析技术验证选项在不同外部环境下的有效性. 然后, 将选项关系的分析结果应用于 3 个编译相关技术场景: 1) 提高选项自动调优效率, 设计基于选项关系感知的遗传调优算法, 相较于不考虑选项关系的其他算法 (如遗传算法), 性能平均提升 3.65%; 2) 提高编译器测试效率, 设计基于选项关系的测试序列生成方法, 生成效率相较于随机方法提升了 51.7%; 3) 为开发者提供更有效的选项配置方案, 帮助其合理使用选项以达到预期优化目标. 通过这 3 个应用, 表明编译选项关系在自动调优、编译器测试及实际配置中具有关键作用.

关键词: 编译优化; 编译选项; 编译自动调优; 编译器测试

中图法分类号: TP314

中文引用格式: 高秉玉, 姚梦雨, 王子铭, 李锭, 陈向群, 郭耀. GCC编译优化选项关系的分析应用. 软件学报. http://www.jos.org.cn/1000-9825/7644.htm

英文引用格式: Gao BY, Yao MY, Wang ZM, Li D, Chen XQ, Guo Y. Analysis and Application of Relationships Among GCC Compiler Optimization Options. Ruan Jian Xue Bao/Journal of Software (in Chinese). http://www.jos.org.cn/1000-9825/7644.htm

Analysis and Application of Relationships Among GCC Compiler Optimization Options

GAO Bing-Yu[1,2], YAO Meng-Yu[1,2], WANG Zi -Ming[1,2], LI Ding[1,2], CHEN Xiang-Qun[1,2], GUO Yao[1,2]

[1](Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, Beijing 100871, China)

[2](Software Engineering Institute, School of Computer Science, Peking University, Beijing 100871, China)

Abstract: Compiler optimization options serve as a core interface between the compiler and developers, and their proper configuration directly impacts software quality and development efficiency. However, there are a large number of options with complex interrelations, and their optimization effects are dynamically influenced by program features and target machines. The official documentation is also insufficient to provide the configuration guidance needed for practical application scenarios. To address these challenges, this study combines static and dynamic analysis techniques applied to the GCC source code and investigates two types of relationships that affect the effectiveness of compiler options: internal constraint relationships and external environment dependencies. First, 68 pairs of internal constraint relationships (involving 84 options) and 103 pairs of external environment dependencies (including 51 architecture-sensitive options and 52 program feature-sensitive options) are identified through static analysis, and the effectiveness of these options in different external environments is further validated through dynamic analysis. Finally, the analysis results of option relationships are applied to three compiler-related scenarios: 1) improving the effectiveness of compiler auto-tuning by designing a relationship-aware genetic auto-tuning

* 基金项目: 国家重点研发计划 (2022YFB4501802); 北京市自然科学基金-小米创新联合基金 (L243010); 酱酒行业数字化转型联合实验室
收稿时间: 2025-03-10; 修改时间: 2025-10-14, 2025-12-22; 采用时间: 2026-01-30; jos 在线出版时间: 2026-05-13

algorithm, which achieves an average performance improvement of 3.65% compared with algorithms that do not consider option relationships, such as the genetic algorithm; 2) improving the effectiveness of compiler testing by proposing a option sequence test case generation method based on option relationships, achieving a 51.7% increase in generation efficiency compared to random methods; 3) providing developers with more effective option configuration schemes to help them use options properly to achieve their desired optimization goals. These three applications demonstrate the key role of compiler option relationships in compiler auto-tuning, compiler testing, and practical configuration.

Key words: compiler optimization; compiler option; compiler auto-tuning; compiler testing

现代编译器[1]通过提供上百个优化选项, 使开发者能够对代码编译过程进行细粒度控制. 然而, 如何合理使用这些选项以实现特定的优化需求仍面临较大挑战. 首先, 选项之间存在复杂的依赖关系[2] (如启用/禁用逻辑、优先级约束), 开发者往往难以完全掌握; 其次, 选项优化效果受多种外部因素影响, 既受程序特征[3] (如数据结构、控制流数据流模式) 制约, 也与机器架构[4] (如指令集特性、缓存层次) 密切相关. 当外部条件不满足选项的应用前提, 则优化策略无法生效, 难以达到理想的优化效果. 尽管编译器提供了标准优化序列 (例如-O1/O2/O3/Ofast 等) 以满足不同程度的性能优化需求, 但通用的优化配置无法在所有场景下都保证满足开发者的具体需求[5]. 因此, 在实际应用中, 开发者仍需手动配置编译选项以达到优化目标. 然而, 编译器文档提供选项的具体信息有限, 仅凭字面含义直接使用难以获得理想的效果.

编译选项的复杂性首先体现在其内部约束关系[2]. 以内联优化为例, GCC 共提供了 8 个内联相关的优化选项, 分别从不同维度进行内联优化 (如部分内联, 间接内联等). 此外, 编译选项的效果还受到程序特征和机器架构等外部因素的影响. 例如, -fdevirtualize 选项针对 C++程序优化, 通过将虚函数调用转换为直接调用提高程序性能; -fauto-inc-dec 依赖 ARM 架构优化自增和自减操作, 以减少指令数量并提高缓存利用率. 如果程序特征或机器架构无法满足选项的前提条件, 其优化作用将无法生效.

系统地理解选项之间的关系不仅能够帮助开发者正确、高效地使用编译选项, 也能推动编译器相关领域技术的发展. 在编译选项自动调优领域[6,7], 由于选项组合空间过于庞大, 算法需要探索数百个选项在不同状态下的组合, 以获得程序的最优选项序列. 现有的自动调优算法 (如 BOCA[8]、SRTuner[9]等) 局限于将编译器视为黑盒, 仅根据调整选项状态后程序性能的变化, 使用机器学习等方法逐步调整搜索方向, 忽略了编译器及选项本身包含的信息, 因而限制了搜索效率. CFSCA[3]、PDCAT[2]等研究尝试结合编译器文档提供的信息, 引入选项间的关联信息以及与程序特征相关的启发式知识, 在一定程度上减小搜索空间. 但编译器文档中关于选项关系的描述通常较为粗略且不完整, 难以准确反映编译器内部的真实逻辑. 若能在搜索过程中基于编译器源码分析获得更全面、精确的选项关系信息, 并据此更具针对性地调整搜索策略, 就能进一步提高调优效率. 在编译器测试[10]中, 单一的选项配置往往无法覆盖所有代码路径和分支, 从而导致某些特定代码分支被忽略, 隐藏潜在的缺陷. 通过引入多样化的选项序列编译测试样例[11,12], 可以有效扩大测试的覆盖范围, 揭示在单一优化配置下难以发现的漏洞或编译错误, 从而增强编译器测试的有效性. 此外, 对选项关系的深入理解可以指导编译器进行更有针对性的测试, 提高测试效率, 避免重复无效的测试.

系统分析编译选项关系的挑战主要在于编译器本身的复杂性. 现有的程序分析技术难以准确全面地解析完整的编译流程. 编译器的优化过程包括多个阶段, 每个阶段都可能受到不同选项的影响, 这些影响往往是非线性、隐式且难以直接追踪的. 同时, 选项的内部约束关系、对程序特征和机器架构等外部环境的依赖关系往往交织在一起, 一个选项可能同时受到多重因素的影响, 进一步增加了选项关系分析的难度.

针对上述挑战, 本文基于 GCC 编译器源码对编译选项的关系进行系统分析, 揭示了影响编译器选项优化效果的两类主要关系: 内部约束关系和外部环境依赖关系. 其中, 内部约束关系是指由编译器自身的选项处理机制和 Pass 管理器直接决定且始终成立的选项之间的约束, 具体包括控制关系、启用关系和禁用关系; 外部环境依赖关系是指编译器在优化过程中, 为判断特定优化策略是否适用 (即合法且有收益) 时, 必须遵循的、由外部环境决定的约束条件, 其来源主要包括目标机器架构和目标程序特征. 本文首先利用静态分析技术, 基于编译器源码构建优化选项相关的工作流图来识别选项关系, 总结不同关系下选项的优化范式, 识别出 68 对内部约束关系和 103 对外

部环境依赖关系 (包括 51 个机器架构敏感选项和 52 个程序特征敏感选项). 此外, 还设计了基于动态分析的选项有效性自动验证框架, 用于进一步验证在不同外部环境下选项的有效性. 最后, 本文将选项关系分析结果应用于 3 个编译相关的技术场景: 编译选项自动调优、编译器测试和编译器开发调试, 进一步证明了选项关系分析的重要性.

综上所述, 本文的主要贡献如下.

1) 基于编译器源码系统性地分析了编译选项的内外作用机制, 揭示了影响编译优化选项作用效果的两类典型关系: 内部约束关系和外部环境依赖关系; 使用静态分析技术识别这两类关系, 并设计动态框架以评估选项在不同外部环境下的有效性.

2) 在编译选项自动调优方面, 提出基于选项关系感知的遗传调优算法. 在 x86 平台以 GCC-15.2.0 版本提供的全部选项作为搜索空间进行实验. 结果表明, 该算法在 cBench 与 SPEC CPU2017 上相比传统遗传算法的平均性能提升 3.65%.

3) 在编译器测试序列生成方面, 设计基于选项关系的测试序列生成方法. 在 x86 平台 GCC-15.2.0 版本下结合 Csmith 自动生成的程序, 基于指令调度相关选项关系生成多样化的选项序列, 最终相较于随机生成算法, 效率提高了 51.7%.

4) 在编译器开发与调试方面, 通过真实案例展示选项关系分析结果在实际开发中的应用价值. 该分析结果能够为开发者提供可解释的优化路径和依赖约束信息, 辅助定位优化失效原因并指导优化选项配置.

本文第 1 节介绍 GCC 编译优化选项的工作原理. 第 2 节介绍与编译优化选项相关的两个研究方向: 编译器测试和编译选项自动调优. 第 3 节分析影响编译选项优化效果的两类关系. 第 4 节展示编译选项关系的 3 个应用案例, 包括提高编译选项自动调优效率、提高编译测试效率以及提高开发者的开发效率. 第 5 节讨论当前工作的局限性和可扩展性. 第 6 节总结全文.

1 GCC 编译优化选项工作原理

本研究主要针对 GCC 编译器[1]的编译优化选项, 介绍 GCC 编译优化选项的工作原理.

1.1 基于 Pass 的编译流程

GCC 的编译优化过程如图 1 所示. 为了便于组织与调度优化, GCC 采用 Pass 管理机制. Pass 是编译器在某个阶段对源程序或中间表示进行的一次完整遍历或优化处理. 在编译过程中, GCC 将优化操作划分为若干个 Pass, 每个 Pass 完成一个具体的编译优化任务, 且每个 Pass 的输出作为下一个 Pass 的输入. 编译时, GCC 会按预定顺序依次执行所有 Pass, 完成整个编译过程.

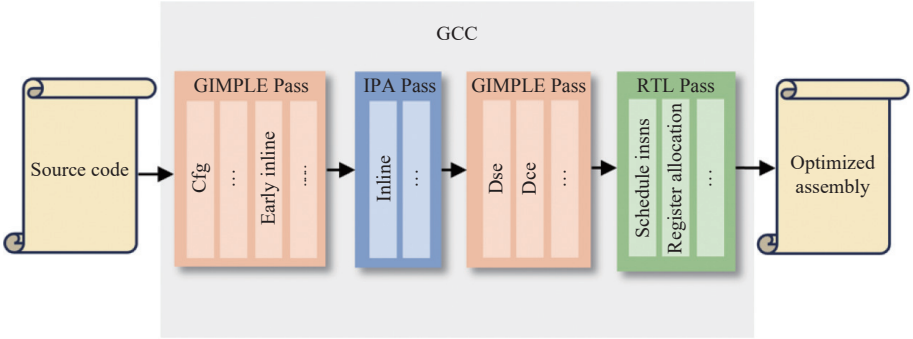


图 1 GCC 编译流程

GCC 的 Pass 可分为 3 种类型, 按照编译过程的整体顺序依次为: GIMPLE Pass、IPA Pass (interprocedural analysis pass) 和 RTL Pass (register transfer language pass). 其中, GIMPLE Pass 主要在较高级别的中间表示上进行优化, 比如常量折叠、条件传播和循环优化等; IPA Pass 主要进行跨函数优化, 针对多个函数之间的关系进行优化, 比如内联优化; RTL Pass 是最接近硬件层的优化, 主要涉及寄存器优化和指令调度等.

在 GCC 中, 每个 Pass 都由两个函数组成: gate 函数和 execute 函数. gate 函数用于判定 Pass 是否执行, execute 函数则决定 Pass 的具体执行逻辑. 在编译过程中, 当某个 Pass 被调用时, 编译器首先检查该 Pass 的 gate 函数返回值. 如果返回值为真, 则继续执行 execute 函数; 否则跳过该 Pass.

1.2 编译优化选项

GCC 的编译优化选项通常以整数型变量表示, 其取值用于控制对应优化的启用或禁用, 用于控制特定类型的优化, 如内联、死代码消除、常量折叠等. 它们通常在 gate 函数中决定 Pass 是否启用, 或在 execute 函数中控制其具体执行内容. GCC 中编译选项之间的依赖与交互关系极为复杂, 每个选项可能会影响多个 Pass, 每个 Pass 可能会受多个选项的影响, 这种复杂的映射关系增加了开发者正确使用编译选项的难度.

在命令行层面, GCC 对优化选项的处理遵循“后项优先”原则. 选项通常通过命令行参数-foption 启用或-fno-option 禁用. 当用户在同一命令中多次指定冲突或重复的选项时, GCC 会按从左至右的顺序读取, 并以最后出现的选项状态作为最终有效状态.

GCC 的 Pass 执行顺序被硬编码于编译器底层的逻辑调度中. 一旦命令行参数解析完毕并确定了各优化选项的最终启用状态, 这些 Pass 将严格按照编译器预设的顺序执行. 因此, 开发者只需决定选项的启用或禁用, 无法干预 Pass 之间的执行先后顺序.

在编译器源码中, 选项以变量 flag_option 的形式存储, 值为 1 时启用, 值为 0 时禁用. 本文同时使用-foption 和 flag_option 表示选项.

2 编译选项相关工作

与编译选项相关的研究方向主要有编译器测试和编译选项自动调优.

2.1 编译器测试

编译器作为软件开发中的核心组件, 其软件缺陷可能会导致自身崩溃, 或生成错误的二进制代码, 进而引发程序执行时触发意外行为. 编译器测试[10]是一种用于发现编译器缺陷的技术, 通过持续输入测试样例以触发编译器的异常行为. 本节从测试程序生成和测试预言构造两方面系统介绍编译器测试技术.

● 在测试程序生成方面, 基于语法的方法通过定义符合语法的生成规则构造测试程序. 其中, Csmith[13]是应用最广泛的随机 C 程序生成器. Lidbury 等人[14]进一步基于 Csmith 开发了 CLsmith, 用于生成测试 OpenCL 编译器的程序. Cummins 等人[15]设计了 DeepSmith, 基于 LSTM 自动学习 OpenCL 语法, 用于指导测试程序生成. 基于变异的方法通过修改已有的种子程序生成新的测试程序. 例如 Zhang 等人[16]引入了骨架程序枚举的概念, 通过替换程序的变量生成新的控制流结构; Orion[17]通过随机修剪未执行的语句对程序进行变异. 灰盒测试方法进一步引入代码覆盖率等反馈信息, 以指导生成新的测试程序, 从而触发编译器更深层次的代码逻辑. 例如, GrayC[18]和 Superion[19]利用覆盖率的变化, 在抽象语法树级别变异生成新的测试程序; 欧先飞等人[20]提出语义可感知的灰盒测试方法, 设计了具备上下文多样性的变异操作符; Tzer[21]基于覆盖率驱动, 同时变异生成测试 IR 文件及其对应的优化 Pass, 用于测试 TVM 编译器; JITfuzz[22]则针对 JVM, 通过覆盖率反馈结合优化激活与控制流增强变异, 提高了测试效果.

● 在测试预言构造方面, 主流方法包括随机差分测试 (randomized differential testing, RDT)[23] 和输入模等价 (equivalence modulo inputs, EMI)[17]. 随机差分测试使用多个具有相同规约的编译器互相揭错, 在给定相同输入时, 若各编译器的输出不一致, 则表明至少有一个编译器存在错误. 差分优化级别测试 (different optimization level, DOL)[24] 是随机差分测试的一种变体, 它通过设置不同的优化级别选项对编译器进行测试. 此外, Chen 等人[12]和 Jiang 等人[11]根据测试程序的特征选择不同的优化选项序列, 以实现更有针对性的测试并进一步提高测试效率. 但现有方法在生成选项序列时将选项视为独立的个体, 未考虑选项之间的复杂关系, 导致生成的序列中存在大量无效选项, 使得编译器后端并未执行预期的转换逻辑, 造成测试不充分. 相比之下, 本文提出的基于选项关系的测试序列生成方法, 通过分析选项关系, 预先剔除不满足约束关系的选项序列, 显著压缩搜索空间, 保证生成的每一组

序列在目标架构与程序下均具有实质性的执行效力, 提高了测试效率及测试深度. 输入模等价方法在种子程序的基础上生成新的等价程序作为测试用例, 通过比较两者输出结果判断编译器是否存在错误. 例如, Athena[25]在未执行代码中插入或者删除语句, 并使用马尔可夫链蒙特卡洛采样策略增强新程序与种子程序的差异性; Hermes[26]则通过在活跃区域插入死代码或其他等价变异手段, 进一步探索编译器的优化行为.

近年来, 研究者们开始尝试将大模型与编译测试技术结合, 进一步提高测试效率[27−29]. Gu 等人[27]提出一种基于大模型的测试样例生成方法, 并设计过滤策略来确保生成的测试样例不包含未定义行为或语法错误, 从而在保证生成测试样例质量的同时提高测试覆盖率. Yang 等人[29]设计了与编译器源码结合的、基于大模型的白盒测试工具 WhiteFox, 用于测试编译器中的深层逻辑错误. Gao 等人[30]利用大模型, 基于历史编译器漏洞的测试样例生成新的测试程序. Ou 等人[31]设计了基于大模型的测试框架 MetaMut, 使用编译器领域知识指导大模型自动完成变异种子生成、测试样例生成到反馈优化的全流程.

2.2 编译选项自动调优

编译选项自动调优技术旨在通过自动化方法搜索更优的选项序列[6,7], 现有研究主要有两大方向: 优化搜索空间[4]和优化搜索策略[32].

搜索空间是指所有可能的选项配置的集合, 大致有两种类型: 一是选择选项的状态或者参数的值, 例如在 GCC 中启用或者禁用内联选项 (-finline 和-fno-inline), 或者选择内联函数的体积上限 (-finline-limit=n); 二是选项顺序的排列, 如 LLVM 中的 Pass 顺序. 优化搜索空间主要有离线优化和在线优化两种方法.

● 离线优化是指在调优前, 基于编译器领域知识或数据驱动方法, 根据选项关系或其重要性对搜索空间进行优化. 例如, Ashouri 等人[33]基于 LLVM 的-O3 优化序列中每个选项出现的频率和顺序构建选项关系图, 将所有选项分为 5 组, 固定每组内选项的相对顺序和出现频率. 调优时以组为粒度选择选项, 从而显著减小搜索空间. Zhu 等人[2]根据 GCC 官方文档中的选项功能描述提取选项关系, 调优过程中生成新序列时遵循该选项关系, 减少无效的探索次数. 此外, 他们还基于 GCC 官方文档中介绍的选项针对的程序优化特征[3], 筛选适合程序的选项. 相比于 Zhu 等人依赖于人工梳理且描述粗略的 GCC 官方文档提取选项关系, 本文方法直接从编译器源码层面提取选项约束关系, 识别结果更加严谨且全面. Liang 等人[34]通过迭代搜索训练集中每个程序的最优编译选项序列, 并建立奖励矩阵评估每个最优序列在训练集中不同程序上的表现, 采用贪心算法选出在训练集中综合表现最优的 50 个选项序列作为核心集合, 调优时以核心集合中的子序列为单位进行搜索, 显著提高调优效率. Liu 等人[35]提出 ICMC, 通过分析选项之间的相互影响程度构建子序列, 调优时以子序列为单位进行搜索, 以减小搜索空间. Pan 等人[36]通过迭代搜索训练集中每个程序, 挖掘具有协同作用的选项对, 调优时根据程序特征为其选择合适的协同选项对作为搜索空间, 进一步提升调优效率. 数据驱动的方法本质上是依赖训练集学习选项与性能间的关联, 效果高度依赖训练集质量, 当目标程序特征偏离训练样本时, 其预测结果往往失效. 本文提出的基于选项关系的遗传算法是一种“白盒”驱动的方法, 直接基于编译器源码获得准确的选项关系, 无需任何数据样本, 直接通过在目标架构上一次性编译目标程序获得选项关系及其有效性的结果, 进一步指导搜索方向, 压缩搜索空间, 在调优成本和效率方面显著优于离线学习方法.

● 在线优化是指在调优过程中根据程序性能反馈, 动态调整搜索空间. Chen 等人[8]设计了 BOCA, 提出只有少部分选项在优化中起关键作用, 构建随机森林模型预测重要选项, 在迭代搜索的过程中集中探索重要选项. Park 等人[9]在调优过程中先初步识别对性能影响较大的选项, 再深入挖掘与之具有协同效应的其他重要选项. Zhu 等人[3]提出 CFSCA, 在调优过程中使用翻转测试筛选关键选项, 并在后续搜索中重点探索这些选项. 现有的在线优化方法完全依赖调优过程中的程序性能反馈逐步优化搜索空间, 但由于单次编译运行程序代价高昂且迭代次数有限, 在高维搜索空间下, 仅靠有限的性能采样数据难以实现高效的空间收敛. 本文提出的基于选项关系的遗传算法在调优开始前, 直接通过对编译器源码中选项关系的静态分析以及选项有效性的验证压缩搜索空间, 并为调优策略的搜索方向提供指导, 避免了调优初期的盲目在线搜索, 从而提高调优效率.

选项自动调优的搜索策略主要有 3 类: 启发式搜索策略、基于机器学习和深度学习的搜索策略以及基于强化

学习的搜索策略.

● 启发式搜索策略中, Bodin 等人[37]首次在自动调优中使用了随机算法, Garciarena 等人[38]提出了基于遗传算法的搜索策略. Ansel 等人[39]认为单一算法无法满足所有程序的调优需求, 提出了 OpenTuner, 该框架可以并行运行多种搜索算法, 包括遗传算法和进化算法等. OpenTuner 在搜索过程中会给表现更好的搜索算法分配更多的测试机会, 并通过共享数据库保存搜索结果, 使得一个算法发现的较优选项配置可被其他算法复用. Gao 等人[40] 在调优过程中使用模拟退火算法, 通过调整接受概率避免搜索陷入局部最优.

● 基于机器学习和深度学习的搜索策略通过训练模型学习程序在不同选项序列下的性能变化, 从而预测最优或较优选项序列. Milepost GCC[41]是第 1 个基于机器学习的自动调优框架, 它根据程序的静态特征 (如控制流和数据流信息) 预测最优的选项序列. Zhu 等人[42]提出 CompTuner, 使用粒子群优化算法[43]根据当前序列与迭代中表现最优序列之间的相似性调整搜索方向. Roy 等人[44]在调优时协同多个轻量级贝叶斯优化模型, 针对不同类型的子空间选择最合适的优化模型, 以实现更具针对性的优化. Hellsten 等人[45]针对搜索空间中未知的参数约束关系, 使用随机森林模型预测参数配置可行的概率, 优先采样既有可能提升性能又具有较高概率合法的配置, 显著提升搜索的效率和稳定性. Cummins 等人[46]使用卷积神经网络直接从程序的中间表示中学习特征, 以预测其最优配置. Mendis 等人[47]设计了 Ithemal, 使用分层递归神经网络预测基于 x86 指令的代码片段吞吐量.

● 基于强化学习的搜索策略将自动调优过程建模为一个马尔可夫决策过程. SRTuner[9]和 DynaTune[48]将搜索策略抽象为多臂老虎机问题[49], 并使用 UCB (upper confidence bound) 平衡探索与利用. Haj-Ali 等人[50]的 Neuro-Vectorizer 基于强化学习选择最佳循环优化参数; Haj-Ali 等人[51]的 AutoPhase 使用深度强化学习选择程序最优的选项序列. Liang 等人[34]使用强化学习预测程序的最优序列. 由于最优序列可能不唯一, 该方法不直接预测具体序列, 而是输出各候选序列为最优的概率分布, 以提高模型的鲁棒性和泛化能力. Cummins 等人[52]设计了 CompilerGym, 是专门为编译器优化任务设计的强化学习环境, 提升了将强化学习应用在编译器优化中的效率, 降低了非人工智能/编译从业人员使用强化学习进行编译器优化的门槛.

此外, 也有研究者开始探索使用大模型进行编译优化. Cummins 等人[53]提出 LLM Compiler, 在 Code LLaMA 的基础上进行预训练和微调, 通过在大规模 IR 语料上学习 Pass 对 IR 性能的影响以及 Pass 之间的相互关系, 实现对 IR 的最优 Pass 序列的预测, 用于减小生成代码体积. Pan 等人[54]进一步提出 Compiler-R1, 利用强化学习技术对大模型进行端到端微调, 显著提升了模型对复杂选项空间的搜索能力, 证明了强化学习技术在选项关系理解和性能建模方面的巨大潜力.

本文提出的选项关系分析方法与现有的搜索策略正交, 可以作为先验过滤器集成到现有的调优策略中, 通过剔除冗余选项配置和基于选项关系引导搜索, 进一步提升调优效率.

3 编译选项关系分析

本节对影响编译选项优化效果的两类关系进行定义和分析, 包括: 内部约束关系和外部环境依赖关系. 本节对每种关系进行定义和介绍, 使用静态分析技术定位满足相应关系的选项, 并设计动态验证框架验证选项在不同外部环境下的有效性.

3.1 选项内部约束关系

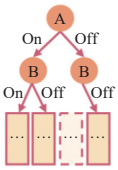
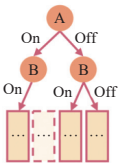
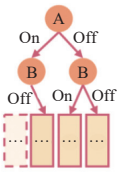
3.1.1 内部约束关系定义

编译选项的内部约束关系是指编译器自身的选项处理机制和 Pass 管理器为保证优化管道 (pipeline) 执行的正确性和有效性而设定的约束关系. 这些约束关系仅源自编译器自身, 完全独立于目标程序或目标机器架构. 具体包括 3 种: 控制关系、启用关系和禁用关系, 其类型和定义如表 1 所示.

● 控制关系: 当选项之间存在控制关系时, 被控选项仅在控制选项启用的前提下才可能生效; 控制选项关闭时, 被控选项始终处于失效状态, 因为它所控制的优化代码路径是不可达的. 如图 2 所示, 在 pass_sched2 的执行流程中, flag_schedule_insns_after_reload 在 gate 函数中决定是否执行指令重载后的调度. 只有当该选项启用时, 位于

execute 函数中的 flag_selective_scheduling2 和 flag_sched2_use_superblocks 控制的优化才有机会被执行, 从而影响调度策略.

表 1　编译选项关系

关系类型	关系定义	关系原理	有效组合	示意图
控制关系	¬flagA ⇒ ¬Eff(flagB)	if(flagA==False) then flagB=False	-fA -fB -fA -fno-B -fno-A -fno-B	
启用关系	flagA ⇒ (flagB)	if(flagA==True) then flagB=True	-fA -fB -fno-A -fB -fno-A -fno-B	
禁用关系	flagA ⇒ ¬Eff(flagB)	if(flagA==True) then flagB=False	-fA -fno-B -fno-A -fB -fno-A -fno-B	

```
/* gcc/sched-rgn.c */
class pass_sched2 : public rtl_opt_pass
{
public:
  pass_sched2 (gcc::context *ctxt)
    : rtl_opt_pass (pass_data_sched2, ctxt)
  {}
  /* opt_pass methods: */
  virtual bool gate (function *){
    return optimize > 0 &&
    flag_schedule_insns_after_reload
      && !targetm.delay_sched2;
  }
  unsigned int pass_sched2::execute (function *){
if (flag_selective_scheduling2)
    run_selective_scheduling();
else {
    if(flag_sched2_use_superblocks)
        schedule_ebbs();
    else
        schedule_insns();
    }
return 0;
}
}
```

图 2　选项控制和禁用关系示例代码

● 启用关系: 当选项之间存在启用关系时, 启用上位选项将强制启用其覆盖范围内的下位选项, 以保证优化意图的一致性. 如图 3 所示, flag_unroll_all_loops (展开所有循环) 是一个比 flag_unroll_loops (基于启发式规则判定的循环展开) 更激进的选项, 前者优化范围完全包含了后者, 因此启用前者时, 会强制启用后者, 确保相关优化被执行.

```
/* gcc/toplev.c */
/* Unrolling all loops implies that standard loop unrolling must also be done. */
if (flag_unroll_all_loops)
    flag_unroll_loops = 1;
```

图 3　选项启用关系示例代码

● 禁用关系: 当选项之间存在禁用关系时, 其中一个选项启用时, 另一个选项所控制的优化逻辑分支会被绕过, 使其状态不会对编译结果产生影响. 这通常是由于两个选项分别控制了两种对立的、不可兼得的优化策略或者代码路径. 如图 2 所示, pass_sched2 包含两种互斥调度策略, 并通过 if-else 控制流结构实现. 启用 flag_selective_scheduling2 将执行 if 分支下的调度策略, 从而必然跳过 flag_sched2_use_superblocks 所在的 else 分支, 其效果等价于被禁用.

3.1.2　内部约束关系识别

本文设计了选项内部约束关系识别算法, 分别识别选项间的 3 类内部约束关系. 这 3 类关系有两种表现形式: 显式约束和隐式约束. 显式约束是指被约束选项的状态会因约束选项的状态而直接改变, 主要体现在编译优化的初始阶段, 编译器读取每个选项的输入值, 并根据内部约束关系直接更新被约束选项的状态, 这种约束主要包括启

用关系和禁用关系. 隐式约束是指被约束选项的状态不被直接改变, 约束选项通过改变编译优化的执行路径, 从而影响被约束选项最终的作用效果, 体现在优化管道每个 Pass 的具体执行过程中, 主要包括控制关系和禁用关系.

具体而言, 我们基于 CodeQL[55] 对 GCC 源码进行静态分析, 为编译优化的初始阶段及优化管道中的每个 Pass 构建控制流和数据流图, 将其称为选项的工作流图. 在构建过程中, 重点关注选项状态的赋值节点和读取节点, 因为这两类节点直接反映选项之间的依赖或触发关系.

具体的关系识别算法如算法 1 所示, 我们分别识别选项的显式约束关系 (第 3–13 行) 和隐式约束关系 (第 14–24 行). 对于显式约束关系, 我们遍历每个选项状态赋值节点, 若其上游条件判断节点的判断对象为另一个选项的状态, 则确认其存在显式约束关系 (第 6–9 行). 随后, 根据被约束选项状态的赋值方式判断具体的约束关系类型 (第 10–13 行). 对于隐式约束关系, 我们遍历每个选项状态读取节点, 若其上游条件判断节点的判断条件是另一个选项的状态时, 则确认其存在隐式约束关系 (第 16–19 行). 随后, 根据两个选项的控制结构类型关系 (如嵌套或互斥分支) 进一步判断具体的约束关系类型 (第 20–24 行).

算法 1. 选项内部约束关系识别算法.

输入: 选项工作流图 G, 选项状态读取节点集合 RN, 选项状态赋值节点集合 WN;
输出: 选项内部约束关系集合 R.

1. **Function** *Identify_ explicit_ internal_relationship*(G, RN, WN):
2. R = {'*control*': [], '*enable*': [], '*disable*': []}
3. **for** each $node_{write} \in WN$ **do** //遍历每个选项状态赋值节点
4. $option_A = node_{write}.write\_option$
5. $value = GetAssignedValue(node_{write})$
6. $upstream\_conditions = FindUpstreamConditionNodes(G, node_{write})$
7. **for** each $node_{up} \in upstream\_conditions$ **do** //遍历每个选项状态赋值节点的上游条件判断节点
8. $option_B = node_{up}.read\_option$
9. **if** $option_B$ not none **then** //若上游节点读取选项状态, 两选项存在显式约束
10. **if** $value$ == false **then** //若 A 选项赋值为 false, 即为禁用关系
11. R['*disable*'].*add*(($option_A$, $option_B$))
12. **elif** $value$ == true **then** //若 A 选项赋值为 true, 即为启用关系
13. R['*enable*'].*add*(($option_A$, $option_B$))
14. **for** each $node_{read} \in RN$ **do** //遍历每个选项状态读取节点
15. $option_A = node_{read}.read\_option$
16. $upstram\_conditions = FindUpstreamConditionNodes(G, node_{read})$
17. **for** each $node_{up} \in upstream\_conditions$ **do** //遍历每个选项状态读取节点的上游条件判断节点
18. $option_B = node_{up}.read\_option$
19. **for** $option_B$ not none **then** //若上游节点读取选项状态, 两选项存在隐式约束
20. $sturct\_type = GetControlStructure(node_{up}, node_{read})$
21. **if** $struct_{type}$ == '*Nested_IF*' **then** //若两节点为嵌套关系, 即为控制关系
22. R['*control*'].*add*(($option_A$, $option_B$))
23. **if** $struct_{type}$ == '*IF_ELIF*' **then** //若两节点为互斥分支, 即为禁用关系
24. R['*disable*'].*add*(($option_A$, $option_B$))
25. **return** R

最终我们在 GCC-15.2.0 中识别出涉及 84 个编译选项的 68 对内部约束关系, 表 2 展示了控制关系的部分示

例, 完整结果提供在第 6 节链接中. 其中, 1 → 2 表示选项 1 控制选项 2; (1,2) → 3 表示选项 1 和 2 共同控制选项 3, 即当选项 1 和 2 同时关闭时, 选项 3 无效.

表 2　控制关系选项示例

组	选项ID	选项名	关系	组	选项ID	选项名	关系
①	1	-fgcse	1 → 2 1 → 3	⑥	1	-ftree-ccp	1 → 6 (1,2) → 3 (1,2) → 4 (1,2) → 5
	2	-fgcse-lm			2	-fipa-cp	
	3	-fgcse-las			3	-fipa-cp-clone	
②	1	-freorder-blocks	1 → 2		4	-fipa-vrp	
	2	-ftracer			5	-fipa-bit-cp	
③	1	-fsplit-wide-types	1 → 2		6	-ftree-bit-ccp	
	2	-fsplit-wide-types-early		⑦	1	-finline	1 → 2 1 → 3 1 → 4 1 → 5 1 → 6 1 → 7 1 → 8
④	1	-fssa-phiopt	1 → 2		2	-fearly-inlining	
	2	-fhoist-adjacent-loads			3	-finline-functions	
⑤	1	-ftree-pre	(1,2) → 3 (1,2) → 4		4	-finline-small-functions	
	2	-fcode-hoisting			5	-finline-functions-called-once	
	3	-ftree-tail-merge			6	-findirect-inlining	
	4	-ftree-partial-pre			7	-fpartial-inlining	
					8	-finline-atomics	

3.2　选项外部环境依赖关系

编译选项的优化效果不仅受到选项内部约束关系的影响, 还受到外部环境因素的制约. 具体而言, 编译器的优化策略当且仅当满足外部环境的约束条件时才能生效, 我们将这种关系称为程序的外部环境依赖关系. 外部环境主要包括目标机器架构和目标程序特征两类因素, 我们将受到这两种因素影响的选项分别称为机器架构敏感选项和程序特征敏感选项. 由于外部环境依赖选项的有效性在不同机器架构或程序特征下存在差异, 本节首先通过静态分析技术识别所有外部环境依赖关系的相关选项, 再利用动态分析技术验证选项在不同环境下的有效性.

3.2.1　外部环境依赖关系定义

● 机器架构敏感选项是指专为特定硬件特性设计的选项. 其优化效果受目标指令架构集、微架构设计等因素约束, 仅在目标机器架构满足相应要求时才能生效. 在编译器的初始化阶段, 编译器通过一系列钩子函数检测目标架构是否具备相关特性. 大部分机器架构敏感选项都属于 RTL Pass, 例如-fauto-inc-dec 依赖 ARM 架构以优化自增和自减操作, 从而减少指令数量并提高缓存利用率; 也有少数位于 GIMPLE Pass, 如涉及预取机制的-floop-prefetch.

● 程序特征敏感选项是指优化效果高度依赖程序特征的选项. 只有当程序特征满足特定约束条件时, 该选项才能生效. 例如-ftree-sra 通过将聚合类型变量 (如数组、结构体等) 拆分为独立的标量变量以提高访存效率, 只有当程序中存在聚合类型变量时, 该优化才会生效.

3.2.2　外部环境依赖关系选项识别

为了识别外部环境依赖关系, 我们使用与第 3.1.2 节相同的静态分析技术, 分别分析对机器架构和程序特征存在依赖关系的选项.

● 机器架构敏感选项的识别算法如算法 2 所示. 由于机器架构类型繁多, 且同一架构系列下的不同微架构在硬件特性上也存在微小差异, 因此编译器在判断机器架构敏感选项的有效性时, 并未采用基于架构标签的显式匹配, 而是采用了一种“架构解耦”的抽象机制, 以保证优化决策的精确性. 在编译初始化阶段, 编译器通过一系列目标钩子函数探测当前机器架构的指令集扩展或物理约束 (如寄存器分配限制等) 特性, 并将这些探测结果封装在宏定义或 targetm、target 等结构体的状态变量中. 在执行机器架构相关优化时, 编译器在优化过程中遵循以下模式: 先检查当前机器架构相关变量是否满足优化要求, 当且仅当变量满足优化要求, 且相应选项开启时, 才会执行相应优化. 因此, 识别机器架构敏感选项时, 我们遍历每个选项状态读取节点, 分析其上游条件判断节点是否读取

了机器架构相关信息 (第 7 行), 若条件判断的依据是机器架构相关变量, 则判定该选项为机器架构敏感选项. 同时, 记录其依赖的机器架构判断节点, 便于后续识别其有效性. 最终, 我们在 GCC-15.2.0 中识别出 51 个机器架构敏感选项, 表 3 展示了部分典型选项及其依赖的机器架构特性, 并列出编译器判断其有效性依据的核心变量. 表 3 中给出的有效值表示在目标机器上该选项能够实际满足触发条件的变量取值域. 完整结果可在第 6 节链接中获取.

算法 2. 机器架构敏感选项识别算法.

输入: 选项工作流图 G, 选项状态读取节点集合 RN;
输出: 机器架构敏感选项及其相关机器架构判断节点集合 TO.

1. **Function** *Identify_ machine_ specific_ option*(G,RN):
2. $TO = \{\}$
3. **for** each $node_{\text{read}} \in RN$ **do** //遍历每个选项状态读取节点
4. $option_A = node_{\text{read}}.read\_option$
5. $upstream\_conditions = FindUpstreamConditionNodes(G, node_{\text{read}})$
6. **for** each $node_{\text{up}} \in upstream\_conditions$ **do** //遍历每个选项状态读取节点的上游条件判断节点
7. $machine\_var = node_{\text{up}}.read\_machine\_variable$
8. **if** $machine\_var$ not none **then** //若上游节点读取机器架构信息
9. $TO[option_A] = node_{\text{up}}$ //该选项即为机器架构敏感选项
10. **return** TO

表 3 目标机器敏感选项示例

选项名	相关机器架构特性	目标机器特性相关变量	有效值
-freorder-blocks -freorder-blocks-and-partition	跳转指令无需绑定寄存器	targetm.cannot_modify_jumps_p()	0
-freschedule-modulo-scheduled-loops	支持硬件循环计数器	target.code_for_doloop_end	≠0
-fshrink-wrap-separate	支持轻量级返回指令	target.shrink_wrap.get_separate_components SHRINK_WRAPPING_ENABLED	1 1
-fno-function-cse	直接调用常量函数地址效率低于调用寄存器存储地址	NO_FUNCTION_CSE	0
-fauto-inc-dec	支持自增/自减寻址模式	AUTO_INC_DEC	0
-fsched-stalled-insns-dep	指令流水线冲突敏感性高	target.sched.is_costly_dependence	≠0
-fbranch-count-reg	支持硬件循环计数器	target.have_doloop_end()	≠0
-fschedule-fusion	支持指令融合	target.sched.fusion_priority	NULL

● 程序特征敏感选项的识别算法如算法 3 所示. 实际上, 几乎所有编译选项的优化都依赖于特定的程序特征. 例如, 内联选项依赖函数调用, 循环优化选项依赖于循环等控制结构. 然而, 在 GCC 编译过程中, 程序特征会不断发生变化, 这意味着某些在原始源码中不存在的特征 (如常量传播后暴露的冗余指令) 可能会在优化过程中生成, 从而在后续优化阶段产生新的优化机会 (如死代码消除). 由于编译过程中程序特征的变化同时受目标程序静态特征和编译流程的影响, 这些变化具有高度的不确定性. 因此, 本研究将分析范围限定在静态特征敏感选项, 其依赖的程序特征必须已存在于源码的静态特征中, 且不会在编译过程中被其他优化阶段间接引入或动态生成. 例如, -finline-atomics 专门针对原子操作进行内联优化, 只有当源代码中包含原子操作时, 该优化才会被触发. 此类原子操作语义无法通过其他优化阶段间接引入, 从而规避了特征在编译过程中的动态性带来的不确定性. 对于该类选项, 编译器在编译优化过程中遵循以下模式: 首先检查该选项是否被启用, 随后检查程序中是否存在该选项所依赖的程序特征; 若存在, 则继续执行优化. 因此, 在识别程序特征敏感选项时, 我们遍历每个选项读取节点, 并分析其下游条件判断节点是否基于程序静态特征进行判断 (第 7 行). 若该条件判断依赖于程序静态特征, 则该选项被判定为程

序特征敏感选项 (第 8、9 行). 同时, 记录判断程序特征的相关节点, 便于后续验证其有效性. 最终, 在 GCC-15.2.0 中识别出 52 个程序特征敏感选项, 并在表 4 中列举了部分典型示例、对应的优化功能以及选项优化所依赖的特定程序特征, 当且仅当待编译程序具备相应列举的特征时, 选项的优化才会有效果. 完整结果可在第 6 节链接中获取.

算法 3. 程序特征敏感选项识别算法.

输入: 选项工作流图 G, 选项状态读取节点集合 RN;
输出: 程序特征敏感选项及其相关程序特征判断节点集合 PO.

1. **Function** *Identify_ program_ specific_ option*(G,RN):
2. $PO = \{\}$
3. for each $node_{\text{read}} \in RN$ **do** //遍历每个选项状态读取节点
4. $option_A = node_{\text{read}}.read\_option$
5. $downstream\_conditions = FindDownstreamConditionNodes(G, node_{\text{read}})$
6. **for** each $node_{\text{down}} \in downstream\_conditions$ **do** //遍历每个选项读取节点的下游条件判断节点
7. $state = ConditionDependsOnProgramFeature(node_{\text{down}})$
8. **if** *state* not none **then** //若下游节点基于程序静态特征判断
9. $PO[option_A] = node_{\text{down}}$ //该选项即为程序特征敏感选项
10. **return** PO

表 4 程序特征敏感选项示例

选项名	优化功能	相关程序特征
-finline-atomics	内联原子操作	原子操作
-fstdarg-opt	优化函数保存到栈上的stdarg寄存器数量	带可变参数的函数
-foptimize-strlen	启用字符串长度优化	字符串类函数
-fjump-tables	对足够大的switch语句使用跳转表	switch语句
-fdevirtualize	将虚函数调用转换为直接调用	虚函数
-fsplit-wide-types	将宽类型变量拆分到独立寄存器中	宽类型变量
-ftree-sra	对聚合变量进行标量替换	聚合变量

3.2.3 外部环境依赖关系验证

由于存在外部环境依赖关系的选项在不同的机器架构和目标程序下有效性不同, 本文构建了基于动态分析的选项有效性自动验证框架. 此处的有效性是指给定机器架构和目标程序, 选项所关联的编译优化 Pass 被真实激活, 且成功执行其核心代码转换逻辑. 该框架按照验证流程主要由 3 个模块组成: 静态注入模块、动态监控模块和有效性判定模块, 具体流程算法如算法 4 所示. 首先, 静态注入模块对编译器源码进行静态分析, 识别外部环境依赖关系选项并判断其有效性的关键节点 (即算法 2 和 3), 作为后续监控的注入点; 其次, 动态监控模块在实际编译过程中启动 GDB 调试器, 在每个选项关键节点的后继位置设置断点, 实时监控编译目标程序的控制流是否经过断点, 并记录其触发状态 (第 3 行); 最后, 有效性判定模块汇总断点触发日志, 根据断点触发情况输出验证结果. 若预设断点被命中, 则判定该选项在当前环境下有效, 反之则无效 (第 4–7 行).

算法 4. 外部环境依赖关系验证算法.

输入: 待编译程序 P, 机器架构敏感选项、程序特征敏感选项及其关键节点集合 KO=TO+PO;
输出: 有效选项集合 VO.

1. **Function** *Validate_ option_ effectiveness*(P,KO):
2. $VO = []$

3. *reach_state* = *Compile*(*P*,*KO*) //编译程序, 记录每个选项对应关键判断节点的触发状态
4. **for** each *option* ∈ *reach_state* **do**
5. *state* = *reach_state*[*option*]
6. **if** *state* **then** //若关键节点在编译过程中被触发,
7. *VO*.*append*(*option*) //则其对应的选项在当前环境中有效
8. **return** *VO*

对于机器架构敏感选项, 由于选项的有效性与目标程序无关, 只需在当前架构下编译任意程序即可验证其有效性, 验证结果可在同一架构下复用, 无需为不同的目标程序重复判断. 对于程序特征敏感选项, 选项有效性的验证结果与机器架构和目标程序相关, 只需在当前机器架构下针对目标程序验证一次有效性, 可以在相同机器架构和目标程序下复用, 无需重复判断.

4 案例分析

本节介绍选项关系在提高编译选项自动调优效率、提高编译测试效率和提高开发者的开发效率这 3 个领域的应用案例.

4.1 提高编译选项自动调优效率

传统的编译选项调优算法将搜索空间视为完全独立的选项组合, 忽略了选项间的语义关联性. 本节将编译选项的内部约束关系作为先验知识引入遗传自动调优 (genetic auto-tuning, GA) 算法, 提出基于选项关系感知的遗传自动调优 (relationship-aware genetic auto-tuning, R-GA) 算法, 以提高算法寻找最优选项序列的效率. 核心思想在于: 互相约束的编译选项在优化类型上通常具有相似性, 因此对于某个程序, 如果在开启某个选项后能够提升性能, 那么调整与其相关的其他选项状态, 也可能带来进一步的性能提升. 例如, 在 cBench 中的 automotive_bitcount 测试中, 开启-funroll-loops 后, 程序性能相较-O3 提升 19.7%, 由于-funroll-loops 受-funroll-all-loops 控制, 而后者进行了更为彻底的循环展开优化, 因此继续开启-funroll-all-loops 后, 程序性能进一步提升至 22.4%. 因此本文将有内部约束关系的选项分为一组, 在迭代过程中以组为单位进行调优, 以提高调优的针对性.

4.1.1 基于选项关系感知的遗传自动调优算法

具体算法如算法 5 所示. 与原先在整个搜索空间范围内随机改变选项状态不同, 本算法在生成新的选项序列时, 遍历每个有内部约束关系的选项组, 并随机选择是否改变组内选项状态 (第 1–8 行). 由于每个组内的选项优化类型相似, 因此程序性能的变化与选择改变的选项组优化类型密切相关.

算法 5. 基于选项关系感知的遗传自动调优算法.

输入: 种群数量 *num*, 迭代轮数 *round*, 变异率 $mutate_{\text{rate}}$, 选项组 *Group*;
输出: 最优性能 $perf_{\text{best}}$, 最优序列 seq_{best}.

1. **Function** *mutate*(*seq*):
2. $seq_{\text{new}} \leftarrow seq$; //变异生成新的选项序列
3. **for** each *group* ∈ *Group* **do** //遍历每个选项组
4. **if** $u \sim U[0,1] \leqslant mutate_{\text{rate}}$ **then** //以小概率随机选择改变选项组状态
5. **for** each *op* ∈ *group* **do**
6. **if** $u \sim U[0,1] \leqslant 0.5$ **then**
7. $seq[op] \leftarrow reverse\_state(seq[op])$;
8. **return** seq_{new}, *get_exec_time*(seq_{new})
9. **Function** *Initialization*(*num*): //初始化种群

10. $P \leftarrow \{(seq_{O3}, get\_exec\_time(seq_{O3})\}$;
11. **for** n from 1 to num **do**
12. $seq_{new}, perf_{new} \leftarrow mutate(seq_{O3})$; //以-O3 配置为起点随机生成新序列, 加入种群
13. $P.add((seq_{new}, perf_{new}))$;
14. **return** P
15. **Function** $crossover(seq_1, seq_2)$: //交叉生成新个体
16. $seq_{child1} \leftarrow \{\}$, $seq_{child2} \leftarrow \{\}$;
17. **for** each $group \in Group$ **do** //以组为单位进行随机交叉
18. **if** $u \sim U[0,1] \leqslant 0.5$ **then**
19. $seq_{child1}[group] = seq_1[group]$;
20. $seq_{child2}[group] = seq_2[group]$;
21. **else**
22. $seq_{child1}[group] = seq_2[group]$;
23. $seq_{child2}[group] = seq_1[group]$;
24. **if** $u \sim U[0,1] \leqslant 0.5$ **then** //随机选择一个新个体作为交叉结果
25. **return** seq_{child1}
26. **return** seq_{child2}
27. **Function** $relationship\_aware\_genetic()$:
28. $P \leftarrow Initialization(num)$;
29. **for** n from 0 to $round$ **do**
30. $seq_{parent1}, seq_{parent2} = random\_select(P)$; //随机选择优秀个体作为母体
31. $seq_{child} = crossover\left(seq_{parent1}, seq_{parent2}\right)$; //交叉生成新个体
32. $seq_{child}, perf_{child} = mutate(seq_{child})$; //在新个体基础上进行变异生成新序列
33. **if** $perf_{child} < perf_{worst}$ **then** //如果新序列性能优于种群中最差序列, 更新种群
34. $P.replace((seq_{worst}, perf_{worst}), (seq_{new}, perf_{new}))$;
35. $round = round + 1$;
36. $perf_{best}, seq_{best} = select\_best(P)$;
37. **return** $perf_{best}, seq_{best}$

在初始化种群阶段, 算法以-O3 为初始状态, 随机改变选项组的状态并测试相应性能, 添加到种群 (第 9–14 行); 在正式迭代过程中, 每次从种群中随机选择两个序列作为母体, 以组为单位进行交叉生成新序列 (第 15–26 行); 再在新个体的基础上进行变异操作, 遍历每个选项组, 以一定的变异率选择是否改变该组的选项状态 (第 1–8 行); 生成变异的新序列后, 对其性能进行测试, 如果新序列的性能超过种群中的最差序列, 则以其替换最差序列, 从而进一步优化种群 (第 33、34 行). 最后, 当达到最大迭代轮数时停止迭代, 从种群中选择性能最优的序列作为最终调优结果.

4.1.2 实验设置

实验使用 GCC-15.2.0 版本编译器, 运行平台为 x86-64 Linux. 初始搜索空间包含 209 个选项, 涵盖从 GCC 文档[56]和源代码中收集的所有用于优化程序性能的选项, 实验设定算法迭代轮数为 500 轮, 种群数量为 50.

- 机器配置: 所有实验都在 Ubuntu 20.04.6 LTS 的工作站上进行. 该工作站配置了 8 核 11th Gen Intel® Core™ i7-11700@2.50 GHz 处理器和 16 GiB 内存.
- 数据集: 我们从 cBench[57] 和 SPEC CPU2017[58]中分别选取 20 个和 10 个程序进行实验, 具体程序信息如表 5

和表 6 所示, 其中运行时间为使用 -O3 编译后的执行时间 (s). cBench 和 SPEC CPU2017 程序的筛选主要遵循以下两项原则: 首先, 程序的输出稳定且可验证, 以确保调优结果的正确性和有效性; 其次, 程序执行时间波动较小, 重复测量的性能方差较低, 从而保证实验结果的稳定且可复现. 因此, 对于 SPEC CPU2017, 本文只选择了 SPECrate 组件中的程序作为基准测试程序, 采用单核和单线程模式, 排除多核和多线程并发带来的性能波动. 最后, 由于 SPEC CPU2017 部分程序执行时间较长, 我们只选择了在 -O3 下执行时间在 500 s 以内的程序.

表 5 cBench 数据集信息

程序名	代码行数	运行时间 (s)	功能介绍	程序名	代码行数	运行时间 (s)	功能介绍
automotive_bitcount	945	3.00	执行位运算	network_dijkstra	199	2.35	最短路径算法
autmotive_qsort1	227	2.59	数组快速排序	network_patricia	634	3.03	路由表查找算法
automotive_susan_e	2 129	2.56	边缘检测算法	office_rsynth	5 412	8.34	语音合成算法
automotive_susan_s	2 129	3.02	角点检测算法	office_stringsearch1	508	2.81	字符串匹配算法
bzip2d	7 200	1.97	文件解压算法	security_blowfish_d	1 524	3.47	加密算法
bzip2e	7 200	3.12	文件压缩算法	security_blowfish_e	1 524	3.49	解密算法
consumer_jpeg_c	26 950	2.75	图像压缩算法	telecom_adpcm_c	307	12.05	音频压缩编码
consumer_lame	21 824	3.05	MP3音频编码	telecom_adpcm_d	6 049	7.38	音频压缩解码
consumer_tiff2bw	22 271	2.97	图像灰度化	telecom_CRC32	389	2.25	校验码算法
consumer_tiffdither	22 184	1.50	图像抖动算法	telecom_gsm	391	2.92	语音编解码算法

表 6 SPEC CPU2017 数据集信息

定点类型				浮点类型			
程序名	代码行数 (k)	运行时间 (s)	功能介绍	程序名	代码行数 (k)	运行时间 (s)	功能介绍
505.mcf_r	3	396.66	路径规划	507.cactuBSSN_r	257	213.70	数值相对论
520.omnetpp_r	134	435.91	离散仿真	519.lbm_r	1	262.86	流体动力学
523.xalancbmk_r	520	290.53	文档转换	526.blender_r	1 577	337.11	3D渲染动画
548.exchange2_r	1	215.41	递归解生成	538.imagick_r	259	445.13	图像处理
557.xz_r	33	168.10	数据压缩	544.nab_r	24	324.49	分子动力学

● 测量方式: 为保证实验结果的准确性, 所有实验均在单台工作站上进行, 每次仅运行一个程序. 我们将目标进程绑定到一个隔离的 CPU 核心上, 以避免其他进程的中断或干扰. 对于 cBench 程序, 使用 Perf 工具[59]测量其执行时间, 取 5 次运行的平均值作为其最终执行时间; 对于 SPEC CPU2017 程序, 遵循其官方标准规范取 3 次运行的中位数值作为其最终执行时间, 由于其执行时间较长, 我们将迭代次数设置为 200 轮. 在每轮调优之前, 使用 -O3 编译并执行程序, 将其输出结果记录为标准输出. 在调优过程中, 当且仅当程序的编译与执行均成功, 且输出和标准输出完全一致时, 其性能数据才会被视为有效结果并纳入统计, 从而保证实验结果的可靠性与可重复性.

● 对比算法: 将随机算法 (RIO)、遗传算法 (GA)、基于启发式算法的 OpenTuner[39]及基于机器学习的 BOCA[8]与基于选项关系感知的遗传自动调优算法 (R-GA) 进行比较, 其中 GA 在迭代过程中通过在整个搜索空间内随机改变选项状态生成新的选项序列, 其余设置均和 R-GA 保持一致, OpenTuner 和 BOCA 的代码和参数设置与原论文保持一致.

4.1.3 实验结果

表 7 展示了不同算法在 cBench 和 SPEC CPU2017 下的最终性能提升百分比和调优时间, 并加粗了每个程序性能提升最大和调优时间最短的数据. R-GA 最终得到的选项序列性能相较-O3 平均提高 14.53%, 在 70% (21/30) 的程序中达到最优, 分别比 OpenTuner 和 BOCA 平均提高 2.39% 和 1.65%. 在调优时间方面, R-GA 平均调优时间为 24.33 h, 在 70% (21/30) 的程序中达到最优, 分别为 OpenTuner 和 BOCA 平均调优时间的 86.74% 和 91.06%. 结果表明在选项关系的指导下, R-GA 能更高效地收敛至性能更优的选项序列. 当目标程序的执行时间较短时 (如 cBench 中程序), 调优时间主要由程序执行时间和算法效率共同决定. 由于 BOCA 在每轮迭代中都需要重复训练

模型指导搜索方向, 因此整体耗时较长. 而当目标程序的执行时间较长时 (如 SPEC CPU2017 中程序), 调优时间主要受程序执行时间决定, 算法的额外开销可忽略, 此时关键在于算法能否快速收敛至高性能选项区域. 由于 R-GA 几乎没有额外计算开销, 且能在选项关系的引导下更快定位潜在的高效区域, 该算法在 cBench 和 SPEC CPU2017 上均表现出较高的调优效率.

表 7　不同算法调优后程序性能相较于-O3 平均性能提升和调优时间

程序名	性能提升 (%)					调优时间 (h)				
	R-GA	GA	BOCA	OP	RIO	R-GA	GA	BOCA	OP	RIO
automotive_bitcount	32.25	31.91	**33.08**	30.95	31.13	**1.83**	2.11	3.55	2.01	2.37
automotive_qsort1	**4.96**	4.29	4.17	3.72	3.57	**1.93**	2.05	4.09	2.06	2.11
automotive_susan_e	**8.99**	6.88	7.62	5.34	5.30	**1.98**	2.03	4.32	2.15	2.10
automotive_susan_s	**20.20**	13.78	14.91	15.13	12.29	**2.25**	2.58	4.52	2.50	2.83
bzip2d	**5.32**	3.49	3.87	4.19	1.50	2.01	**1.95**	4.36	2.30	1.98
bzip2e	**3.82**	2.64	−0.45	1.20	1.41	2.75	**2.74**	4.53	3.63	2.77
consumer_jpeg_c	**5.35**	2.20	2.69	1.67	0.43	**3.14**	3.25	5.43	4.39	3.28
consumer_lame	9.31	11.65	**14.72**	6.07	5.27	**3.62**	4.15	6.32	5.94	4.88
consumer_tiff2bw	**12.67**	8.92	12.38	10.14	6.07	3.14	**3.03**	5.91	4.66	3.08
consumer_tiffdither	**4.93**	1.45	1.91	4.03	0.85	2.11	**1.95**	5.02	3.68	1.97
network_dijkstra	**5.06**	3.00	2.69	3.41	0.71	**1.84**	2.06	4.06	2.08	2.20
network_patricia	6.70	6.36	**7.05**	6.41	5.66	**2.17**	2.26	4.32	2.27	2.44
office_rsynth	**33.25**	32.26	29.26	31.75	28.78	6.59	7.48	7.28	**5.60**	9.63
office_stringsearch1	8.89	7.12	**9.32**	4.78	−0.15	**2.25**	2.65	4.19	2.45	2.78
security_blowfish_d	**28.82**	26.44	23.48	24.90	24.04	**2.15**	**2.15**	3.89	2.25	2.22
security_blowfish_e	**24.16**	23.70	22.21	23.16	22.09	**2.18**	2.21	4.21	2.28	2.29
telecom_adpcm_c	**54.80**	52.93	52.14	53.12	52.26	6.04	6.03	8.73	**4.11**	6.68
telecom_adpcm_d	**17.89**	16.83	16.79	17.03	19.20	**5.10**	5.59	6.53	5.25	5.78
telecom_CRC32	9.37	10.39	7.15	**12.40**	9.07	1.72	1.80	3.28	**1.68**	1.89
telecom_gsm	**29.91**	21.72	28.15	25.81	15.71	**2.35**	2.46	4.44	2.92	2.58
505.mcf_r	0.86	0.43	**1.71**	0.17	1.16	**68.97**	70.66	69.10	69.06	71.04
507.cactuBSSN_r	3.80	1.73	−1.33	**4.31**	0.31	**74.07**	94.2	48.99	76.43	90.21
519.lbm_r	38.17	37.76	**40.05**	35.78	36.94	41.08	95.53	**34.28**	46.49	90.76
520.omnetpp_r	**2.34**	−1.53	2.00	1.93	−1.80	**83.04**	92.04	97.02	91.95	84.00
523.xalancbmk_r	**4.23**	−2.45	−0.98	−8.90	−7.61	**66.09**	131.94	71.61	105.21	170.25
526.blender_r	3.83	1.75	**5.00**	3.06	2.14	**80.67**	96.18	116.22	93.24	231.54
538.imagick_r	**27.80**	13.16	26.76	18.89	15.76	**94.78**	138.30	103.51	96.30	158.74
544.nab_r	**19.76**	9.04	16.03	19.26	9.41	**65.50**	91.25	71.24	75.91	100.06
548.exchange2_r	**5.86**	−22.84	1.79	3.23	−22.92	69.48	111.96	**60.36**	93.03	101.79
557.xz_r	**2.69**	1.41	2.10	1.39	1.53	**29.19**	30.55	30.41	29.60	31.26
平均提升/时间	**14.53**	10.88	12.88	12.14	9.34	**24.33**	33.77	26.72	28.05	39.85

注: OP为OpenTuner; 加粗数据表示最优结果

此外, 在部分基准程序中 (如 consumer_lame、519.lbm_r), R-GA 的最终性能提升低于 BOCA, 这主要源于两者搜索机制的侧重点不同. R-GA 优势在于利用预定义的选项逻辑依赖关系指导搜索方向, 但当某些对程序性能影响较大的选项较为独立且在编译器源码中缺乏明确关系关联时, R-GA 的先验引导效力会有所减弱. 以 consumer_lame 为例, BOCA 在迭代过程中通过概率代理模型捕捉到-ffast-math 与-frounding-math 这两个对性能增益影响显著的选项, 而这两个选项在编译器源码中并无显式的关系体现. 因此, BOCA 凭借其基于贝叶斯优化的搜索策略优势, 能在动态迭代过程中识别出此类较为孤立的重要选项, 从而获得相较于 R-GA 更优的结果. 这表明除基于编译器源码分析外, 基于程序特征识别选项间的隐式关系也是未来 R-GA 优化的重要方向.

4.2　提高编译测试效率

编译选项的关系还可以帮助提高编译测试的效率. 由于 GCC 代码中的部分分支只有在编译选项满足特定要

求时才能到达, 所以如果一直使用编译器提供的默认选项配置进行测试, 可能导致部分分支永远无法被覆盖, 增加隐藏缺陷的风险. 但在不了解编译选项关系的前提下, 遍历选项状态将耗费大量时间, 且由于选项之间存在内部约束关系, 导致很多选项组合效果相同 (例如-fno-inline -finline-functions 和-fno-inline -fno-inline-functions), 实际上无需重复测试. 因此, 理解选项关系可以帮助开发者高效地选择具有代表性的编译选项序列以进行测试.

以指令调度优化为例, GCC 中指令调度优化相关的 Pass 主要为: pass_sched、pass_sched2、pass_live_range_shrinkage 和 pass_sched_fusion 这 4 个, 涉及编译选项包括-fschedule-insns、-fschedule-insns2 等共 22 个. 若想全面探索指令调度优化相关的执行路径, 在完全不理解选项关系的情况下需要测试 2^{22} (约 420 万) 次. 但实际上, 指令调度优化选项对目标机器架构存在依赖关系, 且选项之间有内部约束关系, 若删去不满足优化要求的选项, 再只保留满足控制依赖关系的选项序列, 测试次数便可大大减少, 提高测试效率.

图 4 为指令调度优化涉及的所有 22 个选项, 其中有 2 个选项 (虚线) 由于实验中机器架构不满足优化要求而无效, 其他 20 个选项均为有效选项. 箭头代表选项有效的条件, 例如-fselective-scheduling 当且仅当-fschedule-insns 启用时有效. 虚线框代表框内选项具有相同的生效条件. 如果一个选项同时受几个选项控制, 仅满足一条要求即可有效, 例如, 对于-fsel-sched-pipelining 选项, -fschedule-insns、-fschedule-insn2、-flive-range-shrinkage 这 3 个选项至少打开一个即可生效. 在选项关系指导下, 在迭代生成编译选项序列作为测试样例时, 可以通过考虑选项关系, 减少生成重复选项序列的数量, 例如选择使用-fno-schedule-insns 时, 就不再考虑如何选择-fselective-scheduling 的状态. 通过考虑选项关系, 指令调度优化相关选项的搜索空间从 2^{22} 降低至 183 451 (约 2^{17}), 仅为原来的 4.4%.

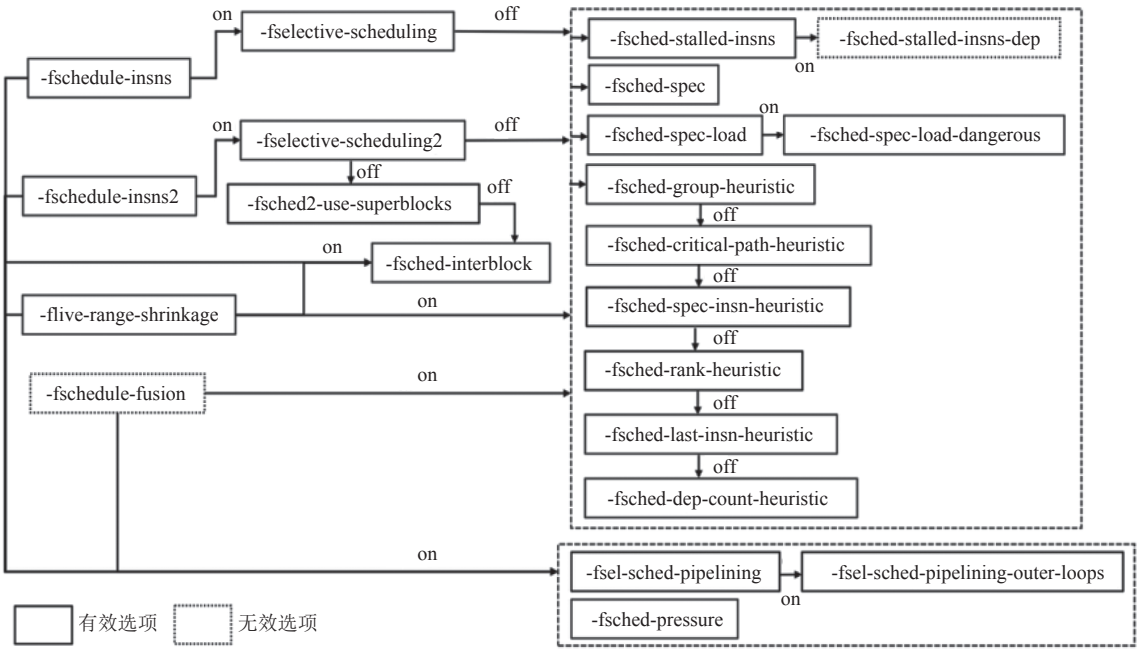


图 4 指令调度选项相关优化选项关系

4.2.1 基于选项关系的选项测试序列生成算法

具体算法如算法 6 所示. 在对目标机器架构和目标程序生成选项序列作为测试样例时, 我们根据选项的约束关系依次遍历选项空间内所有选项, 并按照以下步骤生成每个选项的状态: 首先, 检查选项的外部依赖关系, 如果当前环境下的机器架构或者目标程序特征不满足选项要求, 则直接将该选项的状态设置为关闭 (第 2–4 行). 然后, 遍历当前选项的内部约束关系, 如果已生成的选项序列中的某些选项状态满足当前选项的约束条件, 则直接根据

约束条件生成当前选项状态; 如果当前选项状态不受任何约束条件的影响, 则随机选择该选项状态 (第 6–10 行). 基于选项关系的测试序列生成算法避免了重复生成实际优化效果相同的选项序列, 从而帮助提高编译器测试效率.

算法 6. 基于选项关系的测试序列生成算法.

输入: 选项空间 *Space*, 选项外部依赖关系 O ($O=\{op_1:0,op_2:1,op_3:1,\ldots,op_n:0\}$), 选项内部约束关系 C ($C=\{op_1:\{op_2:[0,1]\},op_2:\{op_3:[1,1],op_5:[0,0]\},\ldots\}$);
输出: 测试样例 *seq*.

```
1. Function Generate_test_sequences(Space,O,G):
2.   for each op ∈ Space do //遍历选项空间内选项
3.     if O[op] == 0 then //若当前外部环境下该选项无效, 则直接关闭
4.       seq[op] = 0
5.     else
6.       for each op_relation ∈ C[op] do //遍历和 op 存在约束关系的 op_relation
7.         if C[op][op_relation][0] == seq[op_relation] then //若已生成序列中 op_relation 状态满足约束条件
8.           seq[op] = C[op][op_relation][1] //直接将选项设置为满足约束关系的状态
9.         else
10.          seq[op] = randint(0,1) //否则随机选择选项状态
11.  return seq
```

4.2.2 实验设置

硬件和软件环境与第 4.1.2 节一致. 为验证基于选项关系的测试序列生成方法在提升测试样例多样性方面的有效性, 我们使用 Csmith 随机生成的 500 个程序作为种子程序. 对每个程序分别使用随机序列生成算法和基于选项关系的测试序列生成算法迭代生成 1 000 个不同的选项序列并进行编译测试. 实验以平均有效测试样例数量 (即 500 个程序中, 每个程序平均生成的不同二进制文件数量) 作为核心评价指标. 对于同一程序, 每个不同的二进制文件均对应编译器一次不同的优化决策或 Pass 执行路径, 从而体现算法在生成样例的多样性方面的差异.

4.2.3 实验结果

图 5 展示了使用随机序列生成方法和基于选项关系的测试序列生成方法生成选项序列编译程序的平均有效测试样例数量随迭代次数的变化情况. 迭代 500 轮和 1 000 轮时, 随机序列生成方法的平均有效测试样例数量分别约为 161 个和 240 个, 基于选项关系的测试序列生成方法的平均有效测试样例数量约为 248 个和 364 个, 有效样例的生成效率分别提升了 54.0% 和 51.7%. 结果表明基于选项关系的测试序列生成方法能够在相同时间开销下生成更多有效的测试样例, 即能触发更多编译器内部的不同优化路径, 从而实现对编译器更充分的测试.

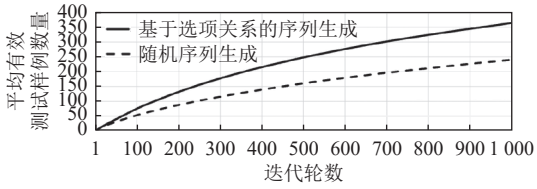


图 5 迭代测试过程中平均有效测试样例数量变化

为了进一步验证基于选项关系的测试序列生成方法的有效性, 我们对其与随机序列生成方法在相同程序上的有效样例生成能力进行了对比分析, 结果如图 6 所示. 横轴表示两种方法迭代 1 000 轮后生成的有效样例数量比值 (基于选项关系的测试序列生成/随机序列方法), 纵轴为累计概率. 结果显示, 除少数结构比较简单的程序的有效样例数量相同 (比值为 1) 外, 绝大部分程序 (97.6%) 的比值均大于 1, 表明基于选项关系的测试序列生成方法能生成

更多的有效样例. 其中, 超过一半 (54%) 的程序比值在 1.5 及以上, 最高可达 1.80. 这表明在相同迭代条件下, 基于选项关系的测试序列生成方法在提升样例多样性与生成效率方面具有显著优势.

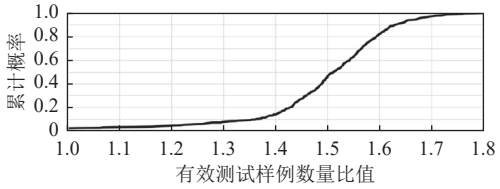


图 6　基于选项关系的测试序列生成方法与随机序列生成方法的有效样例数量比值

4.3 提高开发者的开发效率

由于编译选项关系复杂, 且编译器文档中对编译选项的描述较为有限, 开发者在缺乏对选项关系的全面了解时, 可能会导致对某些选项配置的无效或不合理, 从而无法实现预期的优化目标. 此外, 了解选项之间的关系还可以帮助开发者提升排查故障的效率.

(1) 案例 1: 帮助开发者正确配置选项

在 GitHub 中, 我们发现许多误用选项的真实案例[60−63]. 例如有多位开发者在 Makefile 中使用了-fno-inline -finline-functions 进行编译. 开发者的初衷应该是只开启简单的函数内联优化, 而不考虑间接调用内联、部分内联等高级内联策略, 但-fno-inline 会直接关闭所有类型的内联优化, 随后开启的-finline-functions 实际上没有起到任何优化作用. 这种配置可能导致编译结果并未达到开发者的预期效果. 如果开发者对选项关系有更深入的了解, 就能够避免这种无效配置, 从而更高效、准确地利用编译器选项实现优化目标.

(2) 案例 2: 帮助开发者快速定位故障

在 GCC 源码的 gcc-9.2.0/gcc/testsuite/gfortran.dg/pr46804.f90 测试文件中, 使用如下编译选项序列进行测试: -O -fPIC -fexpensive-optimizations -fgcse -foptimize-register-move -fpeel-loops -fno-tree-loop-optimize, 其中-fno-tree-loop-optimize 禁用了所有基于树结构的循环优化, 但同时开启了-fpeel-loops 进行循环展开优化, 由于-fpeel-loops 是依赖于-ftree-loop-optimize 的子优化, 所以-fpeel-loops 实际在该编译序列中无效. 因此, 如果编译器在使用该序列时出现错误, 错误的触发和-fpeel-loops 无关. 对选项关系的理解可以帮助开发者快速排除无关选项对故障的影响, 从而提高故障定位与修复效率.

5 讨　论

5.1 局限性

本文主要使用静态分析技术结合人工验证定位选项关系, 并定制动态验证框架评估编译器选项的有效性, 分析结论对特定编译器版本存在依赖性. 因此如果更换编译器版本, 必须重新进行人工分析并修改验证框架, 这会带来一定的工作量. 实际上, 不同版本的编译器选项关系较为稳定, 如需更换编译器版本, 只需对已提供的 GCC-15.2.0 版本的选项关系和验证框架做微调即可. 我们在 GCC-13.2.0 和 GCC-11.2.0 上分别比较了编译选项相对于 GCC-15.2.0 的差异, 结果如后文表 8 所示. 可以看到 3 个版本的编译选项差异较小, 只有个别选项存在差异, 具体差异信息在第 6 节链接中提供.

5.2 可扩展性

本文仅对 GCC 编译选项进行分析, 所讨论的选项关系主要适用于指导 GCC 编译优化/测试相关技术, 难以直接应用于 LLVM 编译器. 这是因为尽管 GCC 和 LLVM 都使用 Pass 组织编译过程, 但两者在 Pass 和选项的关系上存在显著差异, 具体差别如表 9 所示. 在 GCC 中, 优化 Pass 的执行顺序是由编译器底层架构预定义的固定序列, 编译选项只能控制部分 Pass 的开关以及部分执行逻辑. 尽管在命令行层面, GCC 遵循“后项优先”原则处理参数, 但一旦各选项的启用状态确定, 用户便无法干预其内部 Pass 的执行先后顺序. 因此, 在规范使用且确定选项集合

的前提下, 开发者只需考虑选项的开关, 无需考虑其出现的顺序和次数. 此外, GCC 中的选项和 Pass 关系为多对多耦合, 单个选项可能影响多个 Pass, 单个 Pass 也可能受到多个选项的影响. 而 LLVM 中, Pass 执行顺序完全由开发者决定, 每个选项直接控制一个 Pass 的开关, 选项和 Pass 一一对应, 具有更高的灵活性. 开发者需要自行决定每个 Pass 的顺序和执行次数. 由于各 Pass 之间的关系是平行的, 各个 Pass (选项) 之间没有显式的控制依赖关系, 难以直接将 GCC 的选项内部约束关系模型直接应用于 LLVM.

表 8　不同版本编译选项及关系变化

GCC版本	新增选项	删减选项	内部约束关系对	运行环境敏感选项数量	程序特征敏感选项数量
15.2.0	—	—	68	51	52
13.2.0	1	4	67	51	53
11.2.0	1	8	66	51	53

表 9　GCC 与 LLVM 的 Pass 组织机制差异

维度	GCC	LLVM
Pass调度方式	预定义, 不可更改	动态选择, 可更改
选项-Pass映射	多对多耦合	一对一绑定
选项使用	只考虑选项开关	考虑选项开关/顺序/出现次数

尽管 LLVM 和 GCC 的 Pass 管理机制不同, 但选项对外部环境 (即程序特征、硬件架构) 的敏感性仍然存在, 本文提出的识别外部环境依赖关系选项技术可以通过合理迁移应用于 LLVM.

6 总　结

本文系统分析了编译选项之间的关系, 基于 GCC 源码结合静态和动态分析技术, 揭示了影响编译器选项作用效果的两类主要关系: 内部约束关系和外部环境依赖关系. 其中内部约束关系包括控制、启用和禁用关系, 外部依赖关系包括对机器架构和程序特征的依赖. 通过选项关系的分析, 本文为开发者提供了更为科学、高效的编译选项配置方案, 帮助开发者高效合理使用编译选项, 达到预期使用效果. 此外, 选项关系的分析还可进一步为选项自动调优和编译器测试领域提供指导. 在选项自动调优中, 引导算法在有约束关系的选项组中进行重点搜索, 提高搜索效率; 在编译器测试中, 通过减少相同编译效果的冗余测试选项序列, 提高编译器测试效率. 我们将选项关系分析结果开源至 https://github.com/LanduBing/OptionRelationship, 其中包括选项有效性的动态验证框架及案例分析中涉及的选项自动调优与编译器测试模块, 促进未来开发者及相关领域研究者更合理、高效地使用编译选项.

References

[1] GCC. GCC, the GNU compiler collection. 2025. http://gcc.gnu.org

[2] Zhu MX, Sun ZY, Hao D. PDCAT: Preference-driven compiler auto-tuning. Proc. of the ACM on Software Engineering, 2025, 2(FSE): 847–867. [doi: 10.1145/3715756]

[3] Zhu MX, Hao D. Compiler auto-tuning via critical flag selection. In: Proc. of the 38th IEEE/ACM Int'l Conf. on Automated Software Engineering. Luxembourg: IEEE, 2023. 1000–1011. [doi: 10.1109/ase56229.2023.00209]

[4] Ashouri AH, Killian W, Cavazos J, Palermo G, Silvano C. A survey on compiler autotuning using machine learning. ACM Computing Surveys, 2019, 51(5): 96. [doi: 10.1145/3197978]

[5] Hoste K, Eeckhout L. Cole: Compiler optimization level exploration. In: Proc. of the 6th Annual IEEE/ACM Int'l Symp. on Code Generation and Optimization. Boston: IEEE, 2008. 165–174. [doi: 10.1145/1356058.1356080]

[6] Chi HY, Chen CB. Survey on automatic tuning of compilers by machine learning. Computer Science, 2022, 49(1): 241–251 (in Chinese with English abstract). [doi: 10.11896/jsjkx.210100113]

[7] Gao GJ, Ren ZL, Zhang JX, Li XC, Jiang H. Selection of compiler-optimization sequences. SCIENTIA SINICA Informationis, 2019, 49(10): 1267–1282 (in Chinese with English abstract). [doi: 10.1360/N112019-00050]

[8] Chen JJ, Xu NX, Chen PQ, Zhang HY. Efficient compiler autotuning via Bayesian optimization. In: Proc. of the 43rd IEEE/ACM Int'l Conf. on Software Engineering. Madrid: IEEE, 2021. 1198–1209. [doi: 10.1109/icse43902.2021.00110]

[9] Park S, Latifi S, Park Y, Behroozi A, Jeon B, Mahlke S. SRTuner: Effective compiler optimization customization by exposing synergistic relations. In: Proc. of the 2022 IEEE/ACM Int'l Symp. on Code Generation and Optimization. Seoul: IEEE, 2022. 118–130. [doi: 10.1109/cgo53902.2022.9741263]

[10] Chen JJ, Patra J, Pradel M, Xiong YF, Zhang HY, Hao D, Zhang L. A survey of compiler testing. ACM Computing Surveys, 2021, 53(1): 4. [doi: 10.1145/3363562]

[11] Jiang H, Zhou ZD, Ren ZL, Zhang JX, Li XC. CTOS: Compiler testing for optimization sequences of LLVM. IEEE Trans. on Software Engineering, 2022, 48(7): 2339–2358. [doi: 10.1109/tse.2021.3058671]

[12] Chen JJ, Suo CY. Boosting compiler testing via compiler optimization exploration. ACM Trans. on Software Engineering and Methodology, 2022, 31(4): 72. [doi: 10.1145/3508362]

[13] Yang XJ, Chen Y, Eide E, Regehr J. Finding and understanding bugs in C compilers. In: Proc. of the 32nd ACM SIGPLAN Conf. on Programming Language Design and Implementation. San Jose: ACM, 2011. 283–294. [doi: 10.1145/1993498.1993532]

[14] Lidbury C, Lascu A, Chong N, Donaldson AF. Many-core compiler fuzzing. ACM SIGPLAN Notices, 2015, 50(6): 65–76. [doi: 10.1145/2813885.2737986]

[15] Cummins C, Petoumenos P, Murray A, Leather H. Compiler fuzzing through deep learning. In: Proc. of the 27th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Amsterdam: ACM, 2018. 95–105. [doi: 10.1145/3213846.3213848]

[16] Zhang QR, Sun CN, Su ZD. Skeletal program enumeration for rigorous compiler testing. In: Proc. of the 38th ACM SIGPLAN Conf. on Programming Language Design and Implementation. Barcelona: ACM, 2017. 347–361. [doi: 10.1145/3062341.3062379]

[17] Le V, Afshari M, Su ZD. Compiler validation via equivalence modulo inputs. ACM Sigplan Notices, 2014, 49(6): 216–226. [doi: 10.1145/2666356.2594334]

[18] Even-Mendoza K, Sharma A, Donaldson AF, Cadar C. GrayC: Greybox fuzzing of compilers and analysers for C. In: Proc. of the 32nd ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Seattle: ACM, 2023. 1219–1231. [doi: 10.1145/3597926.3598130]

[19] Wang JJ, Chen BH, Wei L, Liu Y. Superion: Grammar-aware greybox fuzzing. In: Proc. of the 41st IEEE/ACM Int'l Conf. on Software Engineering. Montreal: IEEE, 2019. 724–735. [doi: 10.1109/icse.2019.00081]

[20] Ou XF, Jiang YY, Xu C. Semantic aware greybox compiler fuzz testing. Ruan Jian Xue Bao/Journal of Software, 2025, 36(7): 2947–2963 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/7333.htm [doi: 10.13328/j.cnki.jos.007333]

[21] Liu JW, Wei YX, Yang S, Deng YL, Zhang LM. Coverage-guided tensor compiler fuzzing with joint ir-pass mutation. Proc. of the ACM on Programming Languages, 2022, 6(OOPSLA1): 73. [doi: 10.1145/3527317]

[22] Wu MY, Lu MH, Cui HM, Chen JJ, Zhang YQ, Zhang LM. JITfuzz: Coverage-guided fuzzing for JVM just-in-time compilers. In: Proc. of the 45th IEEE/ACM Int'l Conf. on Software Engineering. Melbourne: IEEE, 2023. 56–68. [doi: 10.1109/ICSE48619.2023.00017]

[23] McKeeman WM. Differential testing for software. Digital Technical Journal, 1998, 10(1): 100–107.

[24] Chen JJ, Hu WX, Hao D, Xiong YF, Zhang HY, Zhang L, Xie B. An empirical comparison of compiler testing techniques. In: Proc. of the 38th Int'l Conf. on Software Engineering. Austin: IEEE, 2016. 180–190. [doi: 10.1145/2884781.2884878]

[25] Le V, Sun CN, Su ZD. Finding deep compiler bugs via guided stochastic program mutation. ACM SIGPLAN Notices, 2015, 50(10): 386–399. [doi: 10.1145/2858965.2814319]

[26] Sun CN, Le V, Su ZD. Finding compiler bugs via live code mutation. In: Proc. of the 2016 ACM SIGPLAN Int'l Conf. on Object-Oriented Programming, Systems, Languages, and Applications. Amsterdam: ACM, 2016. 849–863. [doi: 10.1145/2983990.2984038]

[27] Gu QH. LLM-based code generation method for golang compiler testing. In: Proc. of the 31st ACM Joint European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. San Francisco: ACM, 2023. 2201–2203. [doi: 10.1145/3611643.3617850]

[28] Yu XZ, Wong WK, Wang S. EMI testing of large language model (LLM) compilers. In: Proc. of the 35th IEEE Int'l Symp. on Software Reliability Engineering Workshops. Tsukuba: IEEE, 2024. 187–190. [doi: 10.1109/ISSREW63542.2024.00076]

[29] Yang CY, Deng YL, Lu RY, Yao JY, Liu JW, Jabbarvand R, Zhang LM. WhiteFox: White-box compiler fuzzing empowered by large language models. Proc. of the ACM on Programming Languages, 2024, 8(OOPSLA2): 296. [doi: 10.1145/3689736]

[30] Gao HY, Yang YB, Sun ML, Wu JC, Zhou YM, Xu BW. Clozemaster: Fuzzing Rust compiler by harnessingLLMs for infilling masked real programs. In: Proc. of the 47th IEEE/ACM Int'l Conf. on Software Engineering. Ontario: IEEE, 2025. 1422–1435. [doi: 10.1109/icse55347.2025.00175]

[31] Ou XF, Li C, Jiang YY, Xu C. The mutators reloaded: Fuzzing compilers with large language model generated mutation operators. In: Proc. of the 29th ACM Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. La Jolla: ACM, 2024. 298–312. [doi: 10.1145/3622781.3674171]

[32] Chen Y, Fang SD, Huang YJ, Eeckhout L, Fursin G, Temam O, Wu CY. Deconstructing iterative optimization. ACM Trans. on Architecture and Code Optimization, 2012, 9(3): 21. [doi: 10.1145/2355585.2355594]

[33] Ashouri AH, Bignoli A, Palermo G, Silvano C, Kulkarni S, Cavazos J. MiCOMP: Mitigating the compiler phase-ordering problem using optimization sub-sequences and machine learning. ACM Trans. on Architecture and Code Optimization, 2017, 14(3): 29. [doi: 10.1145/3124452]

[34] Liang YW, Stone K, Shameli A, Cummins C, Elhoushi M, Guo JD, Steiner B, Yang XM, Xie PT, Leather H, Tian YD. Learning compiler pass orders using coreset and normalized value prediction. In: Proc. of the 40th Int'l Conf. on Machine Learning. Honolulu: PMLR, 2023. 855.

[35] Liu HZ, Luo J, Li Y, Wu ZH. Iterative compilation optimization based on metric learning and collaborative filtering. ACM Trans. on Architecture and Code Optimization, 2022, 19(1): 2. [doi: 10.1145/3480250]

[36] Pan HL, Wei YY, Xing MJ, Wu YJ, Zhao C. Towards efficient compiler auto-tuning: Leveraging synergistic search spaces. In: Proc. of the 23rd ACM/IEEE Int'l Symp. on Code Generation and Optimization. Las Vegas: ACM, 2025. 614–627. [doi: 10.1145/3696443.3708961]

[37] Bodin F, Kisuki T, Knijnenburg P, O'Boyle M, Rohou E. Iterative compilation in a non-linear optimisation space. In: Proc. of the 1998 Workshop on Profile and Feedback-directed Compilation. Paris: HAL, 1998.

[38] Garciarena U, Santana R. Evolutionary optimization of compiler flag selection by learning and exploiting flags interactions. In: Proc. of the 2016 on Genetic and Evolutionary Computation Conf. Companion, Denver: ACM, 2016. 1159–1166. [doi: 10.1145/2908961.2931696]

[39] Ansel J, Kamil S, Veeramachaneni K, Ragan-Kelley J, Bosboom J, O'Reilly UM, Amarasinghe S. OpenTuner: An extensible framework for program autotuning. In: Proc. of the 23rd Int'l Conf. on Parallel Architectures and Compilation. Edmonton: IEEE, 2014. 303–316. [doi: 10.1145/2628071.2628092]

[40] Gao BY, Yao MY, Wang ZM, Liu D, Li D, Chen XQ, Guo Y. Grouptuner: Efficient group-aware compiler auto-tuning. In: Proc. of the 26th ACM SIGPLAN/SIGBED Int'l Conf. on Languages, Compilers, and Tools for Embedded Systems. Seoul: ACM, 2025. 122–133. [doi: 10.1145/3735452.3735530]

[41] Fursin G, Kashnikov Y, Memon AW, Chamski Z, Temam O, Namolaru M, Yom-Tov E, Mendelson B, Zaks A, Courtois E, Bodin F, Barnard P, Ashton E, Bonilla E, Thomson J, Williams CKI, O'boyle M. Milepost GCC: Machine learning enabled self-tuning compiler. Int'l Journal of Parallel Programming, 2011, 39(3): 296–327. [doi: 10.1007/s10766-010-0161-2]

[42] Zhu MX, Hao D, Chen JJ. Compiler autotuning through multiple-phase learning. ACM Trans. on Software Engineering and Methodology, 2024, 33(4): 100. [doi: 10.1145/3640330]

[43] Wang DS, Tan DP, Liu L. Particle swarm optimization algorithm: An overview. Soft Computing, 2018, 22(2): 387–408. [doi: 10.1007/s00500-016-2474-6]

[44] Roy RB, Patel T, Gadepally V, Tiwari D. Bliss: Auto-tuning complex applications using a pool of diverse lightweight learning models. In: Proc. of the 42nd ACM SIGPLAN Int'l Conf. on Programming Language Design and Implementation. ACM, 2021. 1280–1295. [doi: 10.1145/3453483.3454109]

[45] Hellsten EO, Souza A, Lenfers J, Lacouture R, Hsu O, Ejjeh A, Kjolstad F, Steuwer M, Olukotun K, Nardi L. BaCO: A fast and portable Bayesian compiler optimization framework. In: Proc. of the 28th ACM Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. Vancouver: ACM, 2023. 19–42. [doi: 10.1145/3623278.3624770]

[46] Cummins C, Petoumenos P, Wang Z, Leather H. End-to-end deep learning of optimization heuristics. In: Proc. of the 26th Int'l Conf. on Parallel Architectures and Compilation Techniques. Portland: IEEE, 2017. 219–232. [doi: 10.1109/pact.2017.24]

[47] Mendis C, Renda A, Amarasinghe SP, Carbin M. Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks. In: Proc. of the 36th Int'l Conf. on Machine Learning. Long Beach: PMLR, 2019. 4505–4515.

[48] Zhang MJ, Li MH, Wang C, Li MQ. DynaTune: Dynamic tensor program optimization in deep neural network compilation. In: Proc. of the 9th Int'l Conf. on Learning Representations. OpenReview.net, 2021.

[49] Mahajan A, Teneketzis D. Multi-armed bandit problems. In: Hero AO, Castañón DA, Cochran D, Kastella K, eds. Foundations and Applications of Sensor Management. New York: Springer, 2008. 121–151. [doi: 10.1007/978-0-387-49819-5_6]

[50] Haj-Ali A, Ahmed NK, Willke T, Shao YS, Asanovic K, Stoica I. NeuroVectorizer: End-to-end vectorization with deep reinforcement learning. In: Proc. of the 18th ACM/IEEE Int'l Symp. on Code Generation and Optimization. San Diego: ACM, 2020. 242–255. [doi: 10.1145/3368826.3377928]

[51] Haj-Ali A, Huang QJ, Moses WS, Xiang J, Asanovic K, Wawrzynek J, Stoica I. AutoPhase: Juggling HLS phase orderings in random forests with deep reinforcement learning. In: Proc. of the 3rd Conf. on Machine Learning and Systems. Austin: MLsys, 2020. 70–81.

[52] Cummins C, Wasti B, Guo JD, Cui B, Ansel J, Gomez S, Jain S, Liu J, Teytaud O, Steiner B, Tian YD, Leather H. CompilerGym: Robust, performant compiler optimization environments for AI research. In: Proc. of the 2022 IEEE/ACM Int'l Symp. on Code Generation and Optimization. Seoul: IEEE, 2022. 92–105. [doi: 10.1109/cgo53902.2022.9741258]

[53] Cummins C, Seeker V, Grubisic D, Roziere B, Gehring J, Synnaeve G, Leather H. LLM compiler: Foundation language models for compiler optimization. In: Proc. of the 34th ACM SIGPLAN Int'l Conf. on Compiler Construction. Las Vegas: ACM, 2025. 141–153. [doi: 10.1145/3708493.3712691]

[54] Pan HL, Lin HY, Luo HR, Liu Y, Yao KC, Zhang LB, Xing MJ, Wu YJ. Compiler-R1: Towards agentic compiler auto-tuning with reinforcement learning. arXiv:2506.15701, 2025.

[55] CodeQL documentation. 2024. https://codeql.github.com/docs

[56] Optimize Options—Using the GNU Compiler Collection (GCC). 2025. https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

[57] SOURCEFORGE. cBenchmark Files. 2010. https://sourceforge.net/projects/cbenchmark/files/cBench/V1.1/

[58] Bucek J, Lange KD, Kistowski JV. SPEC CPU2017: Next-generation compute benchmark. In: Proc. of the 2018 ACM/SPEC Int'l Conf. on Performance Engineering. Berlin: ACM, 2018. 41–42. [doi: 10.1145/3185768.3185771]

[59] Perf: Linux profiling with performance counters. 2024. https://perf.wiki.kernel.org/index.php/Main_Page

[60] freebsd-ports. 2015. https://github.com/renchap/freebsd-ports/blob/9c7ccad22b5b3056cc65d1730c8d4939ac7e36b8/audio/calf/Makefile#L44

[61] DebugSettings.cmake. 2024. https://github.com/Fracture17/ProjectMCodes/blob/52b99a7e7b55e31feb88154b5ffe7e63fb3a5410/CMake/DebugSettings.cmake#L32

[62] CMakeLists.txt. 2018. https://github.com/vovoid/vsxu/blob/master/CMakeLists.txt

[63] irz_ruh2b_defconfig. 2017. https://github.com/p1ne/irz-devel/blob/7d97287195b12e7fe0c69cd4bc9ee8540c89e698/buildroot/configs/irz_ruh2b_defconfig#L243

附中文参考文献

[6] 池昊宇, 陈长波. 基于机器学习的编译器自动调优综述. 计算机科学, 2022, 49(1): 241–251. [doi: 10.11896/jsjkx.210100113]

[7] 高国军, 任志磊, 张静宣, 李晓晨, 江贺. 编译优化序列选择研究进展. 中国科学: 信息科学, 2019, 49(10): 1267–1282. [doi: 10.1360/N112019-00050]

[20] 欧先飞, 蒋炎岩, 许畅. 语义可感知的灰盒编译器模糊测试. 软件学报, 2025, 36(7): 2947–2963. http://www.jos.org.cn/1000-9825/7333.htm [doi: 10.13328/j.cnki.jos.007333]

作者简介

高秉玉, 博士生, 主要研究领域为编译优化, 编译器测试.

姚梦雨, 博士生, 主要研究领域为机器学习系统安全, 大模型隐私保护.

王子铭, 博士生, 主要研究领域为智能体系统安全, 软件安全性.

李锭, 博士, 助理教授, 博士生导师, CCF 专业会员, 主要研究领域为系统安全, 软件工程.

陈向群, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为操作系统, 软件工程, 移动计算.

郭耀, 博士, 教授, 博士生导师, CCF 高级会员, 主要研究领域为操作系统, 软件工程, 移动计算.