利用大语言模型泛化窥孔优化

CHUNHAO LIAO, 滑铁卢大学，加拿大HONGXU
XU, 滑铁卢大学，加拿大XINTONG ZHOU, 滑铁卢大学，
加拿大ZHENYANG XU, 滑铁卢大学，加拿大
CHENGNIAN SUN,滑铁卢大学，加拿大

arXiv:2603.18477v1 [cs.PL] 19 Mar 2026

窥孔优化是现代优化编译器的核心组成部分。它将特定指令重写为语义等价但效率更高的形式。在实践中，创建新的窥孔优化通常始于一个具体的优化实例，并要求将其提升为更通用的重写规则，以匹配更广泛的指令模式。这一泛化步骤对优化效果至关重要，但也颇具难度：生成既正确又足够通用的规则通常需要大量人工工作和领域专业知识。现有方法（如 Hydra）尝试利用程序综合自动完成这一任务，但其泛化能力往往受到搜索空间爆炸、泛化不足以及对多样化指令领域支持受限等问题的限制。

我们提出了 LPG，即大语言模型辅助的窥孔优化泛化框架。该框架利用大语言模型（LLM）泛化窥孔优化。LPG 的设计源于这一观察：大语言模型擅长语义抽象和探索性推理，而形式化分析对于确保生成的规则既可靠又具有收益性至关重要。基于这一观察，LPG 采用闭环工作流，将大语言模型驱动的符号常量泛化、结构泛化、约束放宽以及位宽/精度泛化，与语法验证、语义验证和收益性检查的反馈相结合。

我们在来自 LLVM 生态系统的真实世界窥孔优化问题上评估 LPG。总体而言，LPG 成功泛化了 102 项优化中的 90 项。在与 Hydra 可直接比较的整数子集上，LPG 泛化了 81 项优化中的 74 项，而 Hydra 泛化了 35 项。此外，在 22 个案例中，LPG 生成的规则严格而言更加通用，而 Hydra 仅有 2 个案例。这些结果表明，LPG 在广泛的指令领域中泛化真实世界的窥孔优化方面更为有效。

附加关键词与短语：编译器、编译器优化、编译器优化泛化、大语言模型

1 引言

窥孔优化是现代编译流水线的基石。这项技术在指令滑动窗口（即“窥孔”）上运行，识别特定的指令模式，并将其替换为语义等价且效率更高的方案 [17, 36, 41]。窥孔优化的重要性可从其在工业编译器中的庞大规模得到印证；例如，LLVM 仅在其 InstCombine 阶段中，就为窥孔优化投入了超过 50,000 行 C++ 代码 [45]。然而，先前研究指出，窥孔优化器是编译器中最容易出错的组件之一 [37, 43, 49, 54]。LLVM 的窥孔优化器便是一个例证，它占已报告错误编译的相当大比例 [27, 28, 55]。因此，在现代编译器系统中，开发有效且可靠的窥孔优化器仍至关重要，也极具挑战性。

通常，新增一项窥孔优化涉及三个步骤 [43]：1 识别出次优的指令序列并推导出优化后的替代序列；2 将这一

作者联系信息：CHUNHAO LIAO，滑铁卢大学，加拿大，chunhao.liao@uwaterloo.ca；Hongxu Xu，滑铁卢大学，加拿大，hongxu.xu@uwaterloo.ca；Xintong Zhou，滑铁卢大学，加拿大，x27zhou@ uwaterloo.ca；ZhenyangXu，滑铁卢大学，加拿大，zhenyang.xu@uwaterloo.ca；ChengnianSun，滑铁卢大学，加拿大，cnsun@uwaterloo.ca。

具体变换泛化为广泛适用的重写规则；以及3 在编译器代码库中实现该广义化规则。这一泛化阶段（即，步骤2）至关重要；它将具体的优化实例抽象为规则，使编译器能够处理超越孤立具体案例的多种指令序列。

编译器开发人员通常会手动泛化优化，使其适用范围超出最初的示例 [21, 43, 51]。然而，这种手动方法存在很大障碍；具体而言，随着优化逻辑日益复杂，推导通用规则也变得愈发困难 [1, 2, 6–10]。在一份关于遗漏的优化的 LLVM 问题报告中 [7], ，一位维护者明确承认：“我不知道如何泛化这个特定的折叠优化。”该问题在分配处理人员后仍长期未获解决，这一事实凸显了手动进行窥孔泛化并非易事。此外，手动方法缺乏系统严谨性，可能导致可靠性风险。这个过程不仅容易出错，而且要求很高，以至于有效的优化常常因手动泛化的负担过重而被放弃 [21, 43, 51]。

此前的研究（如 Hydra） [43] 尝试通过枚举式综合自动泛化优化。尽管这些方法会系统地探索候选泛化方案，但它们存在三个主要局限。首先，搜索空间的组合式增长经常导致综合失败（例如，超时），复杂优化尤其如此。其次，所得优化往往泛化不足，也就是说，其泛化程度低于理论上可达到的水平。例如，Hydra 采用贪心策略，返回找到的第一个有效泛化方案，但它未必是搜索空间中最广义的方案。第三，这些技术通常仅适用于狭窄的指令领域，主要是整数算术；对于涉及浮点或内存指令的优化，则难以进行泛化。§ 2.2 将更详细地讨论 Hydra 的局限性。

LPG。 为弥合这些差距，我们提出 LPG，这是一种基于大语言模型的窥孔优化生成流水线。LPG 集成了形式化验证，以确保广义化重写规则的正确性，并采用性能成本建模来保证其收益性（§ 3.2）。具体而言，LPG 将泛化过程组织为四个阶段：符号常量泛化 （§ 3.3）、结构泛化 （§ 3.4）、约束放宽 （§ 3.5）以及 位宽/精度泛化 （§ 3.6）。LPG 在这一多阶段过程中始终运用大语言模型，直接应对了以往基于综合的方法所面临的局限。首先，LPG 利用大语言模型的推理能力推断程序意图，使其能够探索庞大的变换空间，同时避免搜索空间爆炸。其次，得益于大语言模型提供的语义理解，LPG 不受贪心策略的限制，因此更有可能推导出更通用的重写规则。第三，LPG 不将泛化局限于仅整数的窥孔优化，而是利用大语言模型的生成灵活性，将其范围扩展到更多样、更复杂的优化领域，包括浮点和内存指令。

我们全面评估了 LPG 在 LLVM 生态系统中泛化窥孔优化的能力。为便于与 Hydra 直接比较，我们首先在 Hy-draDataset 上评估了 LPG。该数据集包含原始 Hydra 研究中使用的 40 个基准测试 [43]。在 Hy-draDataset 上，LPG 成功泛化了 40 项优化中的 36 项，而 Hydra 泛化了 23 项。此外，在两个系统均成功的案例中，LPG 在 14 个案例中生成了严格更广义的重写规则，而 Hydra 仅在 1 个案例中做到这一点。为减轻评估基于大语言模型的方法时可能出现的数据泄漏，我们进一步整理了 LPGDataset，其中包含 62 项从 LLVM GitHub 问题中挖掘的真实世界遗漏的优化；这些问题报告于受评估模型的知识截止日期之后。在 LPGDataset 上，LPG 成功泛化了 41 项整数优化中的 38 项、16 项浮点优化中的 12 项，以及 5 项内存优化中的 4 项；而 Hydra 仅泛化了 41 项整数优化中的 12 项。在两个系统均成功的案例中

，LPG 在 8 个案例中生成了严格更广义的重写规则，而 Hydra 仅在 1 个案例中做到这一点。此外，我们评估了 LPG 中所提出的泛化策略的有效性。结果凸显了这些不同策略在提升 LPG 泛化能力方面的关键作用。

贡献。 我们的主要贡献如下:

• 我们首次将大语言模型用于窥孔优化泛化。通过利用大语言模型的语义推理能力，我们的方法克服了传统基于程序综合的方法固有的局限性。

• 我们设计并实现了 LPG，这是一条专用于窥孔优化泛化的流水线。LPG 将四种新颖的泛化策略与形式化验证和反馈引导的细化相结合，以确保生成的优化既正确又具有广泛的适用性。

• 我们使用 LLVM IR 中的现实世界优化对 LPG 进行了全面评估，证明它能够在多种指令领域中泛化优化。LPG 总共成功泛化了 102 个案例中的 90 个。在与 Hydra 可比较的整数子集中，LPG 泛化了 81 个案例中的 74 个，而 Hydra 为 35 个；此外，LPG 在 22 个案例中生成了泛化程度严格更高的规则，而 Hydra 仅在 2 个案例中做到这一点。我们还研究了 LPG 流水线中各项泛化策略的有效性，结果突显了这些策略在提升整体泛化效果方面的重要作用。综上，这些结果彰显了 LPG 相较于当前最先进方法的优势，并确立了它作为一种适用范围广、行之有效的编译器开发工具的地位。

2 背景

我们基于 LLVM 构建了 LPG，以泛化 LLVM IR 中的窥孔优化 [27]。LLVM 是一种模块化、开源的编译器基础设施，采用低层级、基于静态单赋值形式且独立于平台的 IR，旨在支持多种源语言和可复用的工具链组件。我们选择 LLVM，是因为它拥有丰富的工具生态系统，并在工业界和学术界得到广泛采用。虽然当前实现专门面向 LLVM IR，但 LPG 的基本原理仍适用于其他编译器框架。以下内容将介绍 LLVM 中窥孔优化、其泛化以及形式化验证的必要背景。

2.1 窥孔优化

窥孔优化是一种成熟的编译技术，用于执行局部代码变换。优化器通过检查一个小型滑动指令窗口，将特定指令序列替换为语义等价且效率更高的方案。形式上，窥孔优化由以下三元组刻画 [22, 41, 43]:

$$PRE \vDash LHS \Rightarrow RHS$$

其中，左侧 （left-hand side）表示目标指令序列，右侧 （right-hand side）是其经过优化的替代序列，而 部分冗余消除 是定义重写有效性的、关于左侧 和右侧 中操作数的逻辑谓词。形式上，该判断表明，只要部分冗余消除 成立，就可以用右侧 替换左侧 ，且保持语义。如果未明确指定部分冗余消除 ，则部分冗余消除 = 为真，这意味着该重写是无条件的，且左侧和右侧中的操作数不受任何约束。

一个经典的窥孔优化示例是强度削弱 [17], : 用计算开销更低的左移替代乘以 2 的幂:

$$\text{PowerOfTwo}(C) \vDash x \times C \Rightarrow x \ll \log_2(C)$$

之所以优先采用这种替换，是因为按位移位通常比乘法和除法带来更低的 CPU 延迟和能耗。

在 LLVM 中，窥孔优化主要通过 InstCombine 实现。InstCombine 是一个庞大的优化流程，使用 C++ 编写，包含超过 50,000 行手工编写的模式匹配规则，用于识别并变换特定的指令序列 [45]。

2.2 窥孔优化泛化

窥孔优化泛化是一个系统化过程，旨在将具体的优化实例抽象为更通用的变换规则，使其在更广泛的程序类别中仍然有效。初始优化通常从特定代码片段中发现，而泛化的目标是在不损害语义正确性的前提下扩大其适用域。如果某种泛化既能在原始优化处理的所有情形下保持正确性，又能在原始适用域之外的其他情形中实现有益的重写，则可认为该泛化是成功的。形式上，给定两种成功的泛化，如果一种泛化的适用域严格包含另一种的适用域，则前者更具一般性。

为说明窥孔优化泛化这一概念，请看针对某个具体窥孔优化的泛化：

$$\text{true} \vDash ((x \oplus 173)\ \&\ 94) \oplus 57 \Rightarrow (x\ \&\ 94) \oplus 53$$

其中，$\oplus$ 表示异或（异或），& 表示按位与（与）。通过人工检查很难对该实例进行泛化。然而，泛化框架或许能够推断常量之间的潜在关系，并合成广义化重写

$$C_4 = (C_1\ \&\ C_2) \oplus C_3 \vDash ((x \oplus C_1)\ \&\ C_2) \oplus C_3 \Rightarrow (x\ \&\ C_2) \oplus C_4$$

该规则通过对异或应用按位与的分配律推导而来：$(A \oplus B)\ \&\ M = (A \&M)\ \oplus (B \&\ M)$。与仅适用于单个具体常量实例的原始重写不同，广义规则适用于所有满足前置条件 $C_4 = (C_1 \&\ C_2) \oplus C_3$ 的实例。通过严格扩大优化适用的程序范围，窥孔优化泛化对于在各种代码模式中最大限度地提升编译器性能至关重要。**手动泛化。** 传统实现要求开发人员手动泛化每一种具体的窥孔优化。然而，手工处理面临两项挑战。首先，实证分析表明，手动泛化变换规则是窥孔优化中正确性问题的主要来源之一 [43]。其次，开发人员未必总能从特定优化实例中推导出可靠且足够通用的规则，因此可能无法发现有效的优化机会。为说明第二项挑战，我们重新审视 § 1 中介绍的 LLVM 一则遗漏窥孔优化的报告 [7]，其中维护者明确指出，难以对这一折叠进行泛化。对应的*左侧*和*右侧* LLVM IR 代码片段如图 1 所示。泛化这一窥孔优化之所以困难，主要是因为其有效性取决于多个组成部分之间的相互作用。具体而言，要将算术取负、掩码操作、跨位宽规范化（i32 到 i64）、标志敏感行为（nneg、nsw）以及 LLVM poison 语义同时纳入推理，并将其整合到一条可复用的规则中。值得注意的是，为协助 LLVM 开发者实现这一窥孔优化，我们在问题讨论串中发布了 LPG 对该窥孔优化的泛化结果，而 LLVM 维护者明确认可了其正确性和通用性 [24]。

```
定义 i64 @左侧(i32 %arg0) {
  %1 = 减 i32 0,  % arg0
  %2 = 与 i32 %1 , 63
  %3 = 零扩展 nneg i32 %2 转为 i64
  %4 = sub nsw i64 0, %3
  %5 = 逻辑右移 i64 %4, 8
  %6 = 或 i64 %5 , %4
  返回 i64 %6
}
```

(a) 左侧 **(变换前)**

```
定义 i64 @右侧(i32 % 参数0){
  %1 = 与 i32 %参数0, 63
  %2 = icmp ne i32 %1, 0
  %3 = 符号扩展 i1 %2 为 i64
  返回 i64 %3
}
```

(b) 右侧 **（变换后）**

图 1. LLVM 中一个未能实现的窥孔优化示例。在 LLVM IR 中，“sub”是减法指令，“and”是按位与，“zext”是零扩展，“lshr”是逻辑右移，“or”是按位或，“icmp ne”是整数不等比较，“sext”是符号扩展。像“nneg”和“nsw”这样的标志表示该指令不允许某些行为，例如负溢出和有符号溢出；如果发生这些情况，可能会导致未定义行为。

	原始	**广义化**
PRE	true	二的幂$(C_1 + 1)$ $\wedge\ 0 \leq_u V \leq_u C_1$ $\wedge$ 人口计数(C_1) $\leq_u C_2 \leq_u W$ − 人口计数(C_1)
LHS	$((-\text{零扩展}(-x)\ \&\ 63)) \gg_u 8 \mid -\text{零扩展}(-x \&\ 63)$	$(-V) \gg_u C_2 \mid (-V)$
RHS	符号扩展$(x \&\ 63 \neq 0)$	符号扩展$(V \neq 0)$

注：PopCount(C_1) 表示 C_1中 1 位的数量。符号扩展和零扩展分别是 sext 和 zext 操作。对于带下标的运算符，例如 $\gg_u$ 和 $\leq_u$，下标表示运算的有符号性（“u”表示无符号，“s”表示有符号），这与 LLVM IR 中的溢出和比较语义有关。

表 1。图 1 所示窥孔优化的原始形式和广义化形式。

自动泛化。 Hydra 是从具体优化实例自动泛化窥孔优化的最先进方法 [43]。Hydra 构建于 Souper 超级优化器之上 [46], , 通过由启发式方法和贪心算法引导的归纳式综合与枚举式综合来泛化变换。尽管它借助 Alive2 [35], 确保可靠性，但若干固有限制制约了其效果：

• **搜索空间导致的综合失败** Hydra 依赖深度受限的枚举式综合，随着指令序列复杂度增加，这种方法容易导致指数级状态空间爆炸。结果往往是在发现有效的广义化规则之前就出现综合失败（例如 超时）。此外，这些僵化的深度界限严格限制了 Hydra 的表达能力：需要两个以上合取项的复杂前置条件通常会导致综合失败 [43], , 使更复杂的优化模式难以实现。

• **泛化不足：** Hydra 依赖启发式规则和贪心算法，通过机械地重新组合语法片段，而非深入理解代码的语义，来进行处理。因此，它可能无法发现更通用的形式，导致泛化不足。例如，Hydra 会以启发式方式将具体常量绑定到变量位宽。当某项优化包含一个常量 c和一个变量 v，且 Bitwidth(v) = c 时，Hydra 会贪心地推断 c 表示 Bitwidth(v)，并返回一个结果，即使 c 在语义上可能与 v的位宽毫无关联。这

造成了一个固有的两难局面：一方面，应用这一启发式规则可能会错误绑定常量，将规则限制在一个狭窄的适用范围内，使其仅在 Bitwidth(v) = c时成立，从而错过真正通用的形式；另一方面，禁用这一启发式规则又可能会遗漏有效情形，因为在这些情形中，该常量确实表示某个变量的位宽。

• **有限的领域支持：** Hydra 仅适用于整数运算，明确排除了单精度浮点运算和内存指令。这一限制源于综合搜索空间的复杂度过高，以及这些领域中的验证难题。因此，Hydra 无法涵盖现代高性能编译器中大量关键的窥孔优化。

由于存在这些局限，对于图 1 所示的说明性示例，Hydra 无法生成结果。我们对 LLVM 社区的观察表明，手动泛化仍是主流做法 [5, 12, 13]。

2.3 窥孔优化验证

泛化的窥孔优化必须验证其语义正确性和性能收益性。现有多种工具可用于这些验证任务。

形式化语义验证。Alive2 是一种针对 LLVM IR 的有界翻译验证工具 [35]。它通过调用 Z 3 [23] 求解 SMT [20] 查询以进行精化检查，实现全自动验证。给定变换前后的 LLVM I R 序列，它会检查变换后的 IR 序列是否细化变换前的 IR 序列：对于每个使变换前的 IR 序列有定义的输入，变换后的序列都必须产生相同的结果，且不得引入额外的未定义行为。

性能收益评估。 语义正确并不保证性能有所提升：变换可能导致性能退化，例如增加指令延迟或降低吞吐量。因此，通常需要进行收益评估，主要有两种方法。 1 基于启发式的成本模型（如 Hydra 的 [43] ）会为指令分配静态成本，以估算收益性。然而，静态指令成本只能粗略地反映运行时行为，因此可能会误判收益性，纳入导致性能退化的泛化结果。 2 基于模拟的性能分析，例如 LLVM 提供的 llvm-mca（机器代码分析器） [32] ，利用 LLVM 调度模型中的信息来测量机器代码在特定 CPU 上的性能，从而更准确地评估收益性。实际应用中，首先使用 llc（LLVM 编译器）将 LLVM IR 降低为汇编代码， [31],随后由 llvm-mca 分析生成的代码，获取多项性能指标，例如微操作（uOps）数量和估计的周期数。

3 方法论

本节介绍 LPG 如何从具体优化实例中合成广义化的窥孔优化。LPG 作为自动化闭环流水线运行，图 2 概述了其工作流程。LPG 通过以下四个阶段广义化窥孔优化：

(1) **常量符号泛化**(§ 3.3)。将具体常量替换为符号常量。(2) **结构泛化** (§ 3.4)。将固定子表达式（例如 $A + B$）抽象为受约束的符号值（例如 V），消除语法依赖。(3) **约束放宽** (§ 3.5)。通过移除前置条件、弱化前置条件，以及移除指令标志和属性，扩大有效泛化域。

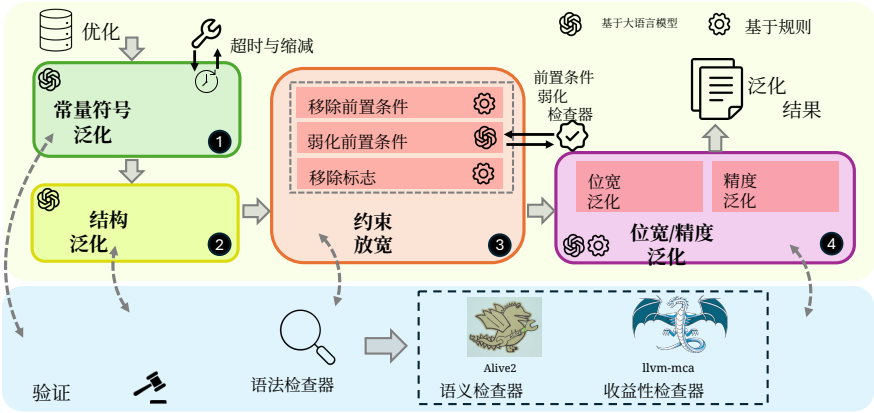

图 2. LPG 的整体工作流程。

(4) **位宽/精度泛化** (§ 3.6)。对于整数优化，LPG 执行位宽泛化，将特定位宽下的泛化扩展到更宽的数据类型范围。对于浮点优化，LPG 执行精度泛化，将特定精度下的泛化扩展到更广的浮点类型范围。

3.1 说明性示例

我们以图 1 中的窥孔优化为例，说明 LPG 的泛化过程。表 2 逐步展示了这一过程，其中每一步都对应一种泛化策略，最终得出表 1 中的泛化结果。

LPG 从一个具体优化实例开始，其有效性取决于字面常量 63 和 8、特定的语法形式 zext(−x & 63)、nsw 标志以及固定的 LLVM 整数位宽。此时，该优化仅适用于这一个具体示例。此工作流程旨在对该示例进行泛化，同时保证其正确性。

步骤①：常量符号泛化。 在常量符号泛化阶段，LPG 将具体常量替换为符号常量 C_1 和 C_2，并引入用于描述该优化所需属性的前置条件。由此得到的候选项保留了原有的计算模式，但不再受限于单个常量实例（63 或 8）。

步骤② 结构泛化。 随后，LPG 执行结构泛化。它不再要求具体的子表达式 zext(−x & C_1)，而是引入一个新的符号值 V，并附加约束 $0 \leq_u V \leq_u \text{zext}(C_1)$。这一步消除了对特定语法树的依赖，只保留优化所需的语义属性。

步骤③ 约束放宽。 获得这一经过结构泛化的候选项后，LPG 进入约束放宽阶段。如表 2 所示，该阶段分为三个子步骤。LPG 首先移除冗余的前置条件 $C_1 \geq_s 0$。随后，它将 C_2 的上界从具体常数 32 放宽为更一般的条件 64 –人口计数(C_1)。最后，它移除 nsw 标志。每个子步骤都会消除从原始优化继承的限制，从而扩大适用域。

步骤④ 位宽/精度泛化。最后，LPG 通过将固定的 LLVM 整数类型替换为符号宽度参数 W 来实现位宽泛化。由此得到的规则不再局限于 i32 和 i64；只要满足 C_1、C_2 和 V 之间的约束，它就适用于任意整数位宽。正如图 2 所示的工作流程所体现的，每个

Step	窥孔优化	变换
原始优化（§ 2.1）	$(0 -_{\text{nsw}}$ 零扩展$(-x : \text{i32} \,\&\, \square) \gg_{\text{u}} \square \mid 0 -_{\text{nsw}}$ 零扩展$(-x \,\&\, 63)$ $\square$ $\Rightarrow \text{sext}(x \,\&\, \square \neq 0)$	
① 常量符号泛化（§ 3.3）	$C_1 : \text{i32} \geq_{\text{s}} 0 \wedge$ 二的幂$(\square + 1) \wedge$ $\square$人口计数$(\leq_{\text{u}} \square)\ 64 \leq_{\text{u}} 32 : \text{i}$ $\vDash (0 -_{\text{nsw}}$ 零扩展$(\square)) \gg_{\text{u}} \square \mid 0 -_{\text{nsw}}$ 零扩展$(\square)$ $\Rightarrow$ 符号扩展$(\square \neq 0)$	$63 \to C_1$ $8 \to C_2$
② 结构泛化（§ 3.4）	$C_1 : \text{i32} \geq_{\text{s}} 0 \wedge$二的幂$(C_1 + 1) \wedge$ 人口计数$(C_1) \leq_{\text{u}} C_2 : \text{i64} \leq_{\text{u}} 32 \wedge$ $0 \leq_{\text{u}} \square : \text{i64} \leq_{\text{u}}$ 零扩展(C_1) $\vDash (0 -_{\text{nsw}} \square) \gg_{\text{u}} C_2 \mid 0 -_{\text{nsw}} \square$ $\Rightarrow$ 符号扩展$(\square \neq 0)$	零扩展$(-x \,\&\, C_1) \to V$ $(-x \,\&\, C_1) \to V$
③约束放宽（§ 3.5）		
移除前置条件	二的幂$(C_1 + 1) \wedge$ 人口计数$(C_1) \leq_{\text{u}} C_2 : \text{i64} \leq_{\text{u}} \square \wedge$ $0 \leq_{\text{u}} V : \text{i64} \leq_{\text{u}}$ 零扩展(C_1) $\vDash (0 -_{\text{nsw}} V) \gg_{\text{u}} C_2 \mid 0 -_{\text{nsw}} V \Rightarrow$ 符号扩展$(V \neq 0)$	移除 $C_1 \geq_{\text{s}} 0$
弱化前置条件	二的幂$(C_1 + 1) \wedge$ 人口计数$(C_1) \leq_{\text{u}} C_2 : \text{i64} \leq_{\text{u}} \square \wedge$ $0 \leq_{\text{u}} V : \text{i64} \leq_{\text{u}}$ 零扩展(C_1) $\vDash (0 \square V) \gg_{\text{u}} C_2 \mid 0 \square V \Rightarrow$ 符号扩展$(V \neq 0)$	$32 \to$ $64 -$ 人口计数(C_1)
标志与属性移除	二的幂$(C_1 + 1) \wedge$ 人口计数$(C_1) \leq_{\text{u}} C_2 : \square 4 \leq_{\text{u}} \square -$人口计数$(C_1) \wedge$ $0 \leq_{\text{u}} V : \square \leq_{\text{u}}$ 零扩展(C_1) $\vDash (0 \square V) \gg_{\text{u}} C_2 \mid (0 \square V) \Rightarrow$ 符号扩展$(V \neq 0)$	移除 nsw
④ 位宽/精度泛化（§ 3.6）	二的幂$(C_1 + 1) \wedge$ 人口计数$(C_1) \leq_{\text{u}} C_2 \leq_{\text{u}} \square -$人口计数$(C_1) \wedge$ $0 \leq_{\text{u}} V \leq_{\text{u}} C_1$ $\vDash (-V) \gg_{\text{u}} C_2 \mid (-V) \Rightarrow$ 符号扩展$(V \neq 0)$	$64 \to W$

注：$\square$ 常量符号泛化；$\square$ 结构泛化；$\square$ 移除前置条件。
$\square$ 弱化前置条件；$\square$ 标志/属性移除；$\square$ 位宽/精度泛化。
W 表示任意位宽；PopCount(C) 表示 C中 1 位的数量。

表 2。示 例：从原始窥孔优化实例进行迭代泛化 .

中间候选项随后都可以传递到验证和下游生成阶段，最终结果就是表 1 所示的泛化的窥孔优化。

3.2 可靠性与收益性的验证

在本文中，我们将验证贯穿于泛化流程，只有当泛化候选方案在语义上可靠且具有性能收益时才予以接受。如图 2 所示，在调用下游验证工具之前，LPG 会先对每个生成的候选方案进行语法检查。格式错误或结构不一致的候选方案会

根据检查器提供的语法错误信息，通过重新生成进行修复。只有语法正确的候选方案才会进入后续的两阶段验证，以检查语义正确性和性能收益性。

形式化语义验证。 为确保每项泛化的窥孔优化在语义上均有效，我们调用 § 2.3 中介绍的 Alive2 [35] ，检查左侧 与右侧 LLVM IR 序列之间的细化正确性。对于任何在 Alive2 中违反语义正确性的泛化候选项，我们都会将其舍弃。

性能收益评估。 如 § 2.3 所述，启发式成本模型只能粗略地反映运行时行为。因此，在使用 Alive2 确立语义正确性后，LPG 使用 llvm-mca [32]评估收益性。我们通过 llc [31] 将 LLVM IR 降低为宿主架构的汇编代码，然后调用 llvm-mca 分析生成的代码。我们的实验采用 x86-64 架构，因为它在工业实践和学术研究中均得到广泛应用。按照先前的 LLVM 超级优化工作 [29], ，我们使用微操作数作为主要成本指标。因此，只有在某项变换具有收益性时，我们才会予以保留：右侧 代码序列的微操作数少于左侧 代码序列。

验证反馈循环。 生成候选泛化后，LPG 会运行一个由验证结果驱动的自动细化循环。对于未通过验证的候选项，所得诊断信息（例如，语法错误、Alive2 反例和 llvm-mca 回归报告）会反馈给大语言模型，以细化后续的泛化。目前，我们仅在第一阶段——符号常量泛化——中引入反馈循环，主要是因为该阶段构成后续阶段的基础，且更适合进行迭代细化。

3.3 符号常量泛化

符号常量泛化将窥孔优化中的具体常量抽象为符号表达式，从而将优化的适用范围扩展到特定字面量值之外。LPG 利用大语言模型提出这些候选泛化。借助其语义推理能力，大语言模型会推断常量的结构性作用以及保证有效性所需的前置条件，例如位宽、值范围或移位合法性约束。例如，具体的移位量 C可以抽象为由操作的位宽W约束的符号变量。相比之下，Hydra 依赖于有界搜索空间内的启发式规则和贪心算法，这从根本上限制了其泛化范围。此外，Hydra 会保守地避免将某些常量（例如，0、1 和 −1）符号化，因为在缺乏深层语义上下文的情况下对其进行抽象往往会产生无效候选项。LPG 克服了这一限制，在适当情况下利用大语言模型安全地泛化这些特殊常量。

递归位宽/精度缩减。 对于特别复杂的候选项，使用 Alive2 进行可靠性验证可能会超时。Alive2 通过将程序逻辑编码为基于位向量语义的 SMT 查询来执行精化检查。由于状态空间急剧膨胀，此类查询可能变得难以计算；状态空间会随着整数位宽或浮点精度的增加而显著扩大。为降低有效泛化因求解器资源限制而遭到拒绝的风险，LPG 会采用递归位宽/精度缩减：整数位宽减半（例如 i16 → i8），浮点类型则逐步降级（例如 双精度 →单精度 → 半精度）。此过程中不包含 1 位整数类型，因为它通常表示布尔谓词（真/假）。每一步缩减都会按相同比例（即减半）统一缩放位宽，从而保持它们之间的相对比例。每次缩减后，LPG 都会重新运行验证，以在缩减后的位宽下验证泛化，直到不再超时或达到缩减上限（例如字面量无法再以缩减后的位宽编码，或者

奇数位宽无法继续减半等等。）。如果验证失败或达到缩减上限，但 Alive2 仍然超时，则认为泛化尝试未成功。

3.4 结构泛化

结构泛化通过将中间计算替换为符号输入，对优化的固定表达式结构进行抽象，消除对特定语法树的依赖，同时保留正确性所需的前置条件。表3展示了一个简单示例。结构泛化将子表达式 $x \,\&\, 127$ 替换为符号值 V，并通过范围约束 $0 \leq_u V \leq_u 127$明确其语义效应。该约束意味着 V为非负值，因此 $0 - V <_s 0$ 等价于 $V \neq 0$：当 $V = 0$ 时谓词为假，而对于所有 $1 \leq_u V \leq_u 127$，谓词均为真。因此，优化 $\text{sext}(0 - V <_s 0) \Rightarrow \text{sext}(V \neq 0)$ 是有效的。前置条件至关重要，因为对于负值 V，该优化会失效。

	原始	结构泛化后
PRE	true	$0 \leq_u V : \text{i8} \leq_u 127$
LHS	符号扩展(□ 7 $<_s$ 0)	符号扩展(0 − □ $<_s$ 0)
RHS	符号扩展(□ 27 $\neq$ 0)	符号扩展(□ $\neq$ 0)

表3. 结构泛化将保留的子表达式 $x \,\&\, 127$ 替换为符号值 V，并根据掩码隐含的范围对其施加约束。

关键挑战在于识别适合抽象的子表达式，并保留其编码的隐式约束，从而确保广义化后的优化仍然有效。LPG 通过提示大语言模型执行结构泛化。具体而言，LPG 使用任务特定提示，引导大语言模型将指令结构与数据依赖解耦。LPG 指示模型抽象带有非关系前置条件 或关系前置条件守卫的子表达式。非关系前置条件包括属性抽象，例如 KnownBits（一个值中已知为零和已知为一的位）、IntegerRange（变量可能取值的范围）、等等。或者一组事实，例如 PowerOfTwo 和 LowBitsZero_k（已知为零的值的低 k 位）。对于关系前置条件，LPG 指示模型合成值之间所需的前置条件（例如 $x < y$）。相比之下，Hydra 将前置条件限制为少量固定且预定义的非关系 前置条件；当某项优化的隐式约束无法通过逐值属性捕获时，这会限制其表达能力。

3.5 约束放宽

在此阶段，LPG 通过放宽优化可应用的条件，扩大优化的适用范围。具体而言，LPG 考虑三种泛化方式：移除前置条件、弱化前置条件，以及移除指令标志和属性。每种候选泛化都会按照第 § 3.2 节所述进行验证，以确保正确性。

此阶段分为三个子步骤：

- **移除前置条件：** LPG 枚举现有的前置条件，并尝试逐一移除。只有当所得优化通过验证时，才会保留该移除操作。
- **弱化前置条件：** 对于无法完全移除的前置条件，LPG 会借助大语言模型提出更宽松的替代条件，以接受更多输入。例如，约束 $C > 0$可以放宽为 $C \geq 0$。只有通过验证且确认严格弱于原前置条件的弱化前置条件才会保留。

• **移除标志和属性：** LPG 还会尝试移除可能不必要地限制优化的指令级标志及相关属性，包括整数溢出标志（例如nsw）、浮点快速数学标志（例如 fast）和函数属性（例如 readonly）。与前述情况一样，只有通过验证的移除操作才会保留。

在这些子步骤中，弱化前置条件尤其具有挑战性。以往的枚举方法（例如 Hydra）只会在有限的数据流域内弱化前置条件，例如 KnownBits。虽然这对某些简单情况可能有效，但对于现实世界中的优化往往不够，因为其前置条件通常取决于这些抽象无法表达的、更丰富的语义关系。更一般地说，有用的放宽谓词空间很大，而且通常是连续的，因此难以高效枚举。例如，放宽整数范围时，不仅要决定是否移除某个边界，还要决定将其放宽多少。LPG 利用大语言模型的语义推理，在这一更广阔的搜索空间中提出候选放宽方案，以应对这一挑战。

严格弱化泛化。 对于前置条件弱化，仅保证语义保持还不够：新的前置条件还必须严格弱于原前置条件。为检查这一性质，LPG 在 Alive2 中使用双向细化测试。在 Alive2 中，部分冗余消除 通过在 LLVM IR 中添加假设内置函数来编码。如果输入不满足该部分冗余消除，相应的假设就会被违反，导致程序出现未定义行为 （UB）。由于触发 UB 的程序在语义上等同于任何程序，因此可以被任何其他程序细化。假设原前置条件为 P，弱化后的前置条件为 P'，其中 $P \Rightarrow P'$。那么，在 P条件下的左侧程序可以被在 P'条件下的左侧 程序细化，因为每个满足 P 的输入也满足 P'。然而，如果弱化是严格的，反向细化就不成立：存在满足 P' 但不满足 P的输入。因此，LPG 会检查两个方向的细化，并且仅当原版本可以细化弱化版本、而反向细化不成立时，才接受候选方案。这确保新的前置条件确实严格弱于原前置条件。

3.6 位宽/精度泛化

在这一阶段，LPG 将优化泛化到不同的整数位宽和浮点精度。这对于最大限度地提高优化的适用性至关重要，因为许多优化在一定范围的位宽或精度下都有效，而将其绑定到特定类型实例会限制其用途。

然而，在单一整数位宽或单一浮点精度下证明某项优化成立，并不意味着它对所有类型变体都有效。在 LLVM IR 中，整数类型和浮点类型都必须指定为明确的具体类型（例如 i8、i16、float、double）；iN 之类的符号形式或抽象精度变量并不是有效的 LLVM IR 语法。此外，验证工具链也只能处理具体类型实例。因此，我们无法直接验证适用于任意位宽或精度的广义化优化。

位宽泛化。 对于整数相关优化，LPG 采用与 Hydra 类似的流程进行位宽泛化。首先，LPG 执行保守的静态分析，拒绝包含显式改变宽度的类型转换（截断指令、符号扩展、零扩展）的候选项，并且只允许简单字面量 $\{0, 1, -1\}$。其次，我们擦除候选项中的所有具体整数类型，然后用每一种具体位宽实例化它们，最大宽度可配置（目前设为 64 位），以便逐一验证每个实例化结果。第三，如果穷举实例化仅对位宽的真子集成功，LPG 会使用大语言模型合成

一个关于位宽的附加谓词（例如 $W_1 \leq 8, W_1/W_2 = 2$）。只有当该谓词所涵盖的所有位宽实例化都通过验证时，LPG 才接受这一泛化。
精度泛化。 对于浮点优化，LPG 执行精度泛化：对于由精度 P参数化的候选规则，我们使用预定义的浮点类型集合实例化 P。我们直接对三种类型实例进行穷举：半精度、单精度和双精度。

4 评估

本节介绍了我们对 LPG 的广泛评估，展示其在泛化真实世界的窥孔优化方面的有效性。我们围绕三个关键研究问题组织评估：

- **RQ1：** LPG 能否有效泛化真实世界的窥孔优化？
- **RQ2：** 每种策略对泛化质量有何影响？
- **研究问题3：** 不同的大语言模型如何影响 LPG 的泛化性能？

4.1 实验设置

本节介绍用于评估 LPG 的实验设置，包括硬件配置、所选模型、数据收集与预处理流程，以及评估指标。

4.1.1 硬件配置。 所有实验均在一台搭载 Ubuntu 22.04（64 位）的服务器上进行，该服务器配备一颗 3.00 GHz 的 Intel Xeon Gold 5217 CPU，拥有 32 个核心和 376 GB 内存。

4.1.2 所选模型。 表4总结了评估中使用的模型。在 RQ1 和 RQ2 中，我们使用 Gemini3-pro-preview 作为默认模型，以便在固定的模型配置下开展有效性评估和策略研究。在 RQ3 中，我们评估了四种不同的模型：Qwen3-235B-Instruct、Qwen3-235B-Thinking、Qwen3-30B-Thinking 和 Qwen3-30B-Instruct，以考察模型规模和推理能力如何影响 LPG 的泛化性能。Gemini3-pro-preview 的知识截止日期为 2025 年 1 月，而四种 Qwen3 变体的知识截止日期均为 2025年4月。

模型	推理	知识截止日期
Gemini3-pro-preview	Yes	2025年1月
Qwen3-30B-Instruct	No	2025年4月
Qwen3-30B-Thinking	Yes	2025年4月
Qwen3-235B-Instruct	No	2025年4月
Qwen3-235B-Thinking	Yes	2025年4月

表4。用于评估的选定模型

4.1.3 数据收集。 为了构建一个稳健的测试套件来评估 LPG 的有效性，我们整理了两个数据集，即 HydraDataset 和 LPGDataset，其中包含从 LLVM 项目的 GitHub 问题跟踪器收集的 102 个不同的窥孔优化实例。为确保测试套件的质量，每个窥孔优化实例都经过了严格验证：我们筛除了

无效案例，仅保留通过可靠性和收益性验证（§ 3.2）的案例。

HydraDataset 为直接与 Hydra 进行比较，Hydra [43],HydraDataset 复现了原始 Hydra 论文评估中使用的所有实例，其中包括从 LLVM GitHub 问题中收集的 40 个整数相关窥孔优化。

LPGDataset 为了验证 LPG 在不同优化领域中的有效性，并消除数据泄漏风险，我们通过挖掘 LLVM GitHub 问题中的 62 项新窥孔优化，整理出一个额外的数据集 LPGDataset。LPGDataset 涵盖多个指令类别的优化，包括 41 项整数优化、16 项浮点优化和 5 项内存优化。值得注意的是，LPGDataset 中的所有实例均来自 2025 年 5 月之后报告的问题，晚于受评估大语言模型的知识截止日期。这确保了对 LPG 性能的评估依据的是其推理能力，而非对训练数据的记忆。

```
定义 i8 @左侧(i64 %arg0) {
  %v0 = 截断指令 i64 % arg0 到 i8
  %v1 = urem i8 %v0, 25
  %v2 = urem i8 %v1, 5
  返回 i8%v2
}

定义 i8 @rhs(i64 %arg0) {
  %v0 = 截断指令 i64 % arg0 转换为 i8
  %v2 = urem i8 %v0, 5
  ret i8 %v2
}
```

（a）剪枝前的窥孔优化

```
定义 i8 @lhs(i8 %newvar_v0) {
  %v1 = urem i8 %newvar_v0, 25
  %v2 = urem i8 %v1, 5
  返回 i8%v2
}

define i8 @rhs(i8 %newvar_v0) {
  %v2 = 无符号余数指令 i8 % 新变量_v0, 5
  返回 i8%v2
}
```

(b) 剪枝后的窥孔优化

图 3. 窥孔优化中剪枝的示例。在 LLVM IR 中，trunc 是将值转换为较小类型的截断指令，而 urem 是无符号余数指令。

4.1.4 数据预处理。此步骤旨在去除真实世界原始优化实例中的噪声，这些实例通常包含与优化无关的代码。为提取核心优化逻辑，我们采用 Hydra [43] 的验证引导剪枝过程。遵循这一思路，我们通过简化来移除不必要的数据流依赖。对于每个以函数对（左侧，右侧）表示的优化实例，我们对两个函数反复执行剪枝迭代。每次迭代中，我们一次考察一个变量定义，并尝试将该变量的所有使用替换为一个新变量，该变量作为额外的函数参数引入。图 3 展示了剪枝过程的一个具体示例，其中图 3a 和图 3b 分别展示剪枝前后的优化实例。剪枝会将原始实例中计算出的中间值（%v0）替换为一个新参数（%newvar_v0）。剪枝过程中，每次尝试替换都会经过可靠性和收益性验证，只有两个验证阶段均成功时才会接受替换；否则便会撤销。每次接受替换后，都会删除死指令，并重复剪枝过程，直至某次迭代不再产生任何变化。因此，每条保留的指令对于确保语义正确性和性能收益性都不可或缺。

表5. LPG 在不同指令领域中的窥孔优化泛化结果。

指令领域	#实例数	#成功数
整数优化	81	74
浮点优化	16	12
内存优化	5	4
总体	102	90

表6。LPG 与 Hydra 在整数相关窥孔优化上的比较。

数据集	#指令数	#LPG 后继。	#Hydra 后继。	#LPG 更好	#Hydra 更好
HydraDataset	40	36	23	14	1
LPGDataset（整数）	41	38	12	8	1
总体	81	74	35	22	2

4.2 研究问题1：LPG 能否有效地广义化真实世界的窥孔优化?

为定量评估 LPG 在真实世界窥孔优化上的泛化能力，我们针对 HydraDataset 和 LPGDataset 中的所有优化实例开展了实验，这些数据集见第 4.1.3 节。为将 LPG 与最先进方法进行比较，我们将其与 Hydra 进行了比较。Hydra 是一种先前提出的泛化方法，仅适用于整数优化。我们在 Hydra 的公开可用实现 [42] 上运行了 Hydra，测试数据包括其原始数据集（HydraDataset）以及 LPGDataset 中与整数相关的窥孔优化。我们向每种工具提供具体的窥孔优化实例，并要求其生成泛化的窥孔优化。

4.2.1 结果。表5总结了 LPG 在数据集中所有实例上的整体效果，并按指令领域进行了分类。总体而言，LPG 成功合成了 90 条广义化重写规则，并通过了验证，共涉及 102 个实例。具体而言，LPG 在 81 项整数优化中成功处理了 74 项，在 16 项浮点优化中成功处理了 12 项，在 5 项内存优化中成功处理了 4 项。这些结果表明，LPG 的泛化能力超越了以往工作所针对的以整数为中心的设置，并且能够针对不同指令领域合成正确且有收益的规则。

表6展示了 LPG 与 Hydra 在整数相关窥孔优化方面的比较。LPG 成功广义化了 81 个实例中的 74 个（HydraDataset 上 40 个中有 36 个，LPGDataset 上 41 个中有 38 个），而 Hydra 仅成功处理了 35 个实例（HydraDataset 上 23 个，LPGDataset 上 12 个）。此外，Hydra 成功广义化的每个实例，LPG 也都成功广义化了。在两种工具都成功广义化的案例中，LPG 在 22 个案例中生成了更广义的重写规则，而 Hydra 仅在 2 个案例中做到了这一点。

4.2.2 分析。除了整体效果外，这两种方法在失败情形上也呈现出截然不同的模式。具体而言，LPG 的错误特征相对集中：大多数失败仅出现在少数语义上较难的情形中，尤其是涉及微妙浮点推理或验证器支持有限的情况。相比之下，Hydra 的失败范围更广，源自其综合流水线的不同阶段。这一区别表明，LPG

主要受限于棘手的边缘情况，而 Hydra 则更常受制于通用搜索和综合瓶颈。

LPG。 LPG 在所评估的多数基准测试中成功泛化了绝大多数实例，这表明其泛化流程对于大多数现实世界中的优化都具有稳健性。其余未成功的案例仅集中在少数语义上较为复杂的优化中。具体而言，这些失败包括少量语义反例，以及一个验证结论不确定的案例。反例表明，当正确的泛化依赖于仅凭局部重写难以推断的细微 IEEE-754 语义时，LPG 面临的挑战最大，其中包括有符号零的区别、对 NaN 敏感的比较和 maxnum 式行为，以及正确性取决于避免派生浮点常量出现溢出、下溢和舍入误差的阈值重写 [11, 15, 16]。这些优化还可能依赖次正规数行为和当前舍入模式，因此，从代数上看似合理的重写未必能在浮点语义下保持精确。对于验证结论不确定的案例，其难点似乎并非源于明显错误的泛化，而是由于验证器对 frexp 和 ldexp 等浮点内建函数组合的支持不完善 [14]。综合来看，这些观察表明，LPG 的局限性主要来自极少数处于浮点推理、内存语义与当前验证器支持能力边界上的优化。

Hydra。 相比之下，尽管 Hydra 在基准测试套件中的部分项目上也取得了成功，但其表现明显不够稳定。其失败出现在泛化流程的多个不同阶段，包括反复缩减失败、常量综合预算耗尽、超时和崩溃。更重要的是，这些失败并非局限于少数语义上较难的重写，而是出现在范围广泛得多的重写模式和程序上下文中。例如，对于某项优化 [3], , Hydra 未能生成可用规则，因为综合出的候选最终被判定为“对任何位宽都无效”。即使 Hydra 确实返回了候选结果，该结果往往仍然过于具体，未能捕捉到底层的优化逻辑。

4.2.3 案例研究。 本节介绍 3 个具有代表性的案例研究，对比 LPG 和 Hydra，并分析二者泛化能力的 3 个关键维度。这些案例研究中的每项优化均取自 LLVM GitHub 问题中报告的真实世界优化。

案例研究 ***1***：搜索空间导致的综合失败。 表 7 展示了一个具有代表性的实例 [33]：Hydra 因搜索空间爆炸而超时。具体而言，Hydra 在综合循环中耗尽了常量综合预算。这是因为，正确的泛化必须表示为参数化重写：优化前指令序列通过 smax/smin 执行基于钳位的范围检查，而等价的优化后指令序列则是形如 $x + (C_1 - C_2) <_u C_3 - C_2 + 1$ 的无符号范围测试，其语义前置条件为（例如， $C_2 \le C_3$ 与$(C_3 - C_2) \neq -1$）。该重写还引入了将多个常量相互关联的仿射约束，使 Hydra 难以通过基于程序综合的方法找到正确的泛化候选。相比之下，基于大语言模型的 LPG 能够识别钳位模式，并得出表 7 所示、经过验证的泛化结果。

案例研究 ***2***：泛化不足。表 8 展示了一个窥孔优化实例 [4]，其中 Hydra 生成了一个泛化不足的重写。在此例中，LPG 恢复了位位置测试的语义刻画：在 C_1 与 C_2 为 2的幂这一 部分冗余消除 下，当且仅当 $x =$ 尾随零计数操作(C_2) –尾随零计数操作(C_1) 时，守卫条件 $(C_2 \,\&\, (C_1 \ll x)) \neq 0$ 成立。相比之下，Hydra 返回了一条正确但实例绑定的规则，它将常量之间的特定关系（例如 $C_5 = 8 \cdot C_6$）硬编码，并保留具体结论 $x = 3$，从而大幅缩小了泛化的适用范围。这一差距源于 Hydra 的

表 7。案例研究1中因搜索空间爆炸导致的泛化超时。smin 和 smax 分别是有符号最小值和最大值运算。

Item	优化
(a) 原始优化	$\mathrm{smin}\,(\mathrm{smax}(x-60,\ -155),\ 100) = x-60$ $\Rightarrow (x+95) <_{\mathrm{u}} 256$
(b) Hydra 泛化	超时
(c) LPG 泛化	$C_2 \leq_{\mathrm{s}} C_3 \wedge (C_3 - C_2) \neq -1$ $\vDash \mathrm{smin}\,(\mathrm{smax}(x+C_1,\ C_2)\ C_3) = x + C_1$ $\Rightarrow (x + (C_1 - C_2)) <_{\mathrm{u}} (C_3 - C_2 + 1)$

综合过程：它采用贪心策略并受时间限制，一旦找到通过验证的候选方案便立即终止。此外，Hydra 主要依据有限的一组预定义数据流事实（例如KnownBits）比较已验证的候选方案，而非依据更广泛的通用性语义概念。当这些事实不足以区分多个已验证的候选方案时，Hydra 就难以确定哪种泛化的适用范围更广，因此可能无法始终优先选择适用范围更广的方案。

表 8. 案例研究 2 中的泛化不足。尾随零计数操作是指返回其参数的二进制表示中尾随零个数的操作。

Item	优化
(a) 原始优化	$16 \,\&\, (2 \ll x) \neq 0 \Rightarrow x = 3$
(b) Hydra 泛化	二的幂$(C_5) \wedge$ 二的幂$(C_6) \wedge C_5 = 8 \cdot C_6$ $\vDash C_5 \,\&\, (C_6 \ll x) \neq 0 \Rightarrow x = 3$
(c) LPG 泛化	二的幂$(C_1) \wedge$二的幂(C_2) $\vDash (C_2 \,\&\, (C_1 \ll x)) \neq 0 \Rightarrow x = \frac{\text{尾随零计数操作}(C_2)}{\text{尾随零计数操作}(C_1)}$

案例研究 ***3***：浮点优化的泛化。 表 9 展示了一个具有代表性的浮点钳位优化 [34]。该案例体现了第三个关键差异：LPG 能处理浮点优化，而 Hydra 不能。浮点优化的泛化需要在 IEEE-754 语义 [25], 下进行推理，其中即使是简单的代数变换，也可能因 NaN、 $\pm\infty$、带符号零或舍入而失效，等等。 例如，表 9(c) 中包含两个看似“冗余”的前置条件：fcmp$^{\mathrm{oeq}}(-C_1/2 \cdot 2,\ -C_1)$ 和 fcmp$^{\mathrm{oeq}}(C_1 \cdot 0,\ 0)$。然而，根据 IEEE-754 语义，这些条件是必要的，以确保 C_1 不会下溢为带符号零，并确保 C_1 是有限值（避免由 $C_1 \cdot 0$ 产生 NaN）。与整数模式相比，浮点运算语义的复杂性扩大了搜索空间，使基于综合的方法更难以在固定搜索预算内综合出正确的泛化结果。此外，Hydra 基于 Souper 超优化器构建其泛化流程，而 Souper 仅支持整数和位向量运算。要扩展至浮点窥孔泛化，需要大量手动工程工作，包括支持浮点相关运算符和 IEEE-754 语义，以及重新调优搜索启发式方法，等等。相比之下，LPG 使用大语言模型生成泛化结果，并同时生成使优化有效所需的 IEEE-754 前置条件（例如表 9(c) 中的有限性和无损缩放约束）。因此，LPG 缓解了枚举整数模式之外的候选前置条件所导致的组合爆炸；这是基于程序合成的方法固有的局限，因为这类方法可能产生大量计算开销。

表 9. 案例研究 3 中浮点优化的泛化。fcmp_oeq 是有序相等浮点比较。

Item	优化
(a) 原始优化	$\text{浮点比较}_{\text{oeq}}(2 \cdot x - 1,\ 0) \Rightarrow x = 0.5$
(b) Hydra 泛化	不支持（仅整数）
(c) LPG 泛化	$\text{fcmp}_{\text{oeq}}(-C_1/2 \cdot 2, -C_1) \wedge \text{浮点比较}_{\text{oeq}}(C_1 \cdot 0,\ 0) \vdash \text{fcmp}_{\text{oeq}}(2 \cdot x + C_1, 0) \Rightarrow x = -C_1/2$

RQ1：LPG 能有效泛化真实世界的窥孔优化，在不同指令领域的 102 个实例中保留了 90 个经验证的泛化结果。在整数相关优化方面，它不仅取得了高于 Hydra 的成功数量（74 对 35），还成功泛化了 Hydra 解决的每一个 实例；在双方共同解决的实例中，LPG 更常得到严格更一般的规则（22 对 2）。案例研究进一步表明，LPG 能克服搜索空间爆炸和泛化不足，同时还能将泛化扩展到 Hydra 能力范围之外的浮点优化。

4.3 RQ2：每种策略对泛化质量有何影响?

为评估 LPG 中每种泛化策略如何影响合成的重写规则的质量，我们通过统计每种策略促成保留的泛化的优化实例数量来量化其有效性，即通过 § 3.2 所述验证并获得更广泛适用性的泛化。本分析涵盖 § 3 中描述的四个阶段。为进行更细粒度的分析，我们进一步将约束放宽分解为三个子步骤：移除前置条件、弱化前置条件，以及移除标志或属性。

我们对 LPG 进行了插桩：当某项策略应用生成了被 LPG 保留的候选规则时，便将该策略应用记录为有效。随后，我们汇总了所有成功泛化的优化实例中的这些有效应用，并报告了每个策略阶段或约束放宽子步骤下被接受的应用数量。

表 10. LPG 泛化策略的贡献分析。“#Effective/#Success”表示有效应用数与所有成功泛化的优化数之比。

策略	#有效/#成功
常量符号泛化	90/90
结构泛化	23/90
约束放宽（移除前置条件）	13/90
约束放宽（弱化前置条件）	30/90
约束放宽（标志与属性移除）	24/90
位宽/精度泛化	78/90

4.3.1 结果。 表 10 显示，高层策略阶段和更细粒度的约束放宽操作都对 LPG 接受的泛化作出了实质性贡献。常量符号泛化和结构泛化分别带来了 90 次和 23 次有效应用，而位宽/精度泛化则贡献了 78 次。在约束放宽中，有效应用来自移除前置条件（13 次）、弱化前置条件（30 次），以及

移除标志或属性（24 次）。综合来看，这些结果表明，LPG 的有效性取决于所有这些策略，它们共同扩大了原始窥孔优化的适用范围。

表11. LPG 中有效策略应用的变化幅度。“受影响数/有效数”表示每次有效应用中转换元素的数量。“受影响数”统计泛化常量、结构抽象元素、被移除的前置条件、被弱化的前置条件，以及被移除的标志或属性的数量。

策略	受影响数/有效数
常量符号泛化	216/90
结构泛化	24/23
约束放宽（移除前置条件）	17/13
约束放宽（弱化前置条件）	34/30
约束放宽（标志与属性移除）	48/24

表11通过刻画每种有效策略应用所带来的变化幅度，展示了贡献数量。常量符号泛化在90次有效应用中泛化了216个常量，而结构泛化在23次有效应用中抽象了24个结构元素。在约束放宽中，移除前置条件、弱化前置条件以及移除标志或属性分别在13、30和24次有效应用中影响了17、34和48个元素。对于位宽/精度泛化，我们没有统计受影响元素的数量，因为它作用于整个优化，而非单个元素。总而言之，这些结果表明，LPG的策略不仅在贡献频率上有所不同，其引入的变换粒度也各不相同。

RQ2：所有策略阶段都对LPG的最终效果有所贡献，这表明高质量泛化依赖于它们的互补性，而非任何单一变换。结果还表明，这些策略作用于不同维度，并共同扩大了泛化规则的适用范围

4.4 研究问题3：不同的大语言模型会如何影响 LPG 的泛化性能?

为回答研究问题3（RQ3），我们使用四种 Qwen3 变体实例化了 LPG：Qwen3-235B-Instruct、Qwen3-235B-Thinking、Qwen3-30B-Thinking 和 Qwen3-30B-Instruct。这些模型涵盖两种模型规模（30B 和 235B）以及两种模式（指令模式和思考模式）：指令模式变体直接遵循提示生成响应，而思考模式变体则会在给出答案前进行显式推理。因此，这使我们能够在受控条件下比较模型能力和显式推理行为如何影响 LPG 泛化真实世界的窥孔优化的能力。为尽量减少潜在数据泄漏的混杂效应，我们仅在 LPGDataset 上进行此项比较。对于每个模型，我们都在相同的 LPGDataset 实例上运行相同的预处理、提示、验证和后处理流程，因此唯一变化的因素是底层大语言模型。具体而言，比较 Qwen3-235B-Instruct 与 Qwen3-30B-Instruct，可以在相同的非推理接口下单独考察模型规模的影响；比较 Qwen3-235B-Thinking 与 Qwen3-30B-Thinking，则可以考察推理模式下相应的规模影响。同样，在相同规模下比较思考模式与指令模式变体，可以评估启用显式推理的边际影响。此外，四种 Qwen3 变体的知识截止日期均相同（2025年4月），这有助于减少训练时长不一致带来的混杂效应。

表12. 研究问题3的模型比较。

模型	推理	#成功/#总计
Qwen3-235B-Instruct	No	33/62
Qwen3-235B-Thinking	Yes	53/62
Qwen3-30B-Instruct	No	28/62
Qwen3-30B-Thinking	Yes	37/62

4.4.1 结果。表12报告了 LPGDataset 上的模型比较结果。在四种变体中，Qwen3-235B-Thinking 表现最佳，保留了 62 项泛化结果中的 53 项。其次是 Qwen3-30B-Thinking，保留了 37 项泛化结果；接着是 Qwen3-235B-Instruct，保留了 33 项；Qwen3-30B-Instruct 则保留了 28 项。从这些结果中可以得出两个主要发现。首先，在两种模型规模下，启用显式推理都能持续提升性能。在 235B 规模下，从指令模式切换到思考模式，保留的泛化结果数量从 33 项增至 53 项；在 30B 规模下，同样的切换使数量从 28 项增至 37 项。这一趋势与以下观点一致：思考模式具有更强的有效推理能力，尤其是在泛化规则需要包含多个合取子句的前置条件时。在这类情况下，模型似乎更善于捕捉原始优化中隐含的语义结构，并将其转化为有效且适用范围更广的重写规则。其次，扩大模型规模也能提升性能，但其影响取决于推理模式。在指令模式下，规模从 30B 扩大至 235B 仅带来有限的提升，数量从 28 项增至 33 项。相比之下，在思考模式下，同样的规模扩大会带来大得多的提升，数量从 37 项增至 53 项。

研究问题3：模型选择会显著影响 LPG 的性能，其中 Qwen3-235B-Thinking 的结果最佳（53/62）。显式推理始终能提升两种规模下的性能，而且在推理模式下，增大模型规模带来的收益要显著更大。

4.5 有效性威胁

首要的有效性威胁是潜在的数据泄漏。由于我们的任务是泛化先前报道的窥孔优化，因此被评估大语言模型的训练数据可能已经包含与某些基准测试实例相关的信息，从而导致结果虚高。为缓解这一威胁，我们使用被评估大语言模型的知识截止日期之后报告的 LLVM 问题整理了 LPGDataset，并将其结果与 HydraDataset 的结果分开报告。另一个威胁是，收集的基准测试可能无法充分代表所有真实世界的窥孔优化。为缓解这一威胁，我们在涵盖多个指令领域的两个数据集上评估 LPG，并仅保留通过验证流程的精心整理的案例。

5 相关工作

本节将讨论与 LPG 相关的先前工作。

5.1 泛化窥孔优化

Hydra [43] 是与 LPG 最密切相关的工作。它基于 Souper [46]之上的程序综合，对特定的窥孔优化进行泛化。如 § 2.2 所述，Hydra 仍受搜索空间爆炸的限制，在处理复杂模式时会失败；其启发式、贪心综合策略可能在缺乏语义理解的情况下产生泛化不足的结果，而且它仅支持整数域中的窥孔优化。该方法基于超优化和枚举搜索，

Optgen [21] 是另一项窥孔优化泛化工作，支持符号常量和前置条件。Optgen 通过枚举所有可能的前置条件和常量表达式来运作，但不允许用户对特定的、尚未泛化的优化进行泛化，而且仅支持 8 位和 32 位优化。此外，它不支持数据流事实（例如 二的幂），而这对泛化至关重要。总的来说，它仅支持有限的一组优化，不太适合对特定的现实世界中的优化或更复杂的重写进行泛化。Proofgen [51] 采用了不同的路径来泛化窥孔优化。它采用以证明为中心的方法，从显式正确性证明中推导出广义化优化，而不是采用生成并验证工作流。原则上，这一方向很有力，但也存在一些局限。首先，泛化严格受限于显式证明结构；同一证明树的不同证明，甚至不同线性化顺序，都可能产生彼此完全无法比较的规则。其次，其能力受预定义公理所限，系统只能处理在该领域内被明确涵盖的概念。最后，这种以证明为中心的方法尚未证明能够扩展到复杂的现实世界泛化任务。

相比之下，LPG 采用了更加灵活且可扩展的方法来进行窥孔优化的泛化。LPG 将泛化组织为一个闭环流水线，通过反馈结合了 LLM 引导的符号常量泛化、结构泛化、约束放宽以及位宽/精度泛化，并利用语法检查、语义验证和收益性评估的结果进行调整。这种方法使 LPG 能够处理范围广泛得多的优化模式，并发现更加通用的优化规则。

5.2 超优化

超优化是一种更激进的优化技术，旨在为给定的程序片段搜索等价但更优的实现，通常借助求解器或形式化验证进行验证。Massalin 等人通过穷举枚举较短的指令序列，引入了经典的超优化器 [39]。随后，Josh 等人提出了 Denali，将超优化形式化为由定理证明引导的目标导向搜索 [26], ；与此同时，Bansal 和 Aiken 展示了如何利用暴力搜索自动构建窥孔超优化器 [18]。Schkufza等人提出了 STOKE，通过对无循环 x86 代码采用随机搜索，进一步提升了可扩展性 [47]。在 LLVM 生态系统中，Souper 为 LLVM IR 中受限的、面向数据流的子集合成更高效的等价实现 [46]，而 Minotaur 则将这一方向扩展到面向 SIMD 的代码以及更广泛的指令支持 [29]。

这些系统擅长发现具体的重写，但通常针对固定的程序片段和具体常量。它们无法直接解决我们的场景：从一个具体的遗漏的优化出发，推导出包含符号常量、数据流事实和前置条件、可重复使用的广义化规则。

5.3 自动检测遗漏的优化

另一类互补工作侧重于自动发现生产级编译器遗漏的优化机会。Barany 使用差分测试来识别遗漏的编译器优化 [19]。Taneja 等人使用离线的、基于 SMT 的检查来揭示静态分析中的精度不足，而这些不足可能导致遗漏的优化 [50]。Theodoridis 等人提出将死代码消除作为发现 LLVM 和 GCC 中遗漏的优化的切入点 [52], ，随后还表明，由于意外的优化交互，即使向编译器提供精化输入，也可能导致优化后的输出质量下降 [53]。DITWO 将差分测试扩展到 WebAssembly 优化器 [30]，而 SBC 则交叉验证协调一致的优化行为，以检测不同优化阶段之间遗漏的机会 [56]。最近，Lampo 将大语言模型与形式化验证相结合，以发现 LLVM 中遗漏的窥孔优化 [54]。

这些技术与 LPG 互为补充。它们有助于发现遗漏的机会或导致这些遗漏的分析精度不足，但并不侧重于从具体的优化实例中合成广义化的优化规则。

5.4 寻找最弱前置条件

另一类相关工作侧重于为已知变换推断最弱或足够弱的前置条件，而非合成变换本身。PSyCO [38] 是这一领域早期的编译器专用系统，能够在预定义的前置条件语言中为编译器优化合成最弱前置条件。随后，Alive-Infer [40] 针对以 Alive 表达的 LLVM 窥孔优化采用数据驱动策略：它学习能够区分正例和反例的谓词，并报告最弱前置条件和简洁的部分前置条件。在编译器优化之外，PIE [44] 和 P-Gen [48] 研究更广泛的程序前置条件推断问题。PIE 根据示例按需学习特征，以推断更丰富的前置条件；P-Gen 则利用反例引导的细化推导必要且充分的前置条件。

这些技术与 LPG 互为补充。它们都假设源程序和目标程序已经确定，目标只是推断优化或程序有效时所需的守卫条件；而 LPG 面向的是一个更具挑战性的场景：从具体优化出发，推断出广义化的优化规则本身及其适用条件。

6 结论

本文提出了 LPG，这是一种利用大语言模型泛化窥孔优化的新型框架。我们的核心见解是，大语言模型擅长在泛化过程中进行语义抽象和创造性推理，而严格的验证对于确保泛化的正确性必不可少。因此，LPG 将窥孔泛化组织为一个闭环流水线，将符号常量泛化、结构泛化、约束放宽以及位宽/精度泛化与语法检查、形式化语义验证和收益评估相结合。通过对 LLVM 生态系统中的真实窥孔优化进行全面评估，我们表明，LPG 在整数、浮点和内存领域的 102 个优化实例中成功泛化了 90 个。对于整数相关优化，LPG 在成功率和通用性方面也显著优于 Hydra，这表明由大语言模型引导的泛化能够克服搜索空间爆炸、泛化不足以及限制以往基于程序合成的方法的领域限制。

我们的结果表明，当大语言模型与强大的验证机制结合使用，而不是作为独立的生成器时，它们可以成为编译器优化的实用基础。LPG 的成功表明，类似的方法也可以应用于其他需要对具体程序变换进行语义抽象的编译器任务。更广泛地说，随着具备推理能力的大语言模型不断进步，我们预计像 LPG 这样的框架将日益成为有用的工具，帮助编译器开发人员为现代编译流水线发现更广泛、更可复用且更可靠的优化规则。

参考文献

[1] 2012。指针差值的 InstCombine 优化遗漏。https://github.com/llvm/llvm-project/issues/12792#issuecomment-1232603132[2] 2021。 [InstCombine] 将乘法和与常量的相等比较折叠。https://github.com/llvm/llvm-project/issues/ 51744#issue-1077401101[3] 2022 [ ] 59519 。[4] 2022 55739 + + InstCombine 未能折叠扩展方式不匹配的“or”。https://github.com/llvm/llvm-project/issues/ 。LLVM 问题 #：未能将移位和布尔转换模式优化为简单比较。https: //github.com/llvm/llvm-project/issues/55739

[5] 2022。优化遗漏：将 icmp sgt (fptosi %x), -1 折叠为 fcmp ogt %x, -1.0。https://github.com/llvm/llvm-project/issues/154242#issuecomment-3218334136[6] 2024。关于 LLVM 问题 #79690 的评论： [当结果与 AND 运算时，向量饱和减法应反转方向] 79690 2914688099 。
[7] 2024 139641 0 0 + 0 0 0 0
https://github.com/llvm/llvm-project/issues/ #issuecomment- 。LLVM 问题：InstCombine：在优化安全时，将 (X * .) . 折叠为 X * .。https://github.com/ llvm/llvm-project/issues/139641#issuecomment-2878901127。访问日期：2026-02-09。[8] 2024。LLVM 问题 #86061： [遗漏的折叠，zext(x + 1) - 1 > C => x > C]。
https://github.com/llvm/llvm-project/issues/ 86061#issuecomment-2011298443[9] 2024。LLVM 问题 #88481： [[InstCombine] 嵌套选择的折叠遗漏]。https://github.com/llvm/llvm-project/ issues/88481#issue-2239303371[10] 2024。LLVM 问题 #97168： [独热编码的优化遗漏]。https://github.com/llvm/llvm-project/issues/ 97168#issuecomment-2735787019[11] 2024 1 > 2 => > 3 85267 。
[12] 2025 ~ ~ +
遗漏的折叠，x fmul C C x C。https://github.com/llvm/llvm-project/issues/ 。a & (a - b) 可优化为 a & (a b)，尽管大多数目标架构都有 ANDN 指令。https://github.com/ llvm/llvm-project/issues/
140639 2892729284 #issuecomment- 。未能识别 add cmp 中不必要的符号扩展。
[13] 2025 +
https://github.com/llvm/llvm-project/issues/130088# issuecomment-2706811286[14] 2026。优化遗漏：将 frexp(x).exp < C 折叠为 fabs(x) < $2^{(C-1)}$。https://github.com/llvm/llvm-project/issues/ 186554
[15] 2026 == > == 153991 。优化遗漏：折叠
[16] 2026
x*a b - x b/a。https://github.com/llvm/llvm-project/issues/ 。优化遗漏：使用 icmp 和 select 实现，并将其替换为 186300
llvm.maxnum。https://github.com/ llvm/llvm-project/issues/[17] Alfred V Aho、
Monica S Lam、Ravi Sethi 和 Jeffrey D Ullman。2006。《编译原理、技术与工具》（第 2 版）。培生教育。[18] Sorav Bansal 和 Alex Aiken。2006。自动生成窥孔超优化器。《SIGOPS 操作系统评论》40 卷 5 期（2006 年 10 月），394–403。doi:10.1145/1168917.1168906[19] Gergö Barany。2018。通过差分测试发现遗漏的编译器优化。载于第 27 届国际编译器构造会议论文集（奥地利维也纳）（CC '18）。计算机协会，纽约，美国，82–92。doi:10.1145/3178372.3179521[20] Clark Barrett 和 Cesare Tinelli。2018。模理论可满足性。载于《模型检测手册》。施普林格，305–343。[21] Sebastian Buchwald。2015。Optgen：局部优化生成器。载于国际编译器构造会议。施普林格，171–189。[22] Keith D. Cooper 和 Linda Torczon。2011。《编译器工程》（第 2 版）。Morgan Kaufmann。[23] Leonardo De Moura 和 Nikolaj Bjørner。2008。Z3：高效的 SMT 求解器。载于《软件理论与实践》第 14 届国际会议暨系统构造与分析工具和算法国际会议论文集（匈牙利布达佩斯）（TACAS'08/ETAPS'08）。施普林格出版社，柏林、海德堡，337–340。[24] dtcxzyw。2026。关于 LLVM 问题 #139641 的评论：优化遗漏：-(x & c1) » shamt | -(x & c1) –> x & c1 ? -1 : 0，其中 2^{shamt} > c1 和 shamt ≤ bitwidth(x)/2。https://github.com/llvm/llvm-project/issues/139641#issuecomment- 3979594887
[25] 2019 10 1109 2019 8766229 IEEE。。IEEE 浮点算术标准。
[26] 2002
doi: 。/IEEESTD. 。Rajeev Joshi、Greg Nelson 和 Keith Randall。。Denali：目标导向超优化器。载于 ACM SIGPLAN 2002 年编程语言设计与实现会议论文集（德国柏林）（PLDI '02）。计算机协会，纽约，美国，
304 314 10 1145 512529 512566 – 。doi: 。/ 。Chris Lattner
[27] 2004
和 Vikram Adve。。LLVM：用于持续程序分析与转换的编译框架。载于国际代码生成与优化研讨会论文集：反馈导向与运行时优化（加利福尼亚州帕洛阿尔托）（CGO '04）。IEEE 计算机学会，美国，75。[28] Vu Le、Mehrdad Afshari 和 Zhendong Su。2014。通过输入模等价验证编译器。载于第 35 届 ACM SIGPLAN 编程语言设计与实现会议论文集（英国爱丁堡）（PLDI ' 14 216 226 10 1145 2594291 2594334 ）。
[29] 2024
计算机协会，纽约，美国，– 。doi: 。/ 。Zhengyang Liu、Stefan Mada 和 John Regehr。。Minotaur：面向 SIMD 的合成超优化器。《ACM 程序设计语言会议论文集》
8, 2, 326 2024 25 10 1145 3689766 OOPSLA 文章（10 月），页码。doi: 。/
[30] 2023
Zhibo Liu 、 Dongwei Xiao 、 Zongjie Li 、 Shuai Wang 和 Wei Meng。。探索 WebAssembly 优化器中的优化遗漏。载于第 32 届 ACM SIGSOFT 软件测试与分析国际研讨会论文集（美国华盛顿州西雅图）（ISSTA 2023）。计算机协会，纽约，美国，436–448。doi:10.1145/3597926.3598068[31] LLVM。2026年。llc - LLVM 静态编译器。
https://llvm.org/docs/CommandGuide/llc.html。访问日期：2026-01-11。

[32] LLVM 开发者。2025年。LLVM 机器码分析器。https://llvm.org/docs/CommandGuide/llvm-mca.html
[33] 2026 157315 157315 LLVM 项目。. LLVM 问题 # 。
[34] 2026 185554 https://github.com/llvm/llvm-project/issues/ LLVM 项目。. LLVM 问题 # 。 https://github.com/llvm/llvm-project/issues/ 185554
[35] 2021 2 Nuno P. Lopes、Juneyoung Lee、Chung-Kil Hur、Zhengyang Liu 和 John Regehr。. Alive：LLVM 的有界翻译验证。载于第 42 届 ACM SIGPLAN 国际编程语言设计与实现会议论文集（线上，加拿大）（PLDI 2021）。计算机协会，纽约，纽约州，美国，65–79。doi:10.1145/3453483.3454030[36] Nuno P. Lopes、David Menendez、Santosh Nagarakatte 和 John Regehr。2015年。使用 Alive 实现可证明正确的窥孔优化。载于第 36 届 ACM SIGPLAN 编程语言设计与实现会议论文集（美国俄勒冈州波特兰）（PLDI '15）。计算机协会，纽约，纽约州，美国，22–32。doi:10.1145/ 2737924.2737965[37] Nuno P. Lopes、David Menendez、Santosh Nagarakatte 和 John Regehr。2015年。使用 Alive 实现可证明正确的窥孔优化。载于第 36 届 ACM SIGPLAN 编程语言设计与实现会议论文集（美国俄勒冈州波特兰）（PLDI '15）。计算机协会，纽约，纽约州，美国，22–32。doi:10.1145/ 2737924.2737965[38] Nuno P. Lopes 和 José Monteiro。2014年。编译器优化的最弱前置条件合成。载于第 15 届验证、模型检查与抽象解释国际会议论文集——第 8318 卷（美国加利福尼亚州圣迭戈）（VMCAI *2014* 203 221 10 1007 978 3 642 54013 4_12 ）。施普林格出版社，柏林，海德堡，– 。doi: 。 / - - - - Henry Massalin。. 超优化器：窥探最小程序。载于第二届编程语言与操作系统体系结构支持国际会议论文集（美国加利福尼亚州帕洛阿尔托）（ASPLOS II）。计算机协会，纽约，纽约州，美国，1987 122 126 10 1145 36206 36194 – 。doi: 。 / 。
[39] David Menendez 和 Santosh Nagarakatte。. Alive-Infer：用于 LLVM 窥孔优化的数据驱动前置条件推断。载于第 38 届 ACM SIGPLAN 编程语言设计与实现会议论文集（西班牙巴塞罗那）（PLDI 2017）。计算机协会，纽约，纽约州，美国，49–63。doi:10.1145/3062341.3062372[41] Steven S. Muchnick。1997年。《高级编译器设计与实现》。Morgan Kaufmann。[42] Manasij Mukherjee。2026年。Hydra：GitHub 仓库。https://github.com/manasij7479/hydra [43] Manasij Mukherjee 和 John Regehr。2024年。Hydra：通过程序合成泛化窥孔优化。《ACM 程序设计语言会议论文集》 8, 1, 120 2024 29 10 1145 3649837 OOPSLA 论文（4月），页码。
[40] 2017 doi: 。 / Saswat Padhi、Rahul Sharma 和 Todd Millstein。. 利用学习到的特征进行数据驱动的前置条件推断。《 SIGPLAN 通讯》51, 6（2016年6月），42–56。doi:10.1145/2980983.2908099[45] LLVM 项目。2026年。LLVM 指令合并通道源代码。https://github.com/llvm/llvm-project/tree/main/ llvm/lib/Transforms/InstCombine.[46] Raimondas Sasnauskas、Yang Chen、Peter Collingbourne、Jeroen Ketema、Gratian Lup、Jubi Taneja 和 John Regehr。 2018 1711 04422 [ ] 1711 04422 。Souper：合成式超级优化器。arXiv: 。cs.PL https://arxiv.org/abs/ 。Eric Schkufza、Rahul Sharma 和 Alex Aiken。. 随机超级优化。载于第 18 届编程语言与操作系统体系结构支持国际会议论文集（美国得克萨斯州休斯敦）（ASPLOS ' *13* 305 316 10 1145 2451116 2451150 ）。计算机协会，纽约，纽约州，美国，– 。doi: 。 / 。Mohamed Nassim Seghir 和 Daniel Kroening。. 反例引导的前置条件推断。载于《编程语言与系统》（《计算机科学讲义》，第 7792 卷）。施普林格出版社，柏林，海德堡，451–471。doi:10.1007/978-3-642-37036-6 25_[49] Chengnian Sun、Vu Le、Qirun Zhang 和 Zhendong Su。2016年。深入理解 GCC 和 LLVM 中的编译器错误。载于第 25 届软件测试与分析国际研讨会论文集（德国萨尔布吕肯）（ISSTA *2016* 294 305 10 1145 2931037 2931074 ）。计算机协会，纽约，纽约州，美国，– 。doi: 。 / 。Jubi Taneja、Zhengyang Liu 和 John Regehr。. 测试静态分析的精确性与可靠性。载于第 18 届 ACM/IEEE 代码生成与优化国际研讨会论文集（美国加利福尼亚州圣迭戈）（CGO '20）。计算机协会，纽约，纽约州，美国， 81 93 10 1145 3368826 3377927 – 。doi: 。 / 。
[50] 2020 2013 2010 Ross Tate、Michael Stepp 和 Sorin Lerner。. 从证明生成编译器优化。载于第 37 届 ACM SIGPLAN-SIGACT 编程语言原理年度研讨会论文集（西班牙马德里）（POPL '10）。计算机协会，纽约，纽约州，美国， 389 402 10 1145 1706299 1706345 – 。doi: 。 / 。Theodoros Theodoridis、Manuel Rigger 和 Zhendong Su。. 从死代码消除的视角发现遗漏的优化。载于第 27 届 ACM 编程语言与操作系统体系结构支持国际会议论文集（瑞士洛桑）（ASPLOS '22）。计算机协会，纽约，纽约州，美国，697–709。doi:10.1145/3503222.3507764[53] Theodoros Theodoridis 和 Zhendong Su。2024年。精化输入，退化输出：编译器行为的反直觉世界。8, PLDI，文章 174（2024年6月），21 页。doi:10.1145/3656404
[51] 2022
[52]

[54] Zhenyang Xu、Hongxu Xu、Yongqiang Tian、Xintong Zhou 和 Chengnian Sun。2026年。LPO：利用大语言模型发现遗漏的窥孔优化。载于第 *31* 届 *ACM* 编程语言与操作系统体系结构支持国际会议论文集，第 *2* 卷（美国）*(ASPLOS '26)*。计算机协会，纽约，纽约州，美国，

1136 1150 10 1145 3779212 3790184 – 。doi: 。 / 。Xuejun Yang、[55] 2011
Yang Chen、Eric Eide 和 John Regehr。. 发现并理解 C 编译器中的错误。载于第 *32* 届 *ACM SIGPLAN* 编程语言设计与实现会议论文集（美国加利福尼亚州圣何塞）*(PLDI '11)*。计算机协会，纽约，纽约州，美国，283–294。doi:10.1145/1993498. 1993532[56] Yi Zhang、Yu Wang、王林章和王科。2025年。同步行为检查：一种发现遗漏编译器优化的方法。《*ACM* 程序设计语言会议论文集》9, OOPSLA2，文章 327（2025年10月），28 页。doi:10.1145/3763105