arXiv:2602.18511v1 [cs.PL] 19 Feb 2026

超越逐遍优化：基于大语言模型的意图驱动中间表示优化

Lei Qiu [1 2] **Zi Yang** [1 3] **Fang Lyu** [1] **Ming Zhong** [4] **Huimin Cui** [1 2] **Xiaobing Feng** [1 2]

摘要

现代编译器通过对中间表示（中间表示）依次执行一系列模块化优化遍次来优化程序。尽管这种逐遍优化范式具有工程优势，却存在遍协调问题：局部有利的变换可能会阻碍后续阶段中收益更大的优化。这一局限源于缺乏明确的优化意图概念。优化意图被定义为一种整体策略，用于协调多个变换以实现全局性能目标。近期基于大语言模型的方法将中间表示优化表述为端到端生成任务，从而避开传统的逐遍优化结构。然而，这些方法中的优化意图仍然是隐式的，迫使模型同时推断优化策略并生成底层变换，从而限制了正确性和性能。我们提出 IntOpt，这是首个意图驱动的中间表示优化器，明确将高层优化意图与底层分析和变换分离。IntOpt 将中间表示优化划分为三个阶段：意图制定、意图细化和意图实现，从而实现全局协调的变换。实验表明，在包含 200 个程序的测试集上，IntOpt 的已验证正确性达到 90.5%，平均加速比为 2.660×，在正确性和性能方面均优于最先进的基于大语言模型的优化器，并在 37 个基准测试上超过了采用 -O3 选项的现代编译器，加速比最高达 272.60×。

1. 引言

现代编译器利用中间表示（IR），实现与编程语言和硬件架构相互独立的编译（Aho 等，2006年；Lattner 和 Adve，2004年）。通过在中间表示层进行优化，抽象掉语言和硬件特定的细节，编译器能够提升程序执行效率（即**“性能”**），适用于各种编程语言、目标架构和应用领域，涵盖传统科学计算（¨Moses 等，2023；Bezanson 等，2017；Ozen 等，2018年）到采用特化张量运算的现代人工智能工作负载（Lattner 等，2021；Chen 等，2018年；Sabne，2020）。因此，设计有效的中间表示优化策略在编译器构造中始终具有重要意义。

然而，中间表示 级优化的有效性从根本上受限于现代编译器组织优化的方式。在实践中，现代编译器将中间表示优化分解为一系列优化遍次，每个遍次都作为一个模块化单元实现，针对特定的优化，例如循环展开或常量传播（Lattner 与 Adve，2004；Stanier 与 Watson，2013）。这些优化遍次通过分析从中间表示中提取的、与优化相关的信息（下文简称**“分析”**），来判断某种变换是否有益。尽管这种**逐遍优化范式**在模块化和可维护性方面带来了显著的工程优势，但它本质上存在遍协调问题：早期优化遍次中局部有利的优化，可能会无意中限制甚至阻止后续阶段中收益更大的优化。

如图 1(b) 所示，当编译器优化图 1(a) 中源代码对应的中间表示时，LICM- Pass（循环不变代码外提）会将 D [i] 从 L_3 外提至L_2'，以消除内层循环中的冗余访问。尽管这种变换在局部上有益，却会阻止 Loop-InterchangePass 交换 i 和 j 循环来改善内存局部性。更关键的是，由于无法应用 LoopInter- changePass，后续的 LoopVector- izePass 也无法对 L_4' 进行向量化。因此，最内层循环对 A [j] [i] 和 B [j] [i]，仍采用按列访问模式，这妨碍了 SIMD 利用率的有效提升，并最终导致性能下降。

这种行为源于缺乏明确的**优化意图**这一概念。我们将其定义为一种整体策略，用于协调多个变换，以实现全局性能目标。在 GCC（GCC，

[1]SKLP、ICT、中国科学院、中国 [2]中国科学院大学、中国 [3]中国江南大学 [4]香港中文大学、中国。通信作者：Fang Lyu<flv@ict.ac.cn>、Huimin Cui <cuihm@ict.ac.cn>。

预印本。*2026*年*2*月*24*日。

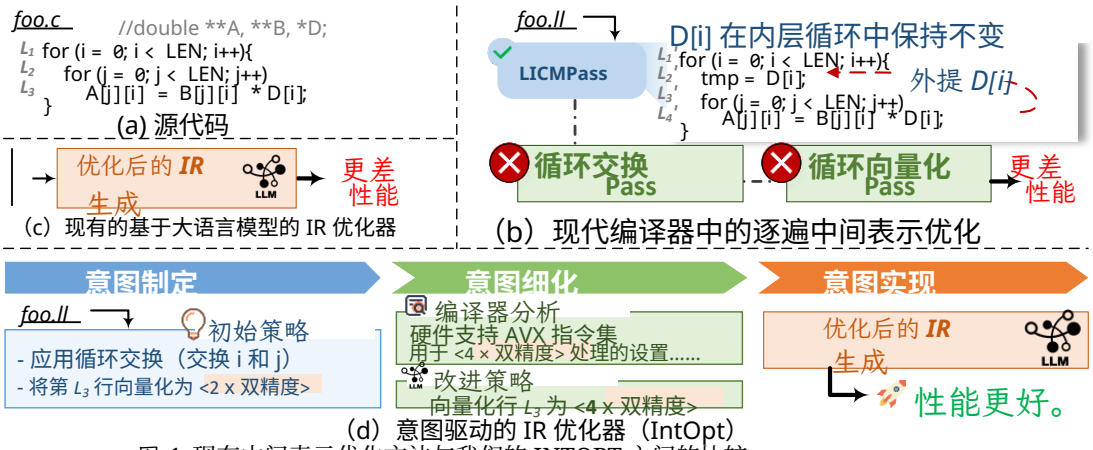

图 1. 现有中间表示优化方法与我们的 INTOPT 之间的比较。

2026年）和 LLVM（LLVM，2026a）等编译器中，这种意图隐含于各个优化遍次之中，且彼此割裂。如图 1（b）所示，*LICMPass* 可减少内存操作，循环交换遍次可改善内存局部性，而循环向量化遍次则旨在进行 SIMD 优化。然而，缺乏明确的全局协同意味着各个优化遍次无法预见其变换可能如何促成或阻碍后续的优化机会（Kulkarni 等人，2006年；Pallister 等人，2015）。

大语言模型（LLM）的最新进展（OpenAI，2025年；Anthropic，2025年；Daya 等人，2025年）凭借强大的程序理解和语义推理能力，为解决这一局限提供了颇具前景的途径。以往基于大语言模型的 IR 优化方法（Cummins 等人，2025年；Yang 等人，2025年）将中间表示优化建模为端到端生成任务，直接将未优化的中间表示映射为优化后的 IR，如图 1(c) 所示，从而避开了编译器中传统的逐遍次优化结构。然而，这种重新表述并未从根本上解决核心问题：优化意图仍然是隐式的，只不过现在隐藏在黑盒大语言模型的生成过程中，而不再是分散在编译器的各个优化遍次中。因此，大语言模型不得不同时推断优化意图并生成精确的变换，这使得保证语义正确性和性能提升的一致性颇具挑战性。我们在第 4.2 节中的实验结果表明，实际仍有很大的改进空间。

本文提出 INTOPT——一种意图驱动的 IR 优化器。它通过将高层优化意图与底层分析和变换解耦，重新构建现代编译器的优化范式，从而实现全局协调的变换。INTOPT 将中间表示优化过程拆解为三个阶段，以实现这一目标：**意图制定**、**意图细化**和**意图实现**。这一流程体现了专家级编译器工程师的工作方式：首先从全局层面推理优化意图，然后通过以下方式将其落到实处

程序分析来评估可行性，最后通过具体变换实现该意图（Muchnick，1998）。

我们的核心洞见是，尽管现有编译器将优化意图隐式地编码在手工编写的优化遍中，但这些意图蕴含着宝贵的领域知识，大语言模型能够学习并将其显式化。基于这一点，IntOpt 首先使用一个专门用于编译器优化的大语言模型，为给定程序推断初始优化策略，以结构化形式表示优化意图。随后，通过编译器分析对该策略进行落地和改进。例如，在 图 1(d) 中，基于硬件感知分析，将初始的向量化策略 `<2 x double>` 改进为 `<4 x double>`。改进后的策略随后指导优化后的中间表示生成，确保各项变换在全局范围内协调一致。在意图改进与实现阶段，IntOpt 利用 GPT-5（OpenAI，2025年）及其预训练知识，探索现有编译器优化遍未能捕捉到的优化机会，如 第 4.4 节所示。

我们在 200 个 LLVM IR 程序上评估 INTOPT，结果表明，与端到端基于大语言模型的优化器和传统编译器相比，INTOPT 均取得了显著提升。我们的结果凸显了显式表达优化意图对于实现全局协调且有语义依据的中间表示优化的重要性。我们希望这些发现能够推动该领域对意图驱动的编译器优化开展更多研究。

总而言之，我们的主要贡献包括：

- **意图驱动的中间表示优化框架**。 我们提出了首个意图驱动的中间表示优化框架，将优化意图与具体的中间表示转换解耦，从而实现更有效、更可靠的优化。
- **基于编译器知识的中间表示优化**。 我们开发了一种新颖的方法，将结构化的编译器知识融入其中，以指导大语言模型进行中间表示优化。
- **全面评估**。 INTOPT 在 200 个程序组成的测试集上，已验证正确性达到 90.5%，平均加速比为 2.660×

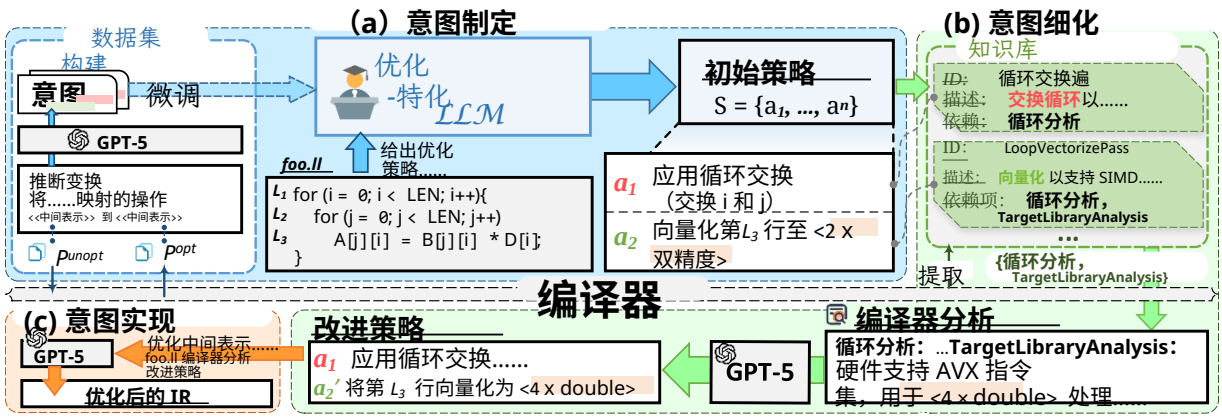

图 2. INTOPT 概述。

在测试集上，INTOPT 的表现优于最先进的基于大语言模型的优化器，正确性高出 7.0%–31.5%，性能最高提升 1.603×；此外，在 37 个基准程序上也超越了 LLVM-O3，加速比最高达 272.60×。

2. 相关工作

传统的编译器中的中间表示优化。 现代编译器（如 LLVM（LLVM，2026a）和 GCC（GCC，2026）通过固定的优化遍次序列执行中间表示优化（例如，`-O3`），其中每个优化遍次在预定义的一组编译器分析与人工设计的启发式规则指导下应用特定变换（Lattner & Adve，2004）。为提供灵活性，编译器公开了大量配置选项，使用户能够启用或禁用特定优化，或调整启发式参数（Muchnick，1998）。然而，各优化遍之间的交互往往复杂且难以预测（Kulkarni 等人，2006年；Pallister 等人，2015），因此难以针对给定程序推理或控制全局优化策略。

编译器优化自动调优。 在逐遍优化基础设施的基础上，一系列研究（Liang 等人，2023；Ansel 等人，2014年；Chen 等人，2021；Zhu 等人，2024；Park 等人，2022；Pan 等人，2025年）探索了通过搜索编译器选项、优化标志或遍次配置来提升性能的自动调优方法。尽管这些方法能够带来性能提升，但它们存在若干固有局限。首先，搜索过程耗时，因为它需要在多种配置下反复编译（对于以运行时性能为目标的情况，还需执行）。其次，优化选项的粒度较粗：禁用、启用或重新排序某个遍次，会影响整个程序中该变换的所有出现位置。例如，如果在选项层面禁用循环不变代码外提，以避免损害特定循环（如图 1（a）所示），那么其他适合进行循环不变代码外提的循环也会受到影响。

用于编译器和代码优化的大语言模型。 近期研究开始探索将大语言模型用于编译器构造和程序优化。一些方法着重提升编译器后端开发效率（Zhong 等人，2024；2025b；a），另一些则针对向量化进行源级程序重写（Taneja 等人，2025年）。与我们工作更密切相关的是直接将大语言模型应用于中间表示级优化的研究（Cummins 等人，2025年；Yang 等人，2025年），这些研究将优化表述为端到端生成任务。尽管这些方法绕过了传统的逐遍流水线，但通常依赖黑盒生成，缺乏对优化意图的显式表示。相比之下，我们将优化意图建模为一个显式的中间步骤，并将基于大语言模型的推理与编译器知识相结合，以实现全局协同的中间表示优化。

3. 提出的方法：INTOPT

为在编译器优化过程中明确地建模并推理优化意图，我们提出了 INTOPT，这是一种意图驱动的中间表示优化框架。如图 2所示，INTOPT 包含三个阶段：意图制定（第 3.1 节）、意图细化（第 3.2 节）和意图实现（第 3.3 节）。这一结构将高层优化意图与底层分析和变换解耦，使 INTOPT 能够进行全局协调的中间表示优化。

3.1. 意图制定

意图形成阶段针对给定的输入 IR 程序 P，生成初始优化策略 S，该策略是对优化意图进行结构化实例化的结果：

$$S = \mathcal{F}_\theta(P)$$

其中，$\mathcal{F}_\theta$ 表示一个由 θ 参数化、经过编译器优化行为微调的优化专用大语言模型。策略 S 表示为一系列有序的变换操作 $\langle a_1, a_2, \ldots, a_n \rangle$，其中每个操作

a_i 用自然语言指定要对哪个程序区域应用哪种变换（例如，“使用<2 x double>向量化第 L_3 行”）。

用自然语言表示操作，可以避免过早限制优化操作空间，使模型能够在更丰富、更灵活的优化策略空间中进行推理。尽管具有这种灵活性，这个问题仍然易于处理：虽然语义等价的中间表示程序空间极其庞大，但现代编译器所提供的变换操作集合是有限的，且由编译器开发者精心挑选。这个有界的操作集合使意图制定适合通过规模适中的数据集进行数据驱动学习。因此，为了微调意图制定模型 $\mathcal{F}_\theta$，我们通过从编译器行为中提炼优化意图来构建数据集。

数据集构建。 如图 2(a) 所示，我们首先从 IR-OptSet（Yang 等人， 2025年）中收集多样化的中间表示程序。IR-OptSet 是一个面向中间表示的数据集，旨在触发真实编译器优化场景中具有代表性的优化变换。我们使用带有 -O3标志的 LLVM 19.1.0（LLVM， 2026a）编译收集到的程序，从而获得成对的未优化和优化后的中间表示程序 ($P^{\text{未优化}}$ P^{opt})，其中 -O3 可启用广泛的优化遍次。随后，我们利用 GPT-5（OpenAI， 2025年）根据观察到的端到端优化行为 P^{unopt}到 P^{opt} 来推断底层优化策略 S，从而为每对中间表示推导出优化意图。最终得到的三元组——未优化的中间表示 P^{unopt}、优化后的 IR P^{opt}以及推断出的策略 S >——构成训练数据集 $\mathcal{D}_{\text{train}}$。

模型训练。我们在Cummins 等人， 2025年提出的 $\mathcal{D}_{\text{train}}$ 上，对用于中间表示优化的语言模型 LLM Compiler FTD 13B（ $\mathcal{F}_\theta$）进行微调，以获得优化专用大语言模型 S。训练目标是最小化在未优化 IR 程序 P^{unopt}条件下生成真实优化策略 P^{unopt}的负对数似然：

$$\mathcal{L}(\theta) = -\mathbb{E}_{(P^{\text{unopt}},S)\sim\mathcal{D}_{\text{train}}}\left[\log p_\theta(S \mid P^{\text{unopt}})\right]$$

选择130亿参数模型是经过深思熟虑的：意图制定主要是对程序结构和编译器优化行为进行模式识别，这并不需要后续细化和实现阶段所需的强大推理能力。这一设计兼顾了效果与计算效率。

3.2. 意图细化

意图形成阶段会生成初始优化策略，明确要应用哪些变换以及应用顺序。尽管该策略捕捉了从编译器行为中学到的高层优化意图，但它是通过数据驱动的方式推断得出的，可能会忽略程序语义中的关键约束或特定于目标的要求。因此，意图细化阶段将初始策略置于具体的编译器分析中加以落实，并在必要时对其进行修订，以确保每项拟议的变换都适用且有充分依据。

意图细化的一项关键挑战是确定细化给定策略S所需的编译器分析。由于不同的变换依赖于不同类型的分析，检索所有可用的分析既低效又没有必要。

为应对这一挑战，我们设计了一种分析感知检索机制，明确捕捉变换遍与其所需分析之间的关系。通过构建结构化编译器知识库，并利用它仅检索与 S中每个变换操作相关的分析，细化过程便能专注于信息量最大的分析信号，同时避免不必要的开销。

编译器知识库构建。 我们构建了一个结构化知识库 $\mathcal{K}$，用于收录编译器优化遍次（即变换遍）及其分析依赖关系。每个条目 $k \in \mathcal{K}$ 表示为：

$$k = (ID, Desc, Deps),$$

其中， ID 唯一标识一个变换遍（对应其遍类名称，例如 LoopVectorizePass）， $Desc$ 提供该遍功能和优化目标的自然语言描述， $Deps$ 表示该遍所需的编译器分析集合。

我们通过从 LLVM 编译器基础设施中自动提取信息来构建 $\mathcal{K}$ 。首先，我们解析 llvm/lib/Passes/PassRegistry.def（其中列举了所有变换遍），以获取遍标识符集合 ID。然后，对于每个遍，我们从 LLVM 官方遍文档中提取其功能描述[1]，以填充相应的 $Desc$字段。

为提取分析依赖关系 $Deps$，我们利用 LLVM 的一项关键设计原则：每个变换通道都必须在其 run 方法中调用 AnalysisManager::getResult，明确声明其所依赖的分析通道。因此，LLVM 提供了一系列分析通道，每个通道都负责从中间表示程序中提取特定的优化相关信息，以指导变换决策。这种显式依赖机制使我们能够系统地追踪每个变换通道所需的分析通道，从而可靠、完整地明确其分析需求。例如，LoopVectorizePass 会按如下方式请求循环结构分析和目标特定分析：

保留的分析 `LoopVectorizePass::run(...,AnalysisManager&AM) { ...`

[1] https://llvm.org/docs/Passes.html

```
auto&LI = AM.getResult<循环分析>(F); auto&TLI = AM.getResult<TargetLibraryAnalysis>(F); ...}
```

因此，我们通过在 $Deps$ llvm/lib/Transforms 下定位每个优化遍次的运行实现并静态解析所有AnalysisManager::getResult调用，得到了该字段。如图 2(b) {所示，LoopVectorizePass的依赖集合为} LoopAnalysis、TargetLibraryAnalysis。

检索分析结果。借助构建的编译器知识库 $\mathcal{K}$，我们检索与初始优化策略中的每项变换操作相对应的具体编译器变换通道及其依赖的分析通道 a_i。 S具体而言，对于每项操作 a_i，我们将其 a_i 自然语言描述作为查询，并与文档语料库进行匹配 $\{Desc(k)\}_{k\in\mathcal{K}}$。查询和文档均采用词频-逆文档频率（TF-IDF，Zhang 等人，2011）表示，并使用 1–3 元语法分词；根据余弦相似度选取得分最高的 m 个匹配优化遍次。

$$\text{sim}(a_i, k) = \cos(\text{TF-IDF}(a_i), \text{TF-IDF}(Desc(k)))$$
$$\{k_i^{(j)}\}_{j=1}^{m} = \arg\text{top-}m_{k\in\mathcal{K}}\text{sim}(a_i, k)$$

这种设计兼顾了效率与覆盖范围：检索得分最高的 m 个结果能够涵盖多个优化遍次实现相似目标的情况（例如，EarlyCSEPass、DCEPass 和 GVNPass 均执行冗余消除），同时避免不加区分地执行分析所带来的计算开销。

确定优化遍次 $\{k_i^{(j)}\}_{j=1}^{m}$ 后，我们从知识库中检索其所需的分析通道 依赖项($k_i^{(j)}$)，并调用 LLVM 编译器，在当前 IR 程序 P 上生成相应的分析结果：

$$R_S = \bigcup_{a_i\in S}\bigcup_{j=1}^{m}\text{LLVM_Generate}(Deps(k_i^{(j)}), P)$$

分析结果通过 LLVM 的分析检查接口（例如 opt -p=print<analysis>）获取，这些接口会以文本形式生成分析信息。这些信息对应于变换遍在编译器优化过程中为确保正确性和性能而查询的相同分析结果。因此，它们为后续的意图细化和意图实现提供了可靠的信息。

细化。 给定初始优化策略 S 和检索到的分析结果 R_S，细化阶段会生成改进策略 S:

$$\hat{S} = \mathcal{G}(P, S, R_S)$$

其中， $\mathcal{G}$ 表示通过对程序、初始策略和编译器分析信息进行推理来执行意图细化的大语言模型。细化过程中使用的提示模板见附录 B。

这一细化过程以编译器分析信息为依据，对第 3.1 节中的初始策略进行验证和增强：它根据程序约束筛除不可行的变换，同时利用分析信息和大语言模型的预训练知识，以超越传统编译器启发式 S 的方式改进这些变换。最终形成的策略具备约束意识，并且有分析信息作为依据，从而在意图制定与具体意图实现之间提供可靠的桥梁。

3.3. 意图实现

ˆ 意图实现阶段最终将策略 S 应用于未优化的中间表示程序 P，以生成优化后的中间表示程序 P:

$$\hat{P} = \mathcal{G}(P, \hat{S}, R_S)$$

其中 $\mathcal{G}$ 表示细化阶段使用的同一个语言模型（GPT-5）。此阶段通过应用ˆ中指定的变换操作 S，同时遵循分析结果中编码的约束 R_S。提示模板见附录 B。

通过这一由制定、细化和实现组成的三阶段流水线，INTOPT 将高层优化意图转化为具体且有充分依据的中间表示变换。借助 GPT-5 进行意图细化和实现，INTOPT 能够探索传统固定遍历编译器通常无法生成的优化策略。

4. 实验

T本节探讨以下研究问题 化：

• **研究问题 1**我能否通过意图驱动的中间表示优化超越现有端到端大语言模型中间表示优化方法？（第 4.2 节）

• **研究问题 2**意图驱动的中间表示优化如何与传统编译器相比，其性能如何？显式优化意图是否能带来超越逐遍优化范式的性能提升？（第 4.3 节 和 第 4.4 节）

• **研究问题 3** 意图驱动的中间表示优化的每个组成部分如何影响其整体结果？（第 4.5 节）

4.1. 实验设置

4.1.1. 数据集

我们从 IR-OptSet（Yang 等人，2025年）构建所有数据集。IR-OptSet 是一个面向中间表示的数据集，旨在触发真实编译器优化场景中常见的代表性优化变换；每个 IR 样本（未优化的中间表示加上优化后的 IR）最多包含 5,000 个词元。我们随机抽取 4,000 个程序组成训练集 $\mathbf{INT}_{\mathbf{train}}$

表 1. 六种基于大语言模型的 IR 优化方法在正确性和性能方面的比较。正确性通过结合使用 Alive2 与差分测试进行评估，性能则以相对于输入的未优化中间表示的加速比衡量。

方法	正确性		性能			
	Alive2	存活2+差分测试	平均加速比	加速比 $> 1.1\times$	加速比 $> 1.5\times$	加速比 $> 2.0\times$
GPT-5	**73.0%** (146)	83.5% (167)	1.057×	26.0% (52)	10.5% (21)	6.5% (13)
Claude Haiku 4.5	54.0% (108)	61.0% (122)	0.721×	15.5% (31)	5.0% (10)	2.5% (5)
DeepSeek-V3.2	51.0% (102)	59.0% (118)	0.719×	14.5% (29)	6.5% (13)	3.5% (7)
LLM Compiler FTD 7B	57.0% (114)	60.5%（121）	0.702×	13.0%（26）	6.0%（12）	2.5%（5）
LLM Compiler FTD130亿参数	61.5%（123）	64.5%（129）	0.757×	12.0%（24）	4.0%（8）	2.5%（5）
INTOPT	71.0% (142)	**90.5%** (181)	**2.660**×	**37.5%** (75)	**17.5%** (35)	**10.0%** (20)

，并抽取 500 个程序组成验证集 $\mathbf{INT}_{\mathbf{val}}$。为进行评估，我们通过多阶段筛选流程构建了一个独立的测试集 $\mathbf{INT}_{\mathbf{test}}$，其中包含 200 个 IR 样本。首先，保留 Alive2（Lopes 等人，2021）能够验证未优化的中间表示与 LLVM `-O3`之间语义等价性的程序。其次，排除不适用于差分测试（McKeeman，1998）的程序，例如包含自定义外部函数、外部定义的全局变量或非确定性行为的程序。这样可以确保当基于大语言模型的优化器生成超出 Alive2 验证能力的变换时，仍可可靠地使用差分测试验证其正确性。

4.1.2. 基线

我们纳入两类基线进行比较：

端到端的基于大语言模型的 IR 优化器。我们同时考察通用型和特化型的基于大语言模型的优化器，它们可以直接将未优化的中间表示映射为优化后的 IR。具体而言，我们评估了三种在代码相关任务上能力出众的通用大语言模型：***GPT-5*** (OpenAI，2025年）、***Claude Haiku 4.5***(Anthropic，2025年），以及***DeepSeek-V3.2*** (DeepSeek-AI 等人，2025年）。所有模型均使用附录 C所示的模板进行提示。我们还与 LLM Compiler（Cummins 等人，2025年）进行比较。它是最先进的特化型端到端 IR 优化器；我们评估了其 70亿参数和 130亿参数两个版本（***LLM CompilerFTD 7B***和***LLM Compiler FTD13B***）。为确保比较公平，两个 LLM Compiler 版本均使用 $\text{INT}_{\text{train}}$ 和 INT_{val} 按照附录 E进行微调。

编译器基线。我们采用 ***LLVM 19.1.0*** (LLVM，2026a) 的***-O3*** 优化级别作为编译器级别的参照。`-O3` 会积极针对执行速度进行优化（LLVM，2026c），代表着高度调优的中间表示优化策略。

4.1.3. 实现细节

对于 IntOpt 的意图形成阶段，我们在 I $\mathcal{F}_\theta$ NToPT 中获得优化专用大语言模型，使用 $\text{INT}_{\text{train}}$ 和 INT_{val}，并遵循第 3.1 节中描述的流程。在分析结果检索阶段，我们选取匹配度最高的 3 个优化遍次，并收集其对应的 LLVM 分析结果，用于支持意图细化阶段。

4.1.4. 评估指标

正确性验证。我们采用两阶段验证流程来检验优化后的中间表示程序的正确性。首先，我们使用 Alive2 对优化后的中间表示和未优化的中间表示进行严格的形式化语义等价检查。由于 Alive2 在验证控制流或内存行为复杂的程序时可能超时（Lopes 等人，2021），当形式化验证不可行时，我们还会进行差分测试。具体而言，我们使用 LibFuzzer（LLVM，2026b）生成多样化的测试输入，并验证优化后的程序和未优化的程序在不同执行路径下是否产生相同的输出。详情见附录 F。

性能评估。我们使用正确性验证期间由 LibFuzzer 生成的相同输入来测量性能，以反映程序在各种控制流路径上的执行行为。每个程序执行 10,000 次迭代，以获得稳定的计时结果并减少测量噪声。我们将 性能 报告为加速比：加速比 =未优化运行时间/优化后运行时间。当基于大语言模型的优化器生成不正确的中间表示时，我们将加速比设为 0。更多详情见 附录 G。

4.2. 相较于基于大语言模型的中间表示优化器的改进

我们首先在包含 200 个样本的基准测试 INT_{test} 上，将 INTOPT 与端到端的基于大语言模型的 IR 优化器进行比较。如表 1所示，在正确性和性能方面，INTOPT 的整体表现均优于所有基线方法。

正确性。在仅使用 Alive2 进行验证时，GPT-5 在基线方法中的正确率最高（73.0%），而 INTOPT 的正确率也达到相近的 71.0%。然而，这一指标本质上受限于 Alive2 的验证能力：即使是 LLVM -O3 生成的复杂优化，也不一定总能通过验证（Lopes 等人，2021）。纳入差分测试后，INTOPT 的已验证正确率达到 90.5%，比所有其他基线方法高出 7.0%–31.5%。这些结果凸显了 INTOPT 在减少语义错误方面相较于端到端的基于大语言模型的中间表示优化方法的优势。

性能。INTOPT 相较于未优化的中间表示实现了 2.660× 的平均加速比，显著优于所有基于大语言模型的端到端基线方法。表现最强的基线方法

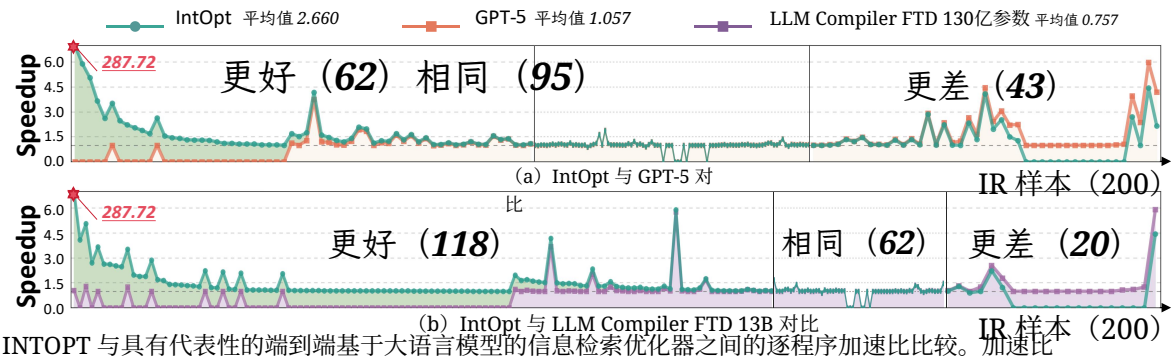

图 3. INTOPT 与具有代表性的端到端基于大语言模型的信息检索优化器之间的逐程序加速比比较。加速比差异在 ±2% 以内时视为相同，以考虑运行时波动。

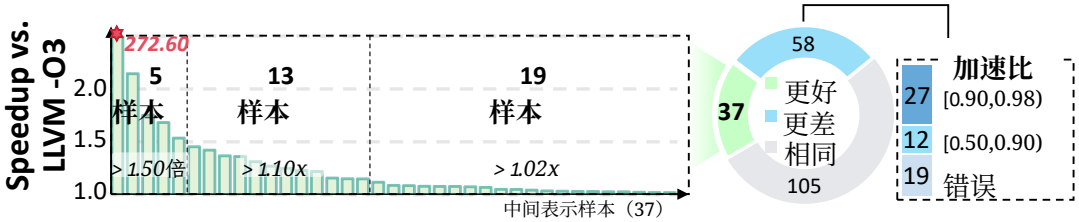

图 4. INTOPT 与 LLVM -O3 的性能比较。加速比定义为 RunTime(LLVM -O3)/RunTime(opt)，其中 $[0.98, 1.02]$ 视为相同。INTOPT 在 105 个程序上与 LLVM -O3 表现相同，在 37 个程序上优于它（最高为 272.60×）；在表现较差的 58 个案例中，大多数只是略微变慢（$[0.90, 0.98]$ 中有 27 个），另有 19 个结果不正确。

这一类模型 GPT-5 的平均加速比仅为 1.057×，导致 1.603× 的性能差距。除了平均性能之外，INTOPT 在所有加速比阈值上也始终占据优势，这表明其性能提升广泛分布于整个基准测试中。

为了更好地理解这些改进，我们进一步逐个程序比较 IntOpt 与两个最强基线 GPT-5 和 LLM Compiler FTD 13B 的表现，如图 3所示。与 GPT-5 相比，INTOPT 在 62 个程序上取得了更高的加速比；而与 LLM Compiler FTD 13B 相比，优势更为明显，INTOPT 在 118 个程序上胜过它。值得注意的是，与未优化的中间表示相比，INTOPT 在单个程序上取得的最高加速比可达 287.72× 。这些结果表明，INTOPT 更善于发现并实现激进且有利可图的优化机会。

对 ***RQ.1*** 的回答：通过将优化意图的制定与具体变换生成解耦，INTOPT 减少了语义错误，并且比现有的端到端大语言模型中间表示优化方法更有效地发现有利可图的优化机会。

4.3. 相较于传统编译器的改进

我们进一步将 IntOpt的性能与 LLVM-O3进行比较；-O3 是一种高度优化且广泛采用的编译器优化级别。图 4展示了比较结果。

INTOPT 在 105 个程序上的性能与 LLVM -O3 相当，并在 **37** 个程序上优于 LLVM -O3。在这些性能提升的案例中，各程序的加速比分布展现出不同程度的性能增益：5 个程序的性能显著提升，加速比大于 1.5×；另有 13 个程序取得了明显提升，加速比超过 1.1×。值得注意的是，图 3中加速比最高的程序，相较于 LLVM -O3 的性能提升也最大，加速比高达 272.60×。我们在第 4.4 节中详细分析了这一案例，展示 INTOPT 如何发掘超越传统编译器能力的优化机会。

4.4. 案例分析

为了深入了解意图驱动的优化机制，我们展示了两个具有代表性的案例，说明 IntOpt 如何探索超越传统逐遍编译器范式的优化机会。受篇幅限制，我们给出中间表示的伪代码表示；完整的代码清单见附录 H。如图 5所示，传统编译器主要通过匹配由编译器开发人员设计的优化遍次中预定义的模式，并应用变换（例如，消除基于栈的加载/存储操作）来优化程序。

相比之下，INTOPT 执行意图引导优化，从而实现协同的语义感知变换。在 chocolateFeast 中，INTOPT 在意图细化过程中识别出一种算法捷径，将迭代计算的时间复杂度降至 $O(1)$，并指导后续优化，实现了 287.72× 加速比。在反转比特中，INTOPT 通过协调多项变换实现了 1.97× 加速比。在将循环的高层语义识别为比特反转之后，INTOPT 将原始计算替换为

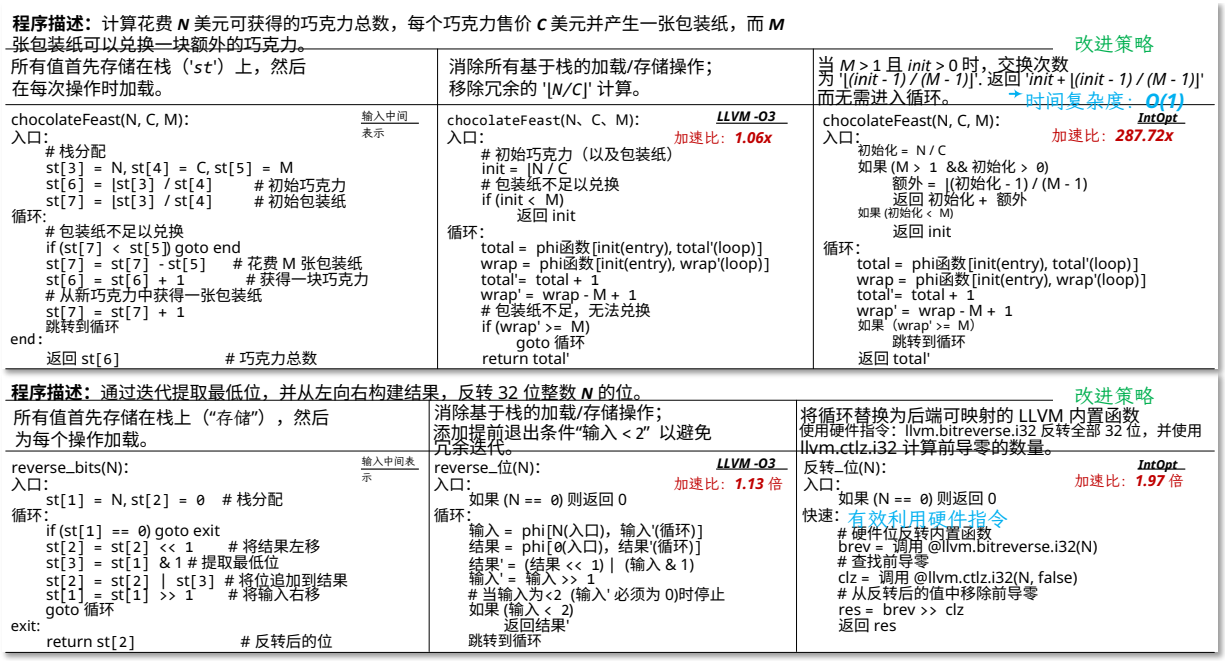

图 5. 案例研究展示 INTOPT 如何超越传统的逐遍编译器范式来探索优化机会。

表 2. 对 INTOPT 设计的消融研究。意图细化和编译器分析对正确性和性能的影响，其中性能以相对于输入的未优化中间表示的加速比衡量。

方法	正确性		性能			
	Alive2	Alive2+差分测试	平均加速比	加速比 $> 1.1\times$	加速比 $> 1.5\times$	加速比 $> 2.0\times$
INTOPT	71.0% (142)	**90.5%** (181)	**2.660**×	**37.5%** (75)	**17.5%** (35)	**10.0%** (20)
不进行意图细化	**71.5%**(143)	87.0% (174)	1.122×	30.0% (60)	15.0% (30)	7.5% (15)
无分析	65.0% (130)	77.0% (154)	2.415×	29.0% (58)	13.0% (26)	6.5% (13)

llvm.bitreverse 内置函数，并进一步应用一项使用 llvm.ctlz 与基于移位的操作的辅助变换，以对原始循环的退出条件进行建模。现有编译器很难实现这些优化，因为它们所需的语义推理超出了固定逐遍流水线中局部模式匹配所能捕获的范围。

研究问题 **2** 的回答：显式的优化意图使 INTOPT 能够实现超越传统逐遍编译器能力的优化。

4.5. INTOPT 设计的有效性

意图形成阶段是该流程中不可或缺的第一阶段，而意图实现则是生成优化后的中间表示的最后阶段；两者均不可跳过。因此，我们的消融研究聚焦于意图细化阶段的作用，以及编译器分析对意图驱动优化的影响。如表 2所示，移除意图细化会略微提高仅使用 Alive2 时的正确性，但会明显降低已验证正确性和性能，使平均加速比从 $2.660\times$ 降至$1.122\times$。从细化和实现两个阶段中移除编译器分析（无分析）会进一步将已验证正确性降至 77.0%，因为细化和实现决策是在缺乏充分语义约束的情况下作出的。尽管如此，性能仍然相对较高（平均$2.415\times$ ）。这可能是因为，以意图形成阶段生成的、源自编译器的优化意图为起点，大语言模型有时能够识别出非常激进的变换，包括图 3 所示、实现了 287.72×加速比的情况。

研究问题 **3** 的回答：意图细化与编译器分析对 INTOPT 都至关重要，使其能够有效发现优化并可靠地加以实现。

5. 结论

本文介绍了 INTOPT，这是首个意图驱动的中间表示优化框架。通过将高层优化意图与底层分析和变换解耦，INTOPT 解决了中间表示优化中一个至关重要却尚未得到充分探索的维度。实验结果表明，INTOPT 的表现显著优于端到端的基于大语言模型的优化器和传统编译器优化，这说明，明确表达优化意图对于实现全局协调、语义可靠且高性能的中间表示优化至关重要。

影响声明

本研究探讨了利用大语言模型辅助编译器中的中间表示优化；在这一领域，正确性、可靠性和性能至关重要。通过对优化意图进行显式建模，并将其与底层变换分离，所提出的框架提升了大语言模型辅助编译器优化的可解释性与可检查性，使开发者能够更好地理解并验证这些系统生成的优化策略。我们认为，意图驱动优化为编译器与机器学习交叉领域的未来研究奠定了有前景的基础，有助于推动更可信、更可解释的人工智能辅助程序优化。

参考文献

Aho, A.V.,Lam, M.S., Sethi,R., and Ullman, J. D. 编译器：原理、技术与工具（第 *2* 版）. Addison-Wesley Longman Publishing Co., Inc., USA, 2006年. ISBN 0321486811.

Ansel, J., Kamil, S., Veeramachaneni, K., Ragan-Kelley, J., Bosboom, J., O'Reilly, U.-M., and Amarasinghe, S. Opentuner：一种可扩展的程序自动调优框架。载于*2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*，第 303–315 页，2014年。doi: 10.1145/2628071.2628092。

Anthropic。ClaudeHaiku 4.5。https://www.anthropic.com/claude/haiku，2025年。

Bezanson, J.、Edelman, A.、Karpinski, S.与Shah, V. B. Julia：数值计算的全新方法。 *SIAMRev.*，59(1):65–98，2017年1月。ISSN 0036-1445。doi: 10.1137/141000671。URLhttps://doi.org/10.1137/141000671。

Chen, J.、Xu, N.、Chen, P.与Zhang, H. 基于贝叶斯优化的高效编译器自动调优。载于*2021 IEEE/ACM*第*43*届软件工程国际会议（*ICSE*），第1198–1209页，2021。doi: 10.1109/ICSE43902. 2021.00110。

Chen, T.、Moreau, T.、Jiang, Z.、Zheng, L.、Yan, E.、Cowan, M.、Shen, H.、Wang, L.、Hu, Y.、Ceze, L.、Guestrin, C.与Krishnamurthy, A. TVM：面向深度学习的自动化端到端优化编译器。载于第*13*届*USENIX*操作系统设计与实现会议论文集，OSDI'18，第579–594页，美国，2018年。USENIX协会。ISBN 9781931971478。

Cummins, C., Seeker, V., Grubisic, D., Roziere, B., Gehring, J., Synnaeve, G., 和 Leather, H. Llm compiler：面向编译器优化的基础语言模型。载于第*34*届 *ACM SIGPLAN* 国际编译器构造会议论文集，第 141–153 页，2025年。

Daya, G., Dejian, Y., 等，Z. H. DeepSeek-R1 通过强化学习激励大语言模型进行推理。《自然》，645(8081)：633–638，2025年9月。doi：https://doi.org/10.1038/s41586-025-09422-z。DeepSeek-AI，Liu, A.，等，A. M. Deepseek-v3.2：推动开放大语言模型的前沿，2025年。URL https://arxiv.org/abs/2512.02556。GCC。GNU编译器套件。https://gcc.gnu.org，2026年。

Kulkarni, P., Whalley, D., Tyson, G. 和 Davidson, J. 穷举优化阶段顺序空间探索。载于代码生成与优化国际研讨会（*CGO'06*），第 13–318 页，2006年。doi：10.1109/CGO.2006.15。

Lattner, C. 和 Adve, V. LLVM：用于终身程序分析与变换的编译框架。载于代码生成与优化国际研讨会，*2004*。*CGO 2004*。，第 75–86 页，2004年。doi：10.1109/CGO.2004.1281665。

Lattner, C., Amini, M., Bondhugula, U., Cohen, A., Davis, A., Pienaar, J., Riddle, R., Shpeisman, T., Vasilache, N. 和 Zinenko, O. MLIR：为领域特定计算扩展编译器基础设施。载于*2021 IEEE/ACM* 代码生成与优化国际研讨会（*CGO*），第 2–14 页，2021。doi：10.1109/CGO51591.2021.9370308。

Liang, Y.、Stone, K.、Shameli, A.、Cummins, C.、Elhoushi, M.、Guo, J.、Steiner, B.、Yang, X.、Xie, P.、Leather, H. 和 Tian, Y. 使用核心集和归一化值预测学习编译器遍顺序。载于第 *40* 届国际机器学习会议论文集，ICML'23。JMLR.org，2023 年。

LLVM。LLVM 编译器基础设施项目。http://llvm.org/，2026a。

LLVM。libFuzzer——用于覆盖率引导模糊测试的库，2026b。网址：https://llvm.org/docs/LibFuzzer.html。

LLVM。OptimizationLevel，2026c。网址：https://llvm.org/docs/doxygen/classllvm_1_1OptimizationLevel.html。

Lopes, N. P., Lee, J., Hur, C.-K., Liu, Z. 和 Regehr, J. Alive2：面向 LLVM 的有界翻译验证。载于 第*42*届 *ACM SIGPLAN* 编程语言设计与实现国际会议论文集，PLDI 2021，第65–79页，美国纽约州纽约市，2021。计算机协会。ISBN 9781450383912。doi: 10.1145/ 3453483.3454030。URLhttps://doi.org/10.1145/3453483.3454030。

McKeeman, W. M. 软件差分测试。*Digit.Tech. J.*，10:100–107，1998。URLhttps://api.semanticscholar.org/CorpusID:14018070。

Moses, W. S., Ivanov, I. R., Domke, J., Endo, T., Doerfert, J., and Zinenko, O. 通过高级并行结构实现高性能 GPU 到 CPU 转译与优化。发表于第*28*届 *ACM SIGPLAN* 并行编程原理与实践年度研讨会论文集，PPoPP '23，第119–134页，美国纽约州纽约市，2023年。计算机协会。ISBN 9798400700156。doi: 10.1145/3572848.3577475。网址 https://doi.org/10.1145/3572848.3577475。Muchnick, S. S. 高级编译器设计与实现。Morgan Kaufmann Publishers Inc.，美国加利福尼亚州旧金山，1998年。ISBN 1558603204。OpenAI。介绍 GPT-5。https://openai.com/index/introducing-gpt-5/，2025年。Pallister, J., Hollis, S. J., and Bennett, J. 找出可最大限度降低嵌入式平台能耗的编译器选项。*The Computer Journal*，58(1):95–109，2015年。doi: 10.1093/comjnl/bxt129。Pan, H., Wei, Y., Xing, M., Wu, Y., and Zhao, C. 迈向高效的编译器自动调优：利用协同搜索空间。发表于第*23*届 *ACM/IEEE* 国际代码生成与优化研讨会论文集，CGO '25，第614–627页，美国纽约州纽约市，2025年。计算机协会。ISBN 9798400712753。doi: 10.1145/ 3696443.3708961。网址 https://doi.org/10.1145/3696443.3708961。Park, S., Latifi, S., Park, Y., Behroozi, A., Jeon, B., and Mahlke, S. Srtuner：通过揭示协同关系实现有效的编译器优化定制。发表于*2022*年 *IEEE/ACM* 代码生成与优化国际研讨会（*CGO*），第118–130页，2022年。doi: 10.1109/CGO53902.2022.9741263。Sabne, A. Xla：编译机器学习以获得峰值性能，2020年。Stanier, J. and Watson, D. 命令式编译器中的中间表示：综述。*ACM* 计算调查，45(3)，2013年7月。ISSN 0360-0300。doi: 10. 1145/2480741.2480743。网址 https://doi.org/10.1145/2480741.2480743。Taneja, J., Laird, A., Yan, C., Musuvathi, M., and Lahiri, S. K. Llm-vectorizer：基于大语言模型的经验证循环向量化器。发表于第*23*届 *ACM/IEEE* 国际代码生成与优化研讨会论文集，CGO '25，第 137–149 页，美国纽约州纽约市，2025年。计算机协会。ISBN 9798400712753。doi: 10.1145/3696443.3708929。URLhttps://doi.org/10.1145/3696443.3708929。Yang, Z., Qiu, L., Lyu, Fang Zhong, M., Chai, Z., Zhou, H., Cui, H., and Feng, X. Ir-optset：用于推动大语言模型信息检索优化器发展的优化敏感型数据集。神经信息处理系统进展，38，2025年。Zhang, W., Yoshida, T., and Tang, X. 词频-逆文档频率、潜在语义索引和多词在文本分类中的比较研究。专家系统与应用，38(3):2758–2765，2011年。ISSN 0957-4174。doi: https://doi.org/10.1016/j.eswa.2010.08. 066。URL https://www.sciencedirect.com/science/article/pii/S0957417410008626。Zhong, M., Lyu, F., Wang, L., Geng, H., Qiu, L., Cui, H., and Feng, X. Comback：用于提升编译器后端开发效率的通用数据集。神经信息处理系统进展，37:112310–112328，2024年。Zhong, M., Lv, F., Wang, L., Qiu, L., Wang, Y., Liu, Y., Cui, H., Feng, X., and Xue, J. VEGA：使用预训练变换器模型自动生成编译器后端。载于第*23*届 *ACM/IEEE* 国际代码生成与优化研讨会论文集，CGO '25，第 90–106 页，美国纽约州纽约市，2025a。计算机协会。ISBN 9798400712753。doi: 10.1145/3696443.3708931。URLhttps://doi.org/10.1145/3696443.3708931。Zhong, M., Sun, X., Lv, F., Wang, L., Geng, H., Qiu, L., Cui, H., and Feng, X. BePilot：用于编译器后端开发的人工智能编程助手。*ACM* 软件工程与方法学汇刊，2025b。ISSN 1049-331X。doi: 10.1145/3764585。URLhttps://doi.org/10.1145/3764585。已接收待刊。Zhu, M., Hao, D., and Chen, J. 通过多阶段学习实现编译器自动调优。*ACM* 软件工程与方法学汇刊，33(4)，2024年4月。ISSN 1049-331X。doi: 10.1145/3640330。URLhttps://doi.org/10.1145/3640330。¨Ozen, G., Atzeni, S., Wolfe, M., Southwell, A., and Klimow- icz. Flang 和 LLVM 中的 OpenMP GPU 卸载。载于*2018*年 *IEEE/ACM* 第五届 *LLVM* 编译器基础设施 *HPC* 研讨会（*LLVM-HPC*），第 1–9 页，2018年。doi: 10.1109/LLVM-HPC.2018.8639434。

A. 编译器分析信息示例

本节将通过一个详细示例说明编译器分析信息如何引导输入 IR 程序的优化过程，如图 6所示。其中包含两种类型的分支：(1) 无条件分支（例如，`br label %B1`），这类分支无需任何条件即可直接跳转到另一个基本块；(2) 条件分支（`br i1 %16,label %B2, label %B3`），这类分支根据条件执行跳转。

```
define dso_local 32 位整数 @chocolateFeast(32 位整数 noundef %0 , 32 位整数 noundef %1 , 32 位整数 noundef %2){
B0:
%3 = alloca 32 位整数, align 4
%4 = 分配 32 位整数, 对齐 4
%5 = 分配 32 位整数, 对齐 4
%6 = 分配 32 位整数, 对齐 4
%7 = 分配 32 位整数, 对齐 4
存储 32 位整数 %0, 指针 %3, 对齐 4, !基于类型的别名分析 !5
存储 32 位整数 %1, 指针 %4, 对齐 4, !基于类型的别名分析 !5
存储 32 位整数 %2, 指针 %5, 对齐 4, !tbaa !5
调用 void @LLVM 生命周期开始(i64 4, 指针 %6) #2
%8 =加载 32 位整数, 指针 %3, 对齐 4, !tbaa !5
%9 =加载 32 位整数, 指针 %4, 对齐 4, !基于类型的别
%10 =有符号除法 32 位整数 %8 , %9
存储 32 位整数 %10, 指针 %6, 对齐 4, !基于类型的别名分析 !5
call void @LLVM 生命周期开始(i64 4, 指针 %7) #2
%11 = 加载 32 位整数, 指针 %3, 对齐 4, !基于类型的别名分析 !5
%12 = 加载 32 位整数, 指针 %4, 对齐 4, !基于类型的别
%13 =有符号除法 32 位整数 %11 , %12
存储 32 位整数 %13, 指针 %7, 对齐 4, !基于类型的别
分支标签 %B1
B1:
%14 = 加载 32 位整数, 指针 %5, 对齐 4, !基于类型的别名分析 !5
%15 = 加载 32 位整数, 指针 %7, 对齐 4, !基于类型的别
%16 =icmp sle 32 位整数 %14, %15
br i1 %16 , label %B2 , label %B3
B2:
%17 = 加载 32 位整数, 指针 %7, 对齐 4, !基于类型的别名分析 !5
%18 = 加载 32 位整数, 指针 %5, 对齐 4, !基于类型的别
%19 =sub nsw 32 位整数 %17 , %18
存储 32 位整数 %19, 指针 %7, 对齐 4, !基于类型的别名分析 !5
%20 = 加载 32 位整数, 指针 %6, 对齐 4, !基于类型的别
%21 =加法 nsw 32 位整数 %20 , 1
存储 32 位整数 %21, 指针 %6, 对齐 4, !基于类型的别名分析 !5
%22 = 加载 32 位整数, 指针 %7, 对齐 4, !基于类型的别
%23 =加法 nsw 32 位整数 %22 , 1
存储 32 位整数 %23, 指针 %7, 对齐 4, !基于类型的别
分支标签 %B1, !llvm.循环 !9
B3:
%24 = 加载 32 位整数, 指针 %6, 对齐 4, !基于类型的别名分析 !5
调用 void @llvm.lifetime.end.p0(i64 4, 指针 %7) #2
调用 void @llvm.lifetime.end.p0(i64 4, 指针 %6) #2
ret 32 位整数 %24
}
```

图*6*。输入中间表示示例

下面，我们将详细介绍分析的关键要素，并解释各部分的含义。

A.1. 支配树分析

支配树是一种数据结构，用于识别函数控制流图（控制流图）中基本块之间的支配关系。如果从入口块到 C 的每条路径都必须经过 B，则块 B 支配另一个块 C。支配树通过突出哪些块对于

执行至关重要，以及哪些块可能是冗余的或适合优化。对于函数 chocolateFeast，支配树如下所示：

```
函数 chocolateFeast 的支配树==============================----------
```

• %B0 位于第 1 层，这表明它是入口块（函数的起始点）。作为支配树的根，%B0 支配所有其他块。它是调用该函数时最先执行的块，其他每个块都必须经过它。

• %B1 位于第 2 层，意味着它在控制流中比 %B0 深一层。该块由 %B0 支配，并且是 %B0 之后最先执行的块。

• %B2 和 %B3 都位于第 3 层，意味着它们在控制流中比 %B0 深两层。这些块由 %B1 支配，意味着它们仅在 %B1 之后执行。

A.2. 循环分析

循环是优化的重要目标，因为它们往往是性能瓶颈的来源。循环分析可以识别函数中的循环，并提供其结构的详细信息，例如循环深度和涉及的基本块。在循环结构中，循环头是循环中的第一个基本块，所有来自循环外部的边都会汇聚于此。循环回边是循环内部的一个节点，它有一条边指向循环头。退出基本块是控制流离开循环的基本块，标志着循环的终止点。函数 chocolateFeast 的循环分析如下所示：

```
函数 'chocolateFeast' 的循环信息：深度为 1 的循环包含：%B1<header><exiting>,%
```

• 该循环的深度为 1，意味着它是函数中的顶层循环。

• 该循环涉及 %B1 作为循环头和退出基本块，%B2 作为循环回边。

B. INTOPT 的提示模板

B.1. 意图表述提示模板

INTOPT 中的意图制定阶段会生成初始优化策略 S，它是针对给定输入 IR 程序 P 的优化意图的结构化实例。为了训练微调后的 LLM 编译器 FTD 13B，我们使用以下模板，引导其生成一系列有序的自然语言变换操作：

```
[INST]给定以下 LLVM IR，提出关键的优化转换步骤，以超越 LLVM -O3。将答案写在单个 <code>...</
```

B.2. 意图细化提示模板

意图细化阶段接收初始优化策略 S，该策略为给定的输入 IR 程序 P 指定一系列变换，并通过将策略落实到具体的编译器分析结果 R_S 中来对其进行细化。意图细化阶段所使用的模板如下所示：

```
请优化以下代码，使其性能优于 LLVM -O3。<code>（未优化的 LLVM IR）</code>你可以参考以下建议，但也可根据需
```

B.3. 意图实现提示模板

意图实现阶段采用改进后的优化策略 S，并将其应用于未优化 IR 程序 P，以生成优化后的 IR 程序 P。意图实现阶段所使用的模板如下所示：

```
请优化以下代码，使其性能优于 LLVM -O3。<code>（未优化的 LLVM IR）</code>你可以参考以下建议。<advice>
```

C. 基于大语言模型的优化器提示模板

为了评估通用基于大语言模型的优化器（GPT-5、Claude Haiku 4.5 和 DeepSeek-V3.2）的性能，我们使用以下提示模板来指导优化过程：

```
请优化以下代码，使其性能优于 LLVM -O3。<code>（未优化的 LLVM IR）</code>请保留边界检查。请将完整的优
```

D. 词元统计

表3 展示了$\text{INT}_{\text{train}}$、$\text{INT}_{\text{val}}$和$\text{INT}_{\text{test}}$ 在第4.1节中使用的数据统计信息，包括数量和词元数。具体而言，我们按照第3.1节所述流程，使用INTtrain和INT_{val} 生成了用于优化意图的数据集。

表 *3.* $\text{INT}_{\text{train}}$、$\text{INT}_{\text{val}}$ 和 INT_{test} 的数量与词元数统计。

		总词元数	平均词元数	词元中位数	最小词元数	最大词元数
$\textbf{INT}_{\textbf{train}}$	未优化的中间表示	8,219,999	2,055.0	2,047.0	579	4,304
	优化后的 IR	5,531,194	1,382.8	1,370.5	359	3,373
	优化意图	3,753,392	938.6	933.0	331	1,719
$\textbf{INT}_{\textbf{val}}$	未优化的中间表示	1,017,905	2,035.8	2016.0	685	4,035
	优化后的 IR	691,180	1,382.4	1,359.5	425	3,220
	优化意图	473,842	947.7	941.5	321	1,692
$\textbf{INT}_{\textbf{test}}$	未优化的中间表示	362,455	1,812.3	1,572.5	679	4,188

E. 实验细节

在表4中，我们列出了实验中使用的所有超参数设置，包括对用于中间表示优化的LLM Compiler FTD 70亿参数和130亿参数进行微调，以及对用于意图制定的LLM Compiler FTD 130亿参数进行微调的设置。在本工作中，我们将上下文窗口设为5K，这意味着由于资源限制，输入和输出的总长度不得超过5K。

表*4.* 超参数设置。

模型	LoRA 秩 (r)	LoRA α	LoRA 丢弃率	批次 size	学习 rate	目标模块
LLM Compiler FTD 130亿参数/70亿参数	48	16	0.05	2	1×10^{-4}	{q 投影、k 投影, v-投影、o 投影}

F. 通过差分测试进行正确性验证

差分测试，又称差分模糊测试，是一种软件测试技术，它通过向一系列相似的应用程序（或同一应用程序的不同实现）提供相同的输入，并观察其执行差异来检测缺陷。差分测试是对传统软件测试的补充，因为它非常适合发现那些不会表现出崩溃或断言失败等显式错误行为的语义或逻辑缺陷。在本工作中，我们使用 LibFuzzer 生成覆盖多种控制流路径的各类输入，从而确保对函数行为进行全面测试。

F.1. 面向差分测试的测试工具生成

我们利用 GPT-5 为每个函数对自动生成 LibFuzzer 工具。具体来说，对于每个同时包含基准函数 f 和其优化版本 f opt 的 LLVM IR 文件，我们会提示大语言模型使用以下提示模板合成 C++ 模糊测试工具（fuzz.cc）：

```
你正在生成一个名为 fuzz.cc 的 C++LibFuzzer 模糊测试工具文件。输入：- 一个包含多个函数定义的 LLV
```

为说明这一点，请看下面这个为函数对 numberOfOperations 和 numberOfOperations opt 生成的 C++ 模糊测试工具示例：

```
#include <cstdint>
#include <cstddef>
#include <cstring>

int numberOfOperations(int);
int numberOfOperations_opt(int);

extern "C" int LLVMFuzzerTestOneInput(const uint8_t* data, size_t size) {
    如果（大小 < 4）返回 0;

    int32_t in32;
    std::memcpy(&in32, data, 4);
    int 参数 = static_cast<int>(in32);

    int r_base = numberOfOperations(arg);
    int r_opt = numberOfOperations_opt(arg);

    if (r_base != r_opt) {
        __内置_陷阱();
    }

    return 0;
}
```

函数 LLVMFuzzerTestOneInput 是 LibFuzzer 的入口点，用于接收 LibFuzzer 生成的输入数据。数据首先被解析为一个 32 位整数，然后作为参数分别传递给 numberOfOperations 和 numberOfOperations opt。如果基础函数与优化函数的结果不一致，程序就会使用 builtin trap() 触发陷阱，停止执行并指出存在不一致。

F.2. 差分测试二进制文件的生成与执行

每个生成的工具都会与其对应的 LLVM IR 一起编译并链接，以生成经过插桩的模糊测试二进制文件。编译流程包括：

- **IR 编译**。 由于 llc 在后端运行，此步骤会应用特定于目标的代码生成和后端优化，同时保留不同优化器生成的 IR 级转换。
- **经消毒器检测的链接**。 使用 clang++ 将生成的汇编与模糊测试工具（fuzz.cc）链接，并通过 -fsanitize=fuzzer,address,undefined 启用全面的消毒器插桩。这些消毒器分别用于支持覆盖率引导的输入生成、检测内存安全违规（例如缓冲区溢出和释放后使用），以及识别基础实现与优化实现之间的未定义行为差异。

每个经过插桩的二进制文件都以 200,000 次模糊测试迭代（-runs=200000）执行。执行过程中，LibFuzzer 在代码覆盖率反馈的引导下持续生成多样化的测试输入，以相同的输入调用基准函数和优化函数，并比较它们的行为。基准函数与优化函数之间在输出、崩溃或 sanitizer 违规上的任何差异，都表明存在改变语义的变换，从而使该优化候选项失效。只有成功完成全部 200,000 次运行且未检测到行为差异的函数对，才会被验证为语义等价优化。

G. 性能验证

我们使用 LibFuzzer 在正确性验证期间生成的相同输入来测量性能，从而反映在多样控制流路径上的执行行为。基于这一设计，我们将经过验证的差分模糊测试框架转换为独立的微基准测试，用于性能测量。

具体而言，我们通过一系列源代码级转换，从经过验证的模糊测试框架自动派生微基准测试框架。从一个调用基准函数的差分模糊测试框架开始f 及其优化版本 fopt，我们首先识别相应的函数符号（例如，numberOfOperations 和numberOfOperations opt 在第 F.1 节）。然后，我们将模糊测试入口点 LLVMFuzzerTestOneInput 重命名为标准辅助函数 decode input，该函数继续将原始输入字节解析为具体实参，并对两种实现各执行一次调用。

接下来，我们在 decode 输入中插入细粒度计时插桩。如示例所示，我们使用 clock gettime(CLOCK MONOTONIC RAW) 在每次调用基准函数和优化函数前后立即记录时间戳。执行时间和调用次数会累加到全局计数器中（例如 g t baseline ns、g t opt ns）。为防止死代码消除，我们将返回值的地址折叠到全局 volatile 汇点变量（g sink）中，确保编译器保留这两次函数调用。

最后，我们生成一个独立的 main 函数来驱动基准测试。该驱动程序从模糊测试语料库中读取具体输入，执行固定次数的预热迭代（例如，1,000 次执行，不纳入测量），然后执行大量计时迭代（默认：1,000,000 次）。执行结束后，它会报告基准实现和优化实现的单次调用平均运行时间，以及由此得出的加速比。

所有微基准测试均在 Intel(R) Xeon(R) Gold 6434 处理器上执行，平均而言，每项微基准测试使用模糊测试语料库中的 16.6 个不同输入进行评估。

```
#include <cstdint>
#include <cstddef>
#include <cstring>

#include <time.h>
#include <fstream>
#include <iostream>
#include <vector>
#include <cstdint>

static inline uint64_t now_ns() {
    struct 时间规格结构体 ts;
    clock_gettime(CLOCK_MONOTONIC_RAW, &ts);
    return (uint64_t)ts.tv_sec * 1000000000ull + (uint64_t)ts.tv_nsec;
}

static uint64_t g_t_baseline_ns = 0;
static uint64_t g_t_opt_ns      = 0;
static uint64_t g_n_baseline    = 0;
static uint64_t g_n_opt         = 0;

static volatile uintptr_t g_sink = 0;

int numberOfOperations(int) asm("_Z18numberOfOperationsi");
int numberOfOperations_opt(int) asm("_Z18numberOfOperationsi_opt");
static int decode_输入(const uint8_t* data, size_t size) {
    if (size < 4) return 0;

    int32_t in32;
    std::memcpy(&in32,      data,      4);
    int arg = static_cast<int>(in32);

    uint64_t __t0 = now_ns();

    int r_base = numberOfOperations(arg);

    uint64_t __t1 = now_ns();

    g_t_基准_纳秒 += (__t1 - __t0);

    g_n_基准++;
```

```
    g_sink ^= (uintptr_t)(const void*)&r_base;
    uint64_t __t2 = now_ns();
    int r_opt = numberOfOperations_opt(arg);
    uint64_t __t3 = now_ns();
    g_t_opt_ns += (__t3 - __t2);
    g_n_opt++;
    g_sink ^= (uintptr_t)(const void*)&r_opt;
    如果 (r_base != r_opt) {
        __builtin_trap();
    }

    return 0;
}

static 标准向量<uint8_t> read_file(const char* path) {
    标准输入文件流 ifs(path, 二进制模式);
    if (!ifs) {
        std::cerr << "无法打开: " << 路径 << "\n";
        exit(1);
    }
    return 标准向量<uint8_t>(
        (std::istreambuf_iterator<char>(ifs)),
        std::istreambuf_iterator<char>()
    );
}

int 主函数(int 命令行参数数量, char** 命令行参数) {
    if (命令行参数数量 < 2) {
        std::cerr << "用法: " << 命令行参数[0] << " <语料库_文件> [迭代次数]\n";
        return 1;
    }
    uint64_t 迭代次数 = (命令行参数数量 >= 3) ? strtoull(命令行参数[2], nullptr, 10) : 1000000ull;

    auto data = read_file(命令行参数[1]);
    if (data.empty()) return 0;

    // 不要测量预热时间
    uint64_t sb=g_t_baseline_ns, so=g_t_opt_ns, nb=g_n_baseline, no=g_n_opt;
    for (int i = 0; i < 1000; i++) (void)decode_输入(data.data(), data.size());
    g_t_基线_ns=sb; g_t_opt_ns=so; g_n_基线=nb; g_n_opt=no;

    for (uint64_t i = 0; i < 迭代次数; i++) {
        (void)decode_输入(data.data(), data.size());
    }

    double avg_b = g_n_baseline ? (double)g_t_baseline_ns / (double)g_n_baseline : 0.0;
    双 avg_o = g_n_opt ? (double)g_t_opt_ns / (双精度)g_n_opt : 0.0;

    std::cout << "迭代次数=" << iters << "\n";
    std::cout << "基线调用次数=" << g_n_baseline << " 平均值（纳秒/次调用）=" << avg_b << "\n";
    std::cout << "opt 调用=" << g_n_opt << " 平均值（纳秒/次调用）=" << avg_o << "\n";
    if (avg_o > 0) std::cout << "加速比=" << (avg_b / avg_o) << "x\n";
    std::cout << "（忽略）sink=" << (unsigned long long)g_sink << "\n";
    return 0;
}
```

H. 案例研究

H.1. 算法级优化

程序描述。 chocolateFeast 函数实现了巧克力购买和包装纸兑换模拟。给定初始预算 n、巧克力价格 c，以及兑换率 m （兑换一块巧克力所需的包装纸数量），该函数会计算总共可以吃掉多少块巧克力。算法分为两个阶段：(1)

初始购买用预算购买，并且 (2) 持续兑换包装纸，直到包装纸不足以兑换为止 in.

基准实现（未优化）。未优化的中间表示通过显式内存分配、重复加载和存储，以及一个逐次模拟包装纸兑换的循环来实现这一逻辑。

```
define dso_local 32 位整数 @chocolateFeast(32 位整数 noundef %0 , 32 位整数 noundef %1 , 32 位整数 noundef %2) #0 {
B0:
%3 = alloca 32 位整数, 对齐 4
%4 = alloca 32 位整数, 对齐 4
%5 = alloca 32 位整数, 对齐 4
%6 = alloca 32 位整数, 对齐 4
%7 = alloca 32 位整数, 对齐 4
存储 32 位整数 %0, 指针 %3, 对齐 4
存储 32 位整数 %1, 指针 %4, 对齐 4
存储 32 位整数 %2, 指针 %5, 对齐 4
调用 void @LLVM 生命周期开始(i64 4, 指针 %6) #2
%8 =加载 32 位整数, 指针 %3, 对齐 4
%9 =加载 32 位整数, 指针 %4, 对齐 4
%10 =有符号除法 32 位整数 %8 , %9
存储 32 位整数 %10, 指针 %6, 对齐 4
调用 void @llvm.lifetime.start.p0(i64 4, 指针 %7) #2
%11 = 加载 32 位整数, 指针 %3, 对齐 4
%12 = 加载 32 位整数, 指针 %4, 对齐 4
%13 =有符号除法 32 位整数 %11 , %12
存储 32 位整数 %13, 指针 %7, 对齐 4
分支标签 %B1
B1:
%14 = 加载 32 位整数, 指针 %5, 对齐 4
%15 = 加载 32 位整数, 指针 %7, 对齐 4
%16 =icmp sle 32 位整数 %14, %15
br i1 %16 , 标签 %B2 , 标签 %B3
B2:
%17 = 加载 32 位整数, 指针 %7, 对齐 4
%18 = 加载 32 位整数, 指针 %5, 对齐 4
%19 =sub nsw 32 位整数 %17 , %18
存储 32 位整数 %19, 指针 %7, 对齐 4
%20 = 加载 32 位整数, 指针 %6, 对齐 4
%21 =加法 nsw 32 位整数 %20 , 1
存储 32 位整数 %21, 指针 %6, 对齐 4
%22 = 加载 32 位整数, 指针 %7, 对齐 4
%23 =加法 nsw 32 位整数 %22 , 1
存储 32 位整数 %23, 指针 %7, 对齐 4
分支标签 %B1
B3:
%24 = 加载 32 位整数, 指针 %6, 对齐 4
调用 void @llvm.lifetime.end.p0(i64 4, ptr %7) #2
调用 void @llvm.lifetime.end.p0(i64 4, ptr %6) #2
返回 i32 %24
}
```

LLVM –O3 优化。 LLVM -O3 会对原始程序执行一系列标准的 IR 级优化。首先，通过 mem2reg 将栈分配标量提升为静态单赋值形式，从而消除冗余的内存操作和生命周期内建函数。循环不变计算（例如初始除法 q= n/c）会被外提至循环之外，而循环携带值则通过 PHI 节点表示。因此，LLVM -O3 生成的优化后的 IR 以更紧凑的静态单赋值形式表达包装纸兑换过程，并减少了不必要的内存流量。不过，核心计算仍然是一个循环，不断更新包装纸数量和已消耗的巧克力数量。

```
定义 dso_local 32 位整数 @chocolateFeast(32 位整数 noundef%0，32 位整数 noundef %1，32 位整数
```

```
%5 = phi函数 i32 [ %9, %B1 ], [ %3, %B
```

INTOPT. 相比之下，INTOPT 超越了结构性中间表示清理，进行算法级推理。在应用与 LLVM -O3 相同的基础优化——例如 SSA 提升、循环规范化以及不变量外提——之后，INTOPT 识别出包装纸兑换过程的一个关键数学性质。

具体而言，当 m $>$ 1且 q $>$ 0时，通过包装纸兑换获得的额外巧克力数量可以用闭式计算为：

$$\left\lfloor \frac{q-1}{m-1} \right\rfloor$$

基于这一观　　察，INTOPT 引入一条快速路径，直接将巧克力总数计算为：

$$q + \left\lfloor \frac{q-1}{m-1} \right\rfloor$$

从而在这一常见情况下完全消除该循环。重要的是，边界检查得以保留：仅当 m $>$ 1时才采用快速路径，以避免除以零，并确保与原始程序的语义等价。对于其余情况，INTOPT 回退到一个规范化循环，该循环反映原始行为，包括当 m $\leq$ 1时的不终止。

INTOPT 生成的改进策略如下：

- 将所有栈分配标量提升为静态单赋值（SSA）（mem2reg） - 移除 alloca 指令（%3、%4、%5、%6、%7）、加载/存储指令以及生命周期内建函数。 - 直接使用传入参数，并使用 PHI 节点表示循环携带值。

- 外提循环不变计算并消除冗余加载 - 仅计算一次初始商（%q = sdiv 32 位整数 %n、%c）。 - 使用 SSA 值进行循环比较和更新，而不是重新加载内存。

- 规范化循环并添加前置块保护条件
  - 构建单一循环头，并使用 PHI 节点表示运行计数（consumed 和包装纸）。 - 在前置块中检查是否可以进行交换（m <= 包装纸）。如果不能，则立即返回初始商。

- 使用 SSA 算术替代基于内存的更新，并保留标志 - 在循环体中，计算包装纸 = 包装纸 - m + 1 以及 consumed = consumed + 1，并在适当情况下使用 nsw，以匹配原始语义。

- **当 m > 1时增加快速闭式路径**
  - 当 m > 1 且初始商 q > 0时，交换次数为 floor((q - 1) / (m - 1))。无需进入循环，直接返回 q +floor((q - 1) / (m - 1))。- 保留边界检查：除非 m > 1 ，否则不要计算 (q - 1)/ (m - 1)，以避免除以零。其他情况则回退到循环，保留原有行为（包括 m <= 1时不终止）。

- 收紧函数属性
  - 优化函数仅执行纯算术，不存在内存副作用。将其标记为 nounwind、willreturn 和 memory(none)。保留原始的目标特定属性。

- 保留循环元数据
  - 将 !llvm.loop.mustprogress 附加到循环回边，以帮助进一步优化。

INTOPT 生成的优化后的 IR 如下：

```
define dso_local 32 位整数 @chocolateFeast(32 位整数 noundef %n , 32 位整数 noundef %c , 32 位整数 noundef %m) #0 {
entry:
  %q =有符号除法 32 位整数 %n, %c
  %m_gt1 = icmp sgt 32 位整数 %m, 1
  %q_gt0 = icmp sgt 32 位整数 %q, 0
  %fast = 且 i1 %m_大于1, %q_大于0
  br i1 %fast, label %fastpath, 标签 %preheader

快速路径:
  %q_减1 = 加法 nsw 32 位整数 %q, -1
  %m_减1 = 加法 nsw 32 位整数 %m , -1
  %extra = 有符号除法 32 位整数 %q_减1, %m_减1
  %total = 加法 nsw 32 位整数 %q, %extra
  返回 32 位整数 %total

前置块:
  %can_循环 = 比较有符号小于等于 32 位整数 %m, %q
  条件分支 1 位整数 %can_循环, 标签 %循环, 标签 %exit

循环:
  %包装纸 = phi i32 [ %q, %前置块 ], [ %包装纸.next, %循环 ]
  %consumed = phi函数 i32 [ %q, %preheader ], [ %consumed.next, %loop ]
  %tmp = sub nsw 32 位整数 %包装纸, %m
  %wrappers.next = 加法 nsw 32 位整数 %tmp, 1
  %consumed.next = 加法 nsw 32 位整数 %consumed , 1
  %cond = icmp sle 32 位整数 %m, %wrappers.next
  br i1 %cond, label %loop, label %exit, !llvm.loop !9

exit:
  %结果 = phi i32 [ %q, %前置块 ], [ %consumed.next, %循环 ]
  返回 32 位整数 %结果
}
```

讨论。 这个示例凸显了传统编译器优化与意图驱动优化之间的根本区别。LLVM -O3 通过在同一算法框架内重构程序来提升性能，而 INTOPT 则识别并利用算法本身更高层次的数学不变量。在明确规定的条件下，INTOPT 以闭式计算取代迭代过程，实现了从根本上属于算法层面、而非纯粹语法或结构层面的优化。

H.2. 全局协调优化

H.2.1. 反转位 _

程序描述。 此函数实现了一个依赖于数据的位反转例程。给定输入整数 n，它通过反复执行以下操作来构造输出累加器：(1) 将累加器左移一位；(2) 附加 n 当前的最低有效位；(3) 将 n 右移一位。当 n=0 时，循环终止。直观来说，该函数返回 n 的有效前缀的位反转结果：它反转 n 中截至最高有效 1 位的各位，同时忽略前导零。例如，n=6 (110b) 会得到 011b（=3）。

基准实现（未优化）。 未优化的中间表示以逐字、逐步的方式表达该算法。它为输入、累加器和临时位分配栈槽，并在循环中反复执行内存加载/存储指令。

```
define dso_local noundef 32 位整数 @_Z12reverse_bitsi(32 位整数 noundef %0) #0 {
```

```
call void @LLVM 生命周期开始(i64 4, 指针 %3) #2
分支标签 %B1
B1:
%4 =加载 32 位整数, 指针 %1,  对齐 4
%5 = icmp ne 32 位整数 %4,  0
br i1 %5 , 标签 %B2 , 标签 %B3
B2:
%6 =加载 32 位整数, 指针 %2,  对齐 4
%7 = shl 32 位整数 %6 , 1
存储 32 位整数 %7, 指针 %2,  对齐 4
%8 =加载 32 位整数, 指针 %1,  对齐 4
%9 = and 32 位整数 %8 , 1
存储 32 位整数 %9, 指针 %3,  对齐 4
%10 = 加载 32 位整数, 指针 %3,  对齐 4
%11 = 加载 32 位整数, 指针 %2,  对齐 4
%12 = 或 32 位整数 %10 , %11
存储 32 位整数 %12, 指针 %2,  对齐 4
%13 = 加载 32 位整数, 指针 %1,  对齐 4
%14 =算术右移 32 位整数 %13 , 1
存储 32 位整数 %14, 指针 %1,  对齐 4
分支标签 %B1
B3:
%15 = 加载 32 位整数, 指针 %2,  对齐 4
调用 void @llvm.lifetime.end.p0(i64 4, 指针 %3) #2
调用 void @llvm.lifetime.end.p0(i64 4, 指针 %2) #2
ret 32 位整数 %15
}
```

LLVM –O3 优化。 LLVM -O3 会执行规范化的中间表示清理和循环简化。它会将栈分配变量提升为静态单赋值形式（mem2reg），移除生命周期内建函数，并使用 PHI 节点表示循环携带状态：一个 PHI 节点用于不断变化的输入值，另一个用于累加器。

```
定义 dso_local noundefi32 @_Z12reverse_bitsi_opt(i32 noundef %0) local_unnamed_addr #0 { B0: %1 = i
```

INTOPT. INTOPT 将该循环识别为位反转操作，并通过跨越多个抽象层级的协同变换来实现：（1）识别循环的算法意图；（2）选择可映射到硬件的 LLVM 内置函数；（3）谨慎处理边界，以保留已定义行为。具体而言，INTOPT 用两个编译器内建函数替换整个循环：llvm.bitreverse.i32 用于反转 n 的全部 32 位，llvm.ctlz.i32（其中 is_zero_undef=false）用于计算前导零的数量。

INTOPT 生成的改进策略如下：

– **将循环替换为两条依赖于数据、可由后端映射为硬件指令的操作：**llvm.bitreverse.i32 用于反转全部 32 位，llvm.ctlz.i32（is_zero_undef=false）用于计算前导零的数量。最终结果为 lshr(bitreverse(n), ctlz(n))，对于非负输入，这与原循环的语义完全一致；同时通过检查 n==0来避免未定义行为。

- 保留对 n==0 的边界检查，以避免移位整个字长（这会导致未定义行为）。这样既能保证正确性，又能为所有非零输入启用无分支的快速路径。

- 避免使用 alloca 指令、生命周期内建函数以及冗余加载/存储指令。采用静态单赋值形式，并将函数标记为 readnone、nounwind 和 willreturn，以便 LLVM 在有益时执行更积极的优化和内联。

- 注意：原始 IR 对有符号整数使用算术右移（ashr），并采用 while(n != 0) 终止条件，这会导致负数输入时不终止。如果输入可能为负数，应限制调用者只能传入非负值，或将原始算法改为逻辑移位。下面的优化版本保留了零边界检查，并为所有输入提供已定义行为：当 n==0 时返回 0，当 n>0时执行快速反转。

INTOPT 生成的优化后的 IR 如下：

```
define dso_local noundefi32 @_Z12reverse_bitsi(i32 noundef %n) local_unnamed_addr #0 { entry: %iszero
```

H.2.2. Z3CONII

程序描述。 该函数根据一组守卫条件返回一个类布尔整数（0 或 1）。直观而言，它检查整数 x 是否与 y 的十进制位数相同，并满足额外的有效性约束： $x > 0$,$y \in [1, m)$，其中 m 是全局阈值，以及当 $y = 0$ 时的特殊边界情况。未优化的实现将位数计算为 $\lfloor \log_{10}(y) \rfloor + 1$，并将其与 x 比较，当且仅当二者相同时返回 1，否则返回 0。

基准实现（未优化）。 未优化的中间表示使用多个基本块和基于栈的临时变量来实现该逻辑。它通过整数到浮点数转换、对 log10 的数学库调用以及浮点数到整数的强制转换来计算 $\lfloor \log_{10}(y) \rfloor + 1$。

```
define dso_local noundefi32 @_Z3conii(i32 noundef %0, i32 noundef %1) #0 { B0: %2 =
```

```
B4:
存储 32 位整数 1，指针 %2，  对齐 4
分支 分支标签 %B10
B5:
%11 = 加载 32 位整数，指针 %4，  对齐 4
%12 =icmp sle 32 位整数 1，  %11
br i1 %12 ， label %B6 ， label %B9
B6:
%13 = 加载 32 位整数, 指针 %4，  对齐 4
%14 = 加载 32 位整数, 指针 @m, 对齐 4
%15 =icmp slt 32 位整数 %13，  %14
br i1 %15 ， 标签 %B7 ， 标签 %B9
B7:
%16 = 加载 32 位整数, 指针 %3，  对齐 4
%17 = 加载 32 位整数, 指针 %4，  对齐 4
%18 = sitofp 32 位整数 %17 to double
%19 = 调用 double @log10(double noundef %18) #2
%20 = fptosi double %19 to 32 位整数
%21 =加法 nsw 32 位整数 %20 ， 1
%22 =icmp eq 32 位整数 %16，  %21
br i1 %22 ， label %B8 ， label %B9
B8:
存储 32 位整数 1，指针 %2，  对齐 4
分支标签 %B10
B9:
存储 32 位整数 0, 指针 %2，  对齐 4
分支标签 %B10
B10:
%23 = 加载 32 位整数, 指针 %2，  对齐 4
返回 32 位整数 %23
}
```

LLVM –O3 优化。 LLVM -O3 会执行标准的中间表示级优化，包括将栈变量提升为静态单赋值形式、合并冗余基本块，以及利用 PHI 节点和 select指令简化控制流。这些变换减少了内存流量，并使控制流更加精简。然而，优化后的 IR 仍然依赖于使用 log10 进行浮点位数计算。

```
define dso_local noundefrange(i32 0,2) i32 @_Z3conii_opt(i32 noundef %0, i32 noundef %1) local_unnamed_
```

```
ret i32 %15
}
```

IntOpt。 INTOPT 通过将数据流分析与针对库函数调用的语义推理相结合，实现全局协同优化。通过追踪传递给 log10 的参数来源，INTOPT 识别出该计算针对的是整数输入 y，且其数值域受到前置控制流守卫的严格约束。这使 INTOPT 能够将浮点表达式 $\lfloor \log_{10}(y) \rfloor + 1$ 重新解释为整数位数计数操作，而非一般的库函数调用。

基于这一重新解释，INTOPT 将浮点转换和库调用替换为语义等价的整数形式。位数通过对常量 10 的幂（例如，$y \geq 10$，100，1000，.....）进行一系列比较操作来精确计算，随后累加布尔结果。

INTOPT 生成的改进策略如下：

- 将栈槽提升为静态单赋值：移除对输入参数和结果的 alloca/存储/加载。直接使用传入参数 %0 和 %1，并通过直接的静态单赋值生成结果，从基本块直接返回。- 简化入口守卫：将初始的“%0 <= 0”检查折叠为立即返回 0 的单一分支。这样可避免额外的基本块和内存流量。- 合并两步零值检查：将“%1 == 0”然后“%0 == 1”的链式结构替换为返回 zext(%0 == 1) 的单一基本块，当 %1 == 0 时。这样保留了 log10(0) 所需的边界检查，同时使分支最少。- 规范化范围检查：将“范围内 = (%1 >= 1) && (%1 < m)”计算为单一谓词，并仅基于它分支一次。这样在保留边界检查的同时减少控制流。-**强度削减并消除浮点路径：**将 floor(log10(%1)) + 1使用数学库的位数计算替换为基于与常量 10 的幂进行比较操作的精确、无分支整数十进制位数计算。这样可避免浮点转换、库调用和舍入问题，并且通常比 -O3 基于浮点的代码生成快得多、也更可预测。- 减少出口：从每个决策点直接返回（提前返回），而不是存储到临时变量再读回，或用 φ 节点构建复杂控制流。这有助于在有利时实现更简单的代码生成和尾部复制。- 收紧属性：将该函数标记为只读（它只读取 m，不写内存）。保留 nounwind/必须前进，从而允许进一步优化以及更好的代码

e

生成，且不改变语义。

INTOPT 生成的优化后的 IR 如下：

```
define dso_local noundefi32 @_Z3conii(i32 noundef %x, i32 noundef %y) #0 { entry: %cmp_
```

```
y 零 情况: %x 是 一 = icmp eq i32 %x, 1 %
```

```
范围内 —检查:
  %ge1 = icmp sge 32 位整数 %y, 1 %mval = 加载 32 位整数, 指针
```

```
ret0b:
  ret 32 位整数 0


digits_compute:
  %c1  = icmp sge 32 位整数 %y,  10
  %c2  = icmp sge 32 位整数 %y,  100
  %c3  = icmp sge 32 位整数 %y,  1000
  %c4  = icmp sge 32 位整数 %y,  10000
  %c5  = icmp sge 32 位整数 %y,  100000
  %c6  = icmp sge 32 位整数 %y,  1000000
  %c7  = icmp sge 32 位整数 %y,  10000000
  %c8  = icmp sge 32 位整数 %y,  100000000
  %c9  = icmp sge 32 位整数 %y,  1000000000

  %t1  = zext i1 %c1 转换为 32 位整数
  %t2  = zext i1 %c2 转换为 32 位整数
  %t3  = zext i1 %c3 转换为 32 位整数
  %t4  = zext i1 %c4 转换为 32 位整数
  %t5  = zext i1 %c5 转换为 32 位整数
  %t6  = 将 i1 %c6 零扩展为 32 位整数
  %t7  = 将 i1 %c7 零扩展为 32 位整数
  %t8  = 将 i1 %c8 零扩展为 32 位整数
  %t9  = 将 i1 %c9 零扩展为 32 位整数

  %s1  = 加法 nuw nsw 32 位整数 %t1 , %t2
  %s2  = 加法 nuw nsw 32 位整数 %t3 , %t4
  %s3  = 加法 nuw nsw 32 位整数 %t5 , %t6
  %s4  = 加法 nuw nsw 32 位整数 %t7 , %t8
  %s5  = 加法 nuw nsw 32 位整数 %s1 , %s2
  %s6  = 加法 nuw nsw 32 位整数 %s3 , %s4
  %s7  = 加法 nuw nsw 32 位整数 %s5 , %s6
  %s8  = 加法 nuw nsw 32 位整数 %s7 , %t9
  %digits = 加法 nuw nsw 32 位整数 %s8,  1

  %eqdigits = icmp eq 32 位整数 %x, %digits
  %retv = zext i1 %eqdigits to 32 位整数
  ret 32 位整数 %retv
}
```

讨论。 本示例凸显了 INTOPT 在优化意图引导下进行全局协同优化的优势。INTOPT 识别该计算的语义角色，通过全局守卫验证其适用性，并选择一种与目标执行模型更好对齐的实现。因此，INTOPT 执行了一项跨越多个抽象层的优化，展示了全局协同如何实现传统编译器流水线无法企及的性能提升。