LOOPRAG：利用检索增强型大型语言模型提升循环变换优化

Yijie Zhi 浙江大学
中国浙江杭州

Yayu Cao
浙江大学
中国浙江杭州

Jianhua Dai
浙江行政学院
中国浙江杭州

Xiaoyang Han
浙江大学
中国浙江杭州

Jingwen Pu
浙江大学
中国浙江杭州

Qingran Wu
浙江大学
中国浙江杭州

Sheng Cheng
北京神舟航天
软件技术有限公司
北京，北京，中国

Ming Cai
浙江大学
杭州，浙江，中国

arXiv:2512.15766v1 [cs.PL] 12 Dec 2025

摘要

循环变换是在源代码级应用的保持语义的优化技术，编译器广泛使用这类技术，以最大化向量化和并行化等优化目标。尽管已有数十年的研究，选择最优的循环变换组合仍然颇具挑战，因为这涉及依赖分析和针对优化目标的成本建模等固有复杂性。

近期研究探索了大语言模型（LLM）在代码优化方面的潜力。然而，我们的关键观察是，大语言模型往往难以有效地进行循环变换优化，常常导致错误或优化效果不佳，从而错失显著提升性能的机会。

为弥合这一差距，我们提出了 LOOPRAG，这是一种新颖的检索增强生成框架，旨在指导大语言模型对静态控制部分（SCoP）执行有效的循环优化。我们引入了一种参数驱动方法，以利用可触发各种循环变换的循环属性，并生成多样且合法的示例代码，作为示例来源。为了有效获取信息最丰富的示例，我们提出了一种基于循环特征的循环感知算法，在代码检索中兼顾相似性与多样性。为提升代码生成的正确性和效率，我们引入了一种基于反馈的迭代机制，将编译、测试和性能结果作为反馈来指导大语言模型。LOOPRAG 生成的每段优化代码都会经过变异测试、覆盖率测试和差分测试，以进行等价性检查。

我们在 PolyBench、TSVC 和 LORE 基准测试套件上评估 LOOPRAG，并将其与编译器（GCC-Graphite、Clang-Polly、Perspective 和 ICX）及代表性大语言模型（DeepSeek 和 GPT-4）进行比较。结果表明，与编译器相比，LOOPRAG 在 PolyBench、TSVC 和 LORE 上的平均加速比分别最高可达 11.20×、14.34× 和 9.29×，与基础大语言模型相比，加速比分别最高可达 11.97×、5.61× 和 11.59×。

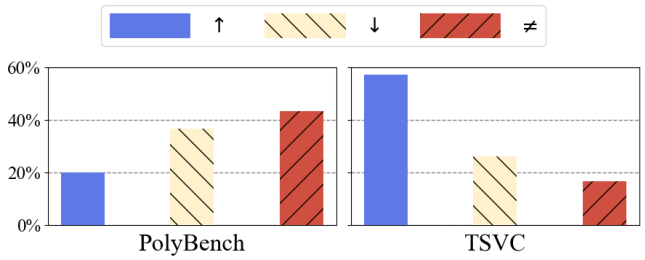

图 1. GPT-4 在 PolyBench [42] 和 TSVC[10]上生成的代码优化结果相较于 PLuTo 的百分比。所有代码均使用 GCC 编译。↑ 和 ↓ 分别表示性能优于和逊于 PLuTo。≠ 表示代码在语义上与原始代码不等价。

1 引言

循环变换是代码优化研究中的一项关键任务，它在保持语义的前提下，于源代码级调整语句的执行顺序，以提升性能。然而，如何应用最优的循环变换组合仍是一项长期存在的挑战，这需要更深入地理解代码逻辑，并具备更强的上下文感知能力 [12, 15]。

随着大语言模型（LLM）的发展，近期研究探索了其高层语义理解和上下文分析能力，并将其用于代码优化，例如自动向量化 [46] 和 OpenMP 指令预测 [44]。研究结果表明，大语言模型在代码优化任务中展现出潜力。然而，其当前能力仍不足以胜任这些任务，在正确性和性能方面仍需显著提升。

为了评估大语言模型相较于编译器的性能，我们通过提示 GPT-4 [39]并使用源到源优化编译器 PLuTo [8]对 C 程序中的静态控制部分（SCoP）执行循环变换，开展了一项实证研究（图1）。结果表明，

与 PLuTo 相比，GPT-4 经常错失实现最优循环变换的优化机会，也难以生成语义等价程序。一种可能的解释是，大语言模型缺乏预测性能影响的内在机制（例如编译器中的代价模型或启发式方法），这限制了它们应用有益循环变换的能力。

受上下文学习（ICL）的启发，[9],通过在提示中加入信息丰富的示例演示，引导大语言模型学习目标代码的潜在最优变换序列，从而增强其优化能力。例如，通过 §6.5 中的示例演示，§2.2 中的清单 1 可实现 71.60× 的加速比，相较于图 1 中 GPT-4 生成的结果。因此，我们强调，提供信息丰富的示例演示对于引导大语言模型高效执行代码优化至关重要。

因此，由优化编译器通过各种循环变换生成的大规模数据集，包含多样化的示例代码及其在静态控制部分中的优化版本，是一项重要资源，可作为示例来源。尽管现有大量循环代码数据集[13, 17] 和循环生成器 [7, 35], ，但它们在提供循环变换组合方面有所局限，导致示例缺乏多样性。基于多样化循环属性（例如循环结构和数据依赖）能够触发各种循环变换这一观察，我们旨在通过引入具有多样性的循环属性，大规模生成示例代码。然而，由于循环属性之间存在相互依赖关系，随机分配循环属性值时会产生矛盾。为解决这一问题，我们提出一种**参数驱动方法**，通过循环参数系统地利用循环属性，从而生成多样且合法的示例代码。

另一个重大挑战是如何在静态控制部分中识别最合适的示例代码。我们的目标是根据示例代码与目标代码之间的语义相似性检索示例代码。然而，现有的检索方法（例如，BM25 [43], GIST [45]和 BGE [54]）更重视文本特征，而非代码语义。循环属性能够反映与循环变换相关的语义信息，但在检索时使用全部属性可能会导致冗余。因此，我们从循环属性中提取循环特征，并提出一种**循环感知算法**，在检索时兼顾相似性与多样性。我们将检索到的示例代码及其优化版本整合为示例，以此提示大语言模型生成优化后的代码。

为提升代码生成的正确性和效率，我们引入了一种**基于反馈的迭代机制**，将编译结果、测试结果和性能排名作为信息反馈纳入提示词，以指导大语言模型。这三类反馈均在迭代生成过程中获得。然而，验证循环变换是否保留原始语义颇具挑战，因为证明语义等价性一直是一个长期存在的且不可判定的问题 [21]。现有方法 [5, 36]仍无法处理具有复杂结构的真实世界程序，也无法应对大语言模型生成过程的黑盒特性。因此，我们不依赖需要考虑数据依赖和控制依赖的理论证明，而是在利用大语言模型进行优化的同时，直接测试具体生成的输出。为了尽可能覆盖更多边界情况，我们采用了编译器测试中的技术，包括变异测试、覆盖率测试和差分测试，以进行语义等价性检查。

本文提出 LOOPRAG，这是一种面向静态控制部分中循环变换优化的新型检索增强生成（RAG）框架。LOOPRAG 旨在通过示例演示引导大语言模型学习潜在的最优变换，同时保留其固有的优化能力。我们分别将 LOOPRAG 与四种编译器（GCC-Graphite[41],Clang-Polly[27],Perspective[3]和 ICX）、两种具有代表性的大语言模型（DeepSeek[20] 和 GPT-4），以及两种基于大语言模型的模型（PCAOT[44] 和 LLM-Vectorizer [46]）进行比较，以评估 LOOPRAG。我们的方法显著提升了程序执行效率。对于编译器，在 PolyBench、TSVC 和 LORE [13],上的平均加速比分别最高可达 11.20×、14.34× 和 9.29× 。对于基础大语言模型，在相同基准测试上的平均加速比分别最高可达 11.97×、5.61× 和 11.59× 。对于基于大语言模型的模型，在 PolyBench 和 TSVC 上的平均加速比分别最高可达 8.10× 和5.97×。

2 背景与动机

2.1 循环属性与静态控制部分

在综合多种示例代码时，对循环进行建模至关重要。在建模过程中，有三类循环属性值得关注。

第一类是循环结构，包括语句数量、循环边界、循环深度（即循环层级）和循环调度。循环调度 [19] 定义语句的执行顺序，可用多种形式表示（例如 2d+1形式和调度树 [51]）。循环调度的常数维度中的值决定循环是完美循环（所有语句均位于最内层循环中），还是非完美循环（并非所有语句都位于最内层）。

第二类与数据依赖有关，考虑依赖的数量、依赖类型和依赖距离 [2, 56]。根据操作之间的关系，数据依赖有三种类型：RAW（读后写）、WAW（写后写）和 WAR（写后读）。对于迭代之间的关系，依赖距离刻画的是……之间的差异

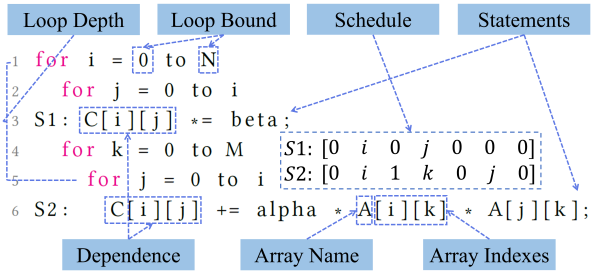

图 2. 原始代码 *syrk*。

清单 1.优化代码 *syrk*。

```
1 #pragma omp parallel 2 for t1 = 0 to N/32 3 3 for
t2 = 0 to t1 4 for t3 = 32*t1 to 32*t1+32 5 for t
4 = 32*t2 to 32*t2+32 6 S1: C[t3][t4] *= beta; 7
for k = 0 to M 8 S2: C[t3][t4] += alpha *A[t3][
k] *A[t4][k];
```

两个相关操作的迭代向量。这个差异还将依赖分为两种类型：循环无关依赖，即存在于循环同一次迭代中的依赖；以及循环携带依赖，即发生在不同迭代之间的依赖 [53]。

第三种类型关注数组访问，其中包括数组的数量、数组名称、数组大小和索引（即访问函数）。数组索引在确定数据局部性方面起着至关重要的作用 [30, 58], ，因为数组的访问顺序会直接影响内存和缓存使用的效率。

这三种循环属性可用于对静态控制部分（SCoP）中的循环进行建模 [6]。SCoP 表示程序中一个结构良好的代码段，其中所有循环边界和条件语句都由周围迭代器及全局参数的仿射函数定义。在 SCoP 中，只允许使用*for* 循环和*if* 条件语句。SCoP 的一个定义性特征是无副作用，这意味着在 SCoP 内，所有操作（包括函数调用）都仅限于不会修改 SCoP 之外任何状态的表达式。由于这种数学上的易处理性，可以使用静态代码分析器（例如 Clan [6], 、ISL [48], 和 pet [50]）以及优化编译器（例如 PLuTo[8] 和 PPCG [49]）对 SCoP 进行分析和优化。

图 2 展示了一个源自 PolyBench 中 *syrk* 的静态控制部分示例。该静态控制部分包含两条语句， $S1$ 和 $S2$，其循环深度分别为 2 和 3。迭代器 i、 j和 k的循环边界分别为 0 到 N、0 到 i和 0 到 M，由此限定了合法迭代空间。调度以二维+1 形式呈现，其中 $S1$：$[0, i, 0, j, 0, 0, 0]$ 和 $S2$：$[0, i, 1, k, 0, j, 0]$，这表明 $S1$ 和 $S2$ 共享同一个循环 i，但在循环 i中， $S1$先于 $S2$ 执行。两个数组， C和 A，在该静态控制部分中被引用。数组索引为：数组C的索引是 $[i, j]$，数组 A的索引是 $[i, k]$, $[j, k]$。操作 $*=$和 $+=$在数组 C上引发了三种依赖关系：写后写依赖、写后读依赖和读后写依赖。

这些属性是定义循环的基础信息，并且可以触发各种循环变换。

2.2 循环变换

循环变换是一系列用于性能优化（例如并行性、向量化和数据局部性）的技术（例如循环交换、循环分块、循环融合、循环分布、循环倾斜和循环移位） [24]。这些优化必须仔细分析循环属性：循环结构、数据依赖和数组访问。清单 1 展示了 GPT-4 在提供信息丰富的示例后，根据图 2 中的原始代码生成的优化代码 *syrk*。

这段优化代码采用了三种循环变换：循环分块、循环融合和循环交换，而 GPT-4 在没有示例的情况下生成的代码中并未采用这些变换。循环分块将迭代器 i 按长度为 32 的块进行划分，并将其拆分为循环 $t1$ 和$t3$中的迭代，从而使块内的数组访问更高效，提升数据局部性和缓存使用效率。迭代器 j、 $t2$ 和 $t4$ 也采用了类似的方法。

在这段优化代码中，循环交换通过交换 $S2$ 周围的迭代器 k和j，变换其调度。 $S2$ 位于循环 k 内，而非 j 内（即 $t2$ 和 $t4$）。随后，循环融合将 $S1$ 和 $S2$ 合并到同一个循环 $t4$中。这两种循环变换相结合，增强了数据的自空间和自时间局部性 [32]。此外，OpenMP [18] 指令中的 *pramga* 指示编译器为该静态控制部分实现并行性。

上述生成代码中的这些优化归功于提供给 GPT-4 的恰当示例演示；与未提供示例演示的 GPT-4 相比，加速比达到 71.60× 。

3 概述

LOOPRAG 是一个面向 C 语言的源到源循环变换优化框架。图 3 展示了 LOOPRAG 的概览，包含三个部分：**数据集合成**、**检索**和**基于反馈的迭代生成**。

首先，LOOPRAG 引入一种参数驱动方法，用于配置循环属性并合成示例代码。这些示例经过源到源优化编译器优化后，分析器会从示例代码中提取数据流信息。随后，示例代码及其对应的优化版本和数据流信息会一并存储为合成数据集（§4.1）。

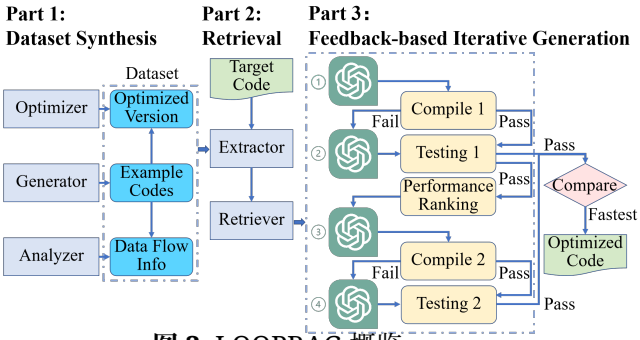

图 3. LOOPRAG 概览。

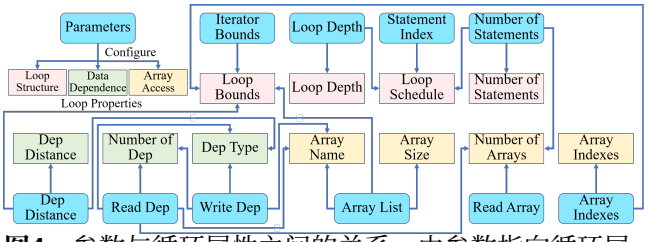

图4。参数与循环属性之间的关系。由参数指向循环属性的有向线段表示该参数可以控制此循环属性。一个循环属性可以由一个或多个参数配置。

接下来，LOOPRAG 从合成数据集中的示例代码和目标代码中提取静态控制部分，并采用一种感知循环的算法，为目标静态控制部分检索合适的示例静态控制部分。该算法利用从循环属性中提取的循环特征，识别示例与目标之间的语法和语义相似性。随后，选取排名前 N 的检索结果及其对应的静态控制部分优化版本，作为提示大语言模型的示例（§4.2）。

为提升正确性和性能，LOOPRAG 在代码生成过程中采用基于反馈的迭代机制。整个流程包含四轮迭代，并利用编译、测试和性能结果提供的反馈。在生成过程中，每个生成的静态控制部分（SCoP）都会被整合回程序中，随后通过变异测试、覆盖率测试和差分测试检查其等价性。最后，对通过测试的代码进行性能比较，并输出速度最快的代码（§4.3）。

4 方法

4.1 数据集合成

目前已有若干 C 语言循环代码数据集（例如 ANG-HABENCH [17]）和循环生成器（例如 COLA-Gen [7]）。然而，它们都无法提供具有多种循环属性的示例代码，限制了大语言模型学习循环变换的多样组合的能力。例如，COLA-Gen 通过改变循环深度和数组数量来生成代码，但仅支持包含循环携带依赖的单语句完美循环。因此，它无法应用循环融合、循环分布或循环移位，也就无法展示循环变换的复杂组合（§6.4.1）。

基于多样化循环属性能够触发各种循环变换这一观察，我们旨在通过纳入多样化循环属性，大规模生成示例代码。然而，由于这些属性之间存在相互依赖关系，随机设置每个循环属性可能会导致矛盾。例如，如果某条语句在数组 A 上存在写后写依赖，就必须对数组进行相应的写操作。但如果独立设置数组访问，就可能违反这一约束，导致对数组进行多次写操作；这也可能由其他写后读依赖或读后写依赖造成。同样，如果分别设置循环边界和数组大小，而循环上界超过数组大小，就可能导致“数组索引越界”错误。

为了解决这一问题，我们引入一种参数驱动方法，使用循环参数配置循环属性，并通过解耦、基于优先级的赋值和矛盾检查机制来消除矛盾。我们将构建十一个循环属性的任务分配给十个参数，并为合成代码创建静态控制部分。参数与循环属性之间的关系如图4所示。更多详情请参见附录A。

我们首先将相互依赖的循环属性划分为不同的组。对于每个组，我们从相关属性中识别影响因素。然后，将这些影响因素抽象为细粒度参数，以解耦它们之间的关系并约束相关属性。例如，循环边界可以由四个参数定义：**迭代器边界**、**数组列表**、**依赖距离**和**数组索引**。如果我们从**数组列表**中选择一个数组 $C[M][N]$（其中 $M < N-1$），并分别将其**数组索引** 和相关的**依赖距离** 设为 $[1,1]$ 和 $[-2,0]$，就会创建两个数组 $C[i+1][j+1]$ 和 $C[i+1-2][j+1]$。这会将循环边界约束为 $1 \le i \le M-1$ 和 $0 \le j \le N-1$。此外，如果我们为 $j < i$ 引入**迭代器边界**，循环边界将变为 $1 \le i \le M-1$ 和 $0 \le j < i$，从而确保数组访问的合法性。

在配置循环属性时，我们采用基于优先级的赋值方式来防止矛盾。例如，数组名称属性由**读依赖**、**写依赖**和**数组列表**构成。在依赖相关参数中，**写依赖**指写后写依赖，而**读依赖**指写后读依赖或读后写依赖。**读依赖**和**写依赖**决定某条语句是否包含会对数组写入产生影响的依赖，而**数组列表**则为写入提供另一种选择。我们制定了一条优先级规则：在数组定义中，依赖相关参数的优先级高于**数组列表**，以防止产生矛盾。

然而，循环参数之间仍可能存在矛盾（例如，两个依赖相关参数之间）。为解决这一问题，我们提出了一种矛盾检查机制。例如，由**写依赖**控制的写后写依赖中的目标数组信息，源自源数组；而由**读依赖**控制的写后读依赖或读后写依赖中的数组读取信息，则源自写入。因此，不允许创建循环依赖 [2]，因为循环追溯依赖来源无法获得数组的有效信息。检测到任何循环依赖后，我们会舍弃与调度中较晚出现的语句相关联的依赖，以确保合法性。

独立的循环属性可直接通过参数配置，且不会产生矛盾。所有参数的值都在指定范围内随机生成，以便触发各种循环变换。

最后，我们将循环属性转换为代码的静态控制部分（SCoP），并将其与周围结构集成，形成完整的程序，从而完成综合（附录 B）。综合示例代码后，我们使用优化编译器和分析器获取优化版本，并提取每个示例的数据流信息。示例代码、优化版本和数据流信息均存储为数据集，以供检索。

4.2 检索

在检索部分，我们提取数据集中的目标代码和示例代码，将其转换为静态控制部分（SCoP），并评估相似度。现有检索方法主要关注文本特征，往往忽略循环属性。然而，SCoP 中的任何细微差异，例如交换数组访问中的索引，都可能导致语义截然不同，并显著影响循环变换的应用。

为了解决这个问题，我们根据循环属性评估 SCoP 之间的相似度。然而，并非所有循环属性都有助于计算相似度。例如，在 SCoP 中重命名数组名称的所有出现之处，不会影响循环变换的应用。此外，某些循环属性可以从其他属性推导出来。例如，当同一数组的相同索引在不同迭代中被访问时，就会产生依赖，而这可以通过循环调度和数组索引来表示。

另一方面，先前研究的观察结果 [52,57] 表明，示例演示中的多样性可能会减轻误导性指导对大语言模型的影响。因此，为了兼顾相似性与多样性，我们提出了一种循环感知算法，根据从目标和示例 SCoP 的循环属性中提取的循环特征计算奖励得分和惩罚得分。循环特征包括循环调度和数组索引，它们与循环变换密切相关。例如，当数组索引与循环嵌套中迭代器的顺序不一致时，可将其作为循环交换的指标，并应获得奖励得分与循环嵌套中的迭代器顺序有关，并值得获得奖励得分。

当示例和目标 SCoP 的语句数量不同时，计算语句不匹配惩罚分**S**数（S_M），如下所示：

$$S_M = |NS^T - NS^E| \times \sum_{0 \le i \le N_F} WP_i \tag{1}$$

其中，NS^T 和 NS^E 分别表示目标和示例 SCoP 中的语句数量**N**，N_F 表示特征**N**的类型数量，WP_i 表示第 i 个特征的惩罚**W**权重。

特征**S**分数（S_F）包含语句匹配时的奖励和惩罚。奖励**R**分数（R）旨在捕捉目标与示例 SCoP 之间的特征交集，如下所示：

$$R_{i,j} = Count(F^T_{i,j} \cap F^E_{i,j}) \times WR_j \tag{2}$$

其中$R_{i,\ j}$是第i条语句中第 j个特征的**R**eward分数，$F^T_{i,\ j}$和$F^E_{i,\ j}$分别表示目标和示例静态控制部分中第 i条语句的第 j个**F**eature的信息，而 WR_j表示第 j个特征的**R**eward权重**W**。

P惩罚分数（P）旨在解决示例代码静态控制部分中的冗余问题，避免检索采用不恰当循环变换的示例静态控制部分。分数的计算方式如下：

$$P_{i,j} = (Count(F^T_{i,j} \cap F^E_{i,j}) - NF^E_{i,j}) \times WP_j \tag{3}$$

其中$P_{i,\ j}$是第i条语句中第 j个特征的**P**惩罚分数，$NF^E_{i,\ j}$表示示例静态控制部分中第 i条语句的第 j个**F**eature数量。$Count(F^T \cap F^E) - NF^E$用于计算示例静态控制部分相较于目标静态控制部分的未匹配特征数量。这意味着，当示例静态控制部分包含的特征多于目标静态控制部分时，应施加惩罚，因为这可能导致出现不需要的循环变换示例演示。相反，尽管相反的情况（示例静态控制部分中的特征较少）会导致示例演示的多样性降低，但其危害小于不恰当的示例演示。因此，我们只在示例静态控制部分包含更多特征时施加惩罚。

分数 R 和 P 按如下方式计算并归入**分**数**特**征 S_F：

$$S_F = \sum_{0 \le i \le min(NS^T, NS^E), 0 \le j \le N_F} \frac{R_{i,j} - P_{i,j}}{NF^T_{i,j}} \tag{4}$$

其中，$NF^T_{i,\ j}$表示目标静态控制部分中第i条语句的第 j个**特**征的**数**量。S_F除以 NF^T进行归一化，以使用相对数量的比例而非绝对数量来计算循环特征相似度，并考察示例与目标之间的相似度。

计算循环属性的分数后，总体循环感知分数（LAScore）定义如下：

$$LAScore = S_B + S_W = S_B + \frac{S_F - S_M}{NS^T} \quad (5)$$

其中，S_B 表示文本相似度的基础分数（例如 BM25 [43]），用于确保语法鲁棒性；S_W 表示高层语义特征的加权分数。S_M 和 S_F 通过 NS^T 进行归一化，以消除语句数量的影响。

与传统文本检索相比，稀疏检索算法应用于代码检索时表现出很强的鲁棒性，因此在领域适配方面尤其具有优势 [47] ，并能提高检索效率。我们选择 BM25 作为代表算法，计算 LAScore 的基础相似度分数。计算出 LAScore 后，我们选取得分最高的前 N 个示例 SCoP。随后，将检索到的示例 SCoP及其优化版本组合起来，用于提示大语言模型。

4.3 基于反馈的迭代生成

大语言模型首次生成时并不总能得到最佳结果，往往需要反馈机制从错误中学习并逐步改进 [37]。因此，在代码生成部分，LOOPRAG 旨在提供有用的反馈，迭代引导大语言模型生成正确且高效的 SCoP。

沿用先前工作 [1, 23], ，我们根据正确性和性能将代码生成中的常见问题分为五类：编译错误（CE）、错误答案（IA）、运行时错误（RE）、执行超时（ET）和低效代码（IC）。

我们提出三种反馈来指导大语言模型：编译结果、测试结果和性能排名。编译结果帮助大语言模型修复语法错误（CE）。对于执行错误（即 IA、RE 和 ET），我们提供测试结果，以便大语言模型改进代码。对于低效代码（IC），受排序学习 [34],的启发，LOOPRAG 根据性能为所有通过测试的代码排序，为大语言模型生成更高效的代码提供提示。

这些反馈机制被整合到基于反馈的迭代生成流程中，该流程由四个步骤组成：

步骤 1：构建示例演示，其中包含为目标 SCoP 检索到的示例 SCoP 及其对应的优化版本。使用带有示例演示的提示词生成优化后的 SCoP，将其整合到程序中，并编译生成的代码；

步骤 2：使用编译结果重新生成在步骤 1 中未能通过测试的代码。对重新生成的代码以及在步骤 1 中通过测试的代码进行正确性测试。执行并评估通过测试的代码，并按性能对其排序；

步骤 3：结合测试结果和性能排名，引导大语言模型从执行错误和有效优化中学习，并启动下一轮代码生成；

步骤 4：重复编译、生成和测试流程。比较性能结果，并选择其中最优的结果作为生成流程的最终输出。

验证循环变换是否保留了原始功能颇具挑战，因为证明两个程序语义等价的问题由来已久且不可判定 [21]。已有多种方法被提出，例如代码不变量 [40],形式化验证 [36] 和检查点比较 [5]。然而，这些方法难以扩展到真实世界的程序 [29]（例如，涉及二维数组的函数[14]）。在实践中，我们采用编译器测试中的方法进行等价性检查 [11], ，并将测试结果与性能排名相结合，以指导大语言模型。

具体而言，我们采用大语言模型辅助的变异方法来生成并扩充测试输入。首先，利用 GPT-4 学习基准程序的输入结构（例如数据类型和数组维度）及其功能，然后生成一组初始化函数作为种子输入 [33]。利用这些种子输入，LOOPRAG 通过基于值、基于运算符和基于语句的变异，高效地创建多样化的测试输入。

覆盖率引导测试旨在实现较高的分支覆盖率。当分支覆盖率达到 100% 或趋于饱和时，LOOPRAG 会认为程序已得到充分测试。通过采用覆盖率引导测试，测试用例的数量显著减少：在基准测试套件中，每个程序的测试用例平均数从 500+ 降至仅 25 个。

我们引入差分测试方法作为预言机，用于交叉检查真实代码与大语言模型生成代码的输出。我们使用两种测试方法：校验和测试和逐元素测试。前者作为快速筛选手段，用于匹配两段代码的校验和结果；后者则通过逐元素比较来检查等价性。

5 实现

如图 5 所示，LOOPRAG 在代码生成器中采用参数驱动的方法，随机设置参数、配置循环属性并合成示例代码，同时确保其合法性。我们使用 Clan 0.8.0 和 CAnDL 0.6.2 作为分析器。Clan 和 CAnDL 均采用默认选项。我们选择 PLuTo 作为示例代码的优化编译器，因为它优化效果可靠且易于操作。我们修改了 PLuTo 0.11.4，以满足提取循环属性信息的需求，并

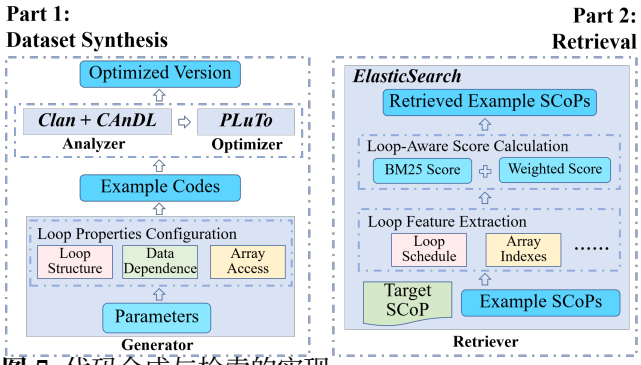

图 5. 代码合成与检索的实现。

添加了用于指定全局参数的函数，从而减少优化版本中冗余的条件分支。LOOPRAG 使用的 PLuTo 选项为 -q -custom-context -tile -parallel-nocloogbacktrack。Clan 和 CAnDL 用于提取数据流信息，PLuTo 用于生成示例代码的优化版本。代码中的静态控制部分由指令 # *pragma scop* 和 #*pragma endscop*包围，用于标示 LOOPRAG 和 PLuTo 将检测并优化的区域。

在我们的检索器中，我们将 Elasticsearch [26] 7.13.2 部署为数据存储与检索平台。LOOPRAG 提出了一种基于 BM25 算法的循环感知检索算法，重点提取目标与示例 SCoP 中循环属性的循环特征（附录 D）。我们将检索到的示例 SCoP 数量 N 设为 10，并从中随机选择 3 个及其优化版本，作为示例演示来提示大语言模型。我们提供了具体的提示规则，引导大语言模型从反馈中学习并迭代生成 SCoP（附录 E）。随后，将 SCoP 补全以形成代码。生成代码的数量 K 设为 7。

对于三种类型的反馈，我们从编译器（例如 GCC）获取编译结果。测试结果综合了种子输入变异、分支覆盖率和差分测试，估算 SCoP 末尾的输出数组，并测量整个 SCoP 的执行时间。种子输入由 GPT-4 生成，我们使用 gcov 工具测试分支覆盖率。考虑到精度，我们使用 OpenMP 中的 *omp_get_wtime*，而不是 *rtclock* 或 *rdtsc* 指令。

6 评估

我们设计实验旨在回答以下研究问题：

1. LOOPRAG 在提升循环优化方面的效果如何？ 2. LOOPRAG 能否超越其优化演示源 PLuTo？ 3. 不同模块对 LOOPRAG 有何贡献？
4. LOOPRAG 如何帮助提高循环优化的质量？

6.1 实验设置

硬件和软件规格。 我们的工作聚焦于CPU目标。所有实验均在一台Linux服务器上进行，该服务器运行64位Ubuntu 20.04.4，配备两颗24核AMD EPYC 7352处理器（@ 2.3GHz，共96个线程）、八张NVIDIA RTX 4090显卡，以及256GB内存，DDR4 @ 3.2GHz。我们采用参数驱动方法合成了135,364个示例代码 [1]。对于基础大语言模型，我们使用DeepSeek-V3（deepseek-v3-0324）和GPT-4（gpt-4o-2024-08-06），并将大语言模型的温度设置为0，以确保代码生成过程中的确定性。我们使用GCC 15.1.0作为基础编译器，并配置选项 -O3、-fopenmp。

基准测试。 我们在三个基准测试套件上评估 LOOPRAG：PolyBench [42], TSVC [10] 和LORE [13]。PolyBench包含从实际应用中提取的30个数值计算，广泛用于评估编译器中的循环优化技术。TSVC由149个 *for* 循环组成，用于评估与向量化和并行性相关的编译器优化。LORE包含从基准测试套件、库和实际应用中提取的大量*for* 循环嵌套，用于评估编译器。

我们从这些测试套件中提取符合静态控制部分要求的基准测试。我们手动为 TSVC 和 LORE 添加指令，使分析器（例如 Clan）能够识别每个基准测试中的静态控制部分。PolyBench 中已包含这些指令。PolyBench 总共包含 30 个基准测试，而 TSVC 和 LORE 分别包含 84 个和 49 个基准测试。对于参数值，PolyBench 使用选项 EXTRALARGE_DATASET，TSVC 使用默认设置，而 LORE 的参数值则由我们手动设置。我们为 TSVC 中的 *attribute* 修饰的 *dummy* 函数添加 ___*((pure))* 限定符，以确保 Clang-Polly 能够正确检测静态控制部分；而 GCC-Graphite 由于 *dummy* 函数的不透明性，无法识别 TSVC 中的静态控制部分。有关我们评估中静态控制部分处理策略的详情，请参阅附录 C

我们将合成代码与基准测试套件中的代码进行比较，确认不存在相同代码。测试期间在程序末尾验证的输出数组已在 PolyBench 和 TSVC 中预先定义。对于 LORE，我们手动识别一些与功能相关的数组（例如，静态控制部分中唯一发生数组写入的数组）。

基线。 我们将 LOOPRAG 与四类基线进行比较：四款领先的编译器、两个具有代表性的大语言模型、两个专为代码优化设计的基于大语言模型的模型，以及 PLuTo（LOOPRAG 的学习目标）。

[1] https://anonymous.4open.science/r/Dataset-for-LOOPRAG-48C5

对于基线编译器，我们使用搭配 Clang 20.1.4 的 Polly [27]、搭配 GCC 15.1.0 的 Graphite [41]、ICX 2024.2.0，以及搭配 Clang 9.0.2 的 Perspective [2] [3]。Polly 和 Graphite 将多面体模型集成到生产级编译器中，以支持循环并行化；而 Perspective 则代表了最先进（SOTA）的系统，它采用推测技术实现自动循环并行化。编译选项分别设为：Clang-Polly 使用 -O3 -fopenmp -mllvm - polly -mllvm -polly-parallel -mllvm - polly-tiling，GCC-Graphite 使用 -O3 -fopenmp -floop-nest-optimize -floop-parallelize-all，ICX 使用 -O3 -qopenmp -xHost。*TRANSFORM_TIMEOUT*和*PROFILE_TIMEOUT*在 Perspective 中均设为 600 秒。PLuTo 0.11.4 使用选项 -tile -parallel -nocloogbacktrack。

对于大语言模型，我们将 LOOPRAG 与 DeepSeek-V3（deepseek-v3-0324）和 GPT-4（gpt-4o-2024-08-06）进行比较。这两个大语言模型既用作 LOOPRAG 的基础大语言模型，也作为对比基线。它们分别代表代码专用语言模型和通用语言模型。我们通过官方 API 使用指令提示来访问这两个模型。

对于基于大语言模型的模型，我们选择 PCAOT [44] 和 LLM-Vectorizer [46]，因为它们代表了代码优化领域的 SOTA 方法。PCAOT 使用指令提示和思维链（COT）策略，提示大语言模型生成用于并行计算的优化代码。LLM-Vectorizer 探索大语言模型和基于人工智能的智能体执行高效循环向量化优化的能力。由于这两种方法均未发布其软件，我们直接报告各自论文中的结果。

评估指标。我们使用三项评估指标评估生成的代码，包括 pass@k [31], 性能加速比和更快代码的比例。pass@k 指标衡量前 K 个生成代码中至少有一个通过一组正确性测试的比例。

加速比指标通过比较优化代码中静态控制部分（SCoP）的执行时间与原始代码的执行时间，计算性能提升的算术平均值（首次运行后测量五次）。默认情况下，所有代码均使用 GCC 在空闲测试环境中编译。

更快代码的比例反映了加速比更高的代码所占的比例，从而避免单一平均加速比指标带来的不稳定性。

我们将代码中静态控制部分（SCoP）的性能限制设为 120 秒，作为 LOOPRAG 能否成功应用高效优化的判定阈值。对于基线，此阈值扩大至 600 秒。失败代码的加速比设为 0。计算平均加速比指标时，会排除离群值（例如，加速比大于 600× 的情况），以降低标准差误差。

[2]https://github.com/普林斯顿大学/cpf

6.2 LOOPRAG 在增强循环优化方面的效果如何?

表 1. LOOPRAG 相较于基线编译器所达到的 pass@k 和加速比。*LD-GCC* 和*LG-GCC* 分别表示将 GCC 配置为基础编译器、DeepSeek 和 GPT-4 配置为基础大语言模型的 LOOPRAG。*LD-Clang* 和*LG-Clang* 分别表示将 Clang 配置为基础编译器、DeepSeek 和 GPT-4 配置为基础大语言模型的 LOOPRAG。*LD-ICX* 和*LG-ICX* 分别表示将 ICX 配置为基础编译器、DeepSeek 和 GPT-4 配置为基础大语言模型的 LOOPRAG。

编译器	PolyBench		TSVC		LORE	
	Pass@k	加速比	Pass@k	加速比	Pass@k	加速比
LD-GCC	70.00	**23.97**	**94.05**	**32.66**	86.71	**20.44**
LG-GCC	70.00	14.58	84.52	31.35	87.76	14.86
Graphite	**96.67**	0.99	-	-	**100**	1.03
LD-Clang	66.67	8.62	94.05	15.82	85.71	**6.49**
LG-Clang	70.00	14.06	84.52	24.00	87.76	5.01
Polly	**93.33**	**14.62**	**100.00**	**27.54**	**100.00**	1.05
Perspective	26.67	1.79	-	-	61.22	3.07
LD-ICX	**66.67**	9.63	**94.05**	12.30	85.71	**5.63**
LG-ICX	**66.67**	**12.38**	83.33	**23.28**	**87.76**	4.90

6.2.1 与编译器的性能比较。为了评估 LOOPRAG 提升代码效率的效果，我们首先将 LOOPRAG 与四种编译器进行比较，包括 Clang-Polly、GCC-Graphite、ICX 和 Perspective。加速比通过比较各编译器下优化代码与原始代码的性能计算得出。

由于每个 TSVC 内核都包含一个虚设 函数调用，除非明确保证该函数无副作用，否则 Graphite 中的 SCoP检测会失败。因此，为确保公平性，我们将 Graphite 排除在 TSVC 对比之外。此外，Perspective 无法处理迭代规模较大的 TSVC 代码，因为最外层循环需要迭代 100,000 次，导致性能分析过程超时。因此，我们也不纳入 Perspective 的 TSVC 结果。

如表 1 所示，在各基准测试套件中，LOOPRAG 相较于 Graphite 和 ICX 均取得了显著的性能提升。具体而言，与 Graphite 相比，LOOPRAG 在 PolyBench 和 LORE 上的平均加速比分别为 19.47× （即 (23.97/0.99 +14.58/0.99)/2）和 17.14×。与 ICX 相比，LOOPRAG 在 PolyBench、TSVC 和 LORE 上的平均加速比分别为 18.18×、27.97× 和 12.67× 。

对于 Polly，LOOPRAG 在不同基准测试套件上的表现各异，在 PolyBench、TSVC 和 LORE 上的平均加速比分别为 0.78×、0.72× 和 5.48× 。具体而言，我们发现*LG-Clang* 优于*LD-Clang*，并且在 PolyBench 和 TSVC 上的性能与 Polly 相当。这表明*LG-Clang*使用 GPT-4 作为基础大语言模型，在与基础编译器 Clang 协同进行代码优化时，可能更为有效。

与 Perspective 相比，LOOPRAG 在 pass@k 和加速比两方面均优于它。在 PolyBench 和 LORE 上，平均加速比（均基于 Clang-9）分别提升了 6.34× 和 1.87×。检查 Perspective 的失败案例后，我们发现它依赖精细且耗时的性能分析和优化流程。例如，TSVC 代码会导致大量超时，而 PolyBench 和 LORE 中复杂度各异的代码则容易使分析过程失败。

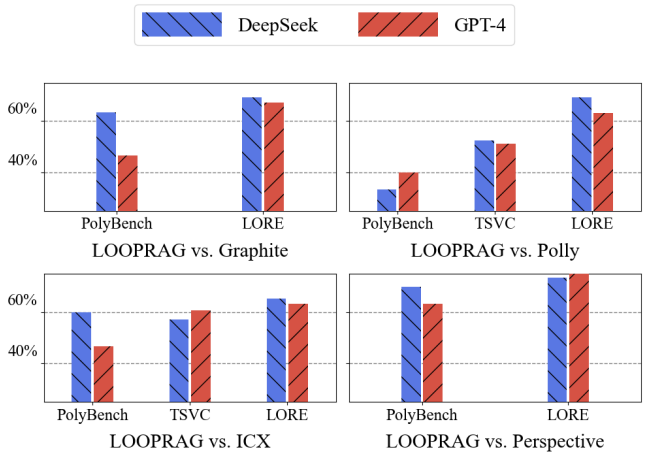

图6。与四种编译器相比，LOOPRAG 生成更快代码的百分比。

图6展示了与四种编译器相比，LOOPRAG 生成更快代码的百分比。我们发现，在所有可用的基准测试套件中，LOOPRAG 生成的代码比 Graphite、ICX 和 Perspective 快的比例均超过 40%，在 LORE 上甚至达到 80%。对于 Polly，LOOPRAG 在 TSVC 和 LORE 上生成的代码有超过 50% 更快，但在 PolyBench 上这一比例要低得多，这体现了 Polly 在处理复杂代码方面的能力。

此外，LOOPRAG 在 TSVC 和 LORE 上生成的更快代码多于在 PolyBench 上的情况。一个可能的原因是，与 TSVC 和 LORE 中的代码相比，PolyBench 中的代码具有更复杂的循环属性。因此，LOOPRAG 更难在 PolyBench 上执行有效的循环优化。

总体而言，与多面体编译器（Polly 和 Graphite）、通用编译器 ICX 以及基于推测的编译器 Perspective 相比，LOOPRAG 在所有基准测试套件中均展现出更优的性能和适应性。

6.2.2 与基于大语言模型的方法比较。对于基于大语言模型的方法，我们分别使用 LOOPRAG 和各个基于大语言模型的基线优化原始代码，然后比较编译和测试后的结果。

如表2所示，与两种具有代表性的基础大语言模型（即 DeepSeek 和 GPT-4）相比，LOOPRAG 在实现显著更高加速比的同时，展现出较高的 pass@k 性能。具体而言，LOOPRAG 在 PolyBench、TSVC 和 LORE 上的平均加速比分别提升了 11.97×（即 (23.97/1.61+14.58/1.61)/2）、5.61× 和 11.59×。这些结果凸显了 LOOPRAG 在执行有效循环优化方面的优势。

表2。LOOPRAG 相较于基于大语言模型的方法所取得的 pass@k 和加速比。对于 LOOPRAG、基础大语言模型和 LLM-Vectorizer，k 的值设为 7，而 PCAOT 使用 12。

方法	LLMs	PolyBench		TSVC		LORE	
		Pass@k	加速比	Pass@k	加速比	Pass@k	加速比
LOOPRAG	DeepSeek	**70.00**	**23.97**	94.05	**32.66**	86.71	**20.44**
	GPT-4	**70.00**	14.58	84.52	31.35	87.76	14.86
基础大语言模型	DeepSeek	66.67	1.61	**97.62**	6.75	**91.84**	1.72
	GPT-4	**70.00**	1.61	92.86	4.91	85.71	1.60
PCAOT	GPT-4	65.35	1.80	-	-	-	-
	CLLama-70B	63.35	2.26	-	-	-	-
LLM-Vectorizer	GPT-4	-	-	68.00	5.25	-	-

比较 DeepSeek 和 GPT-4 上的结果后，我们观察到，基于 DeepSeek（DeepSeek-v3-0324）的 LOOPRAG 比 GPT-4（gpt-4o-2024-08-06）实现了更高的加速比，这主要归因于底层大语言模型发布时间的差异。具体而言，DeepSeek-v2.5 在 PolyBench 上实现的加速比低于 GPT-4。这表明，LOOPRAG 能够帮助大语言模型加深对复杂代码逻辑和上下文的理解，并且在使用更新的基础大语言模型时，仍能有效地对静态控制部分进行代码优化。

对于最先进的大语言模型，我们将 LOOPRAG 与 PCAOT 和 LLM-Vectorizer 进行比较。从表2中可以得出两个关键观察结果。首先，与 PCAOT 相比，LOOPRAG 在 PolyBench 上实现了相当的 pass@k性能，同时平均加速比提升了 8.10× 。其次，与 LLM-Vectorizer 相比，LOOPRAG 的 pass@k性能约高出 16.52%，在 TSVC 上的平均加速比提升了 5.97× 。这些结果凸显了 LOOPRAG 在效率和正确性方面的有效性。

在图7中，尽管 LOOPRAG 为 LORE 生成的优化后的代码（约 60%）多于为 PolyBench 生成的代码（约 50%），但二者的性能提升非常接近，分别为 11.97× 和 11.59×。这一观察结果表明，与 LORE 相比，PolyBench 中的代码为 LOOPRAG 提供了更多进行高效优化的机会。更多详情请参见附录F。

6.3 LOOPRAG 能否超越其优化示例演示来源 PLuTo?

LOOPRAG 从优化编译器 PLuTo 中学习应用恰当的循环变换组合的策略。这引出了一个问题：**LOOPRAG 能否超越其学习目标?**

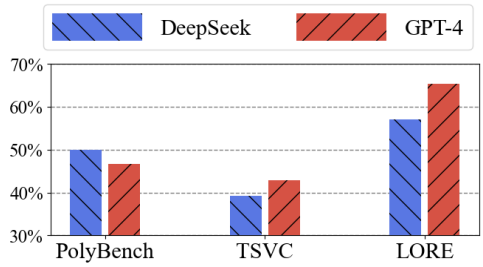

图7. LOOPRAG生成的更快的代码相比基础大语言模型的比例。

表3. LOOPRAG与PLuTo相比取得的pass@k和加速比。

方法	LLMs	PolyBench		TSVC		LORE	
		Pass@k	加速比	Pass@k	加速比	Pass@k	加速比
LOOPRAG	DeepSeek	70.00	23.97	**94.05**	**32.66**	85.71	**20.44**
	GPT-4	70.00	14.58	84.52	31.35	87.76	14.86
PLuTo		**83.33**	**43.29**	**94.05**	5.88	**95.92**	4.03

表 3 对比了 LOOPRAG 和 PLuTo 生成的优化代码，两者的唯一区别在于所使用的优化器。我们的 600 秒执行超时机制导致一些经 PLuTo 优化的代码执行失败，尤其是处理大型数组时。LOOPRAG 在 TSVC 和 LORE 上分别实现了平均加速比提升**5。44× 和 4。38× 然而，在 PolyBench 上，LOOPRAG 的加速比降低了 0。45×**，这与 PCAOT [44] 的研究结果一致：PLuTo 在多面体模型 [22] 方面的专长使其在 PolyBench 上的效率显著更高。相比之下，在 TSVC 和 LORE 上，PLuTo 可能会应用无益的变换（例如，在迭代规模较小时进行循环分块），并因过度分布和剥离而产生额外开销，导致加速比降低。

一般而言，尽管 LOOPRAG 主要从其示例演示中由 PLuTo 生成的优化版本学习并运用循环变换，但它并未限制大语言模型自身的优化能力。因此，LOOPRAG 可以超越 PLuTo。例如，LOOPRAG 可以通过考虑向量化和数据局部性等额外优化目标，生成更有效的循环变换组合；还可以采用标量重命名和归约等辅助技术 [10], ，而不是仅依赖循环变换。

图 8 展示了 LOOPRAG 相较于 PLuTo 生成更快的代码的比例，进一步支持了上述关于 PLuTo 的结论。尽管 PLuTo 在 PolyBench 中超过 60% 的代码上表现优于 LOOPRAG，但 LOOPRAG 在 TSVC 和 LORE 中实现的优化代码数量约多 60%。这些结果表明，LOOPRAG 可作为实用的代码优化工具，并在开发多样化的优化技术方面具有更大潜力。

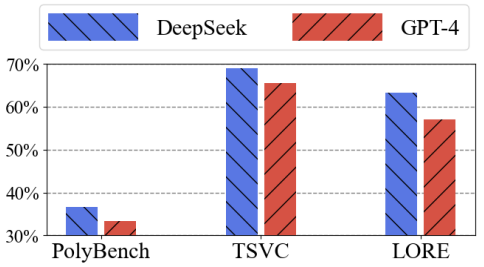

图 8。 LOOPRAG 生成的更快代码占比（与 PLuTo 相比）。

6.4 LOOPRAG 各模块分别有哪些贡献?

为探究 LOOPRAG 中不同组件各自的贡献，我们进行了三项实验，以评估这些方法的影响。

6.4.1 示例代码综合。 为了评估参数驱动方法对数据集合成的影响，我们将 COLA-Gen [7] 作为基线，在默认设置下（即循环深度设为 2，数组读取次数设为 1），使用 LOOPRAG 合成相同数量的示例代码（135,364）。我们开展了三项分析研究，分别探究示例代码中循环属性的分布、优化版本中触发的循环变换类型，以及优化代码的性能。

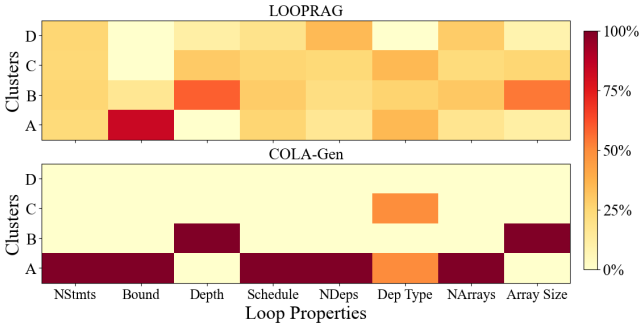

图 9. LOOPRAG 和 COLA-Gen 合成的示例代码中，静态控制部分内循环属性的分布比例。**NStmts**、**Bound**、**Depth**、**Schedule**、**NDeps**、**DepType**、**NArrays** 和 **ArraySize** 分别表示语句数量、循环边界、循环深度、循环调度、依赖数量、依赖类型、数组数量和数组大小等循环属性。

图 9 展示了示例代码中八种循环属性的分布。每种循环属性根据其取值划分为四个簇。例如，对于 **NDeps**，包含 0、1 或 2 个依赖的 SCoP 属于簇 **A**，而包含 3–5、6–10 个以及至少 11 个依赖的 SCoP 分别属于簇 **B**、**C** 和 **D**。我们观察到，在 COLA-Gen 合成的示例代码中，所有循环属性都高度集中于一个或两个簇中，

而在 LOOPRAG 合成的示例代码中，这些属性的分布则更加均匀。这表明，与 COLA-Gen 相比，LOOPRAG 通过参数驱动方法能够合成更多样化的示例代码。

我们使用优化编译器 PLuTo 生成优化版本，并分析这些版本中触发的循环变换。表 4 展示了循环变换类型上的差异。结果表明，LOOPRAG 中由参数驱动的方法合成了更多样化的示例代码，因此优化版本中的循环变换也更加多样。

表4. LOOPRAG 和 COLA-Gen 合成的示例代码在优化版本中触发的循环变换类型差异。

方法	循环变换					
	分块	交换	倾斜	融合	分布	移动
LOOPRAG	✓	✓	✓	✓	✓	✓
COLA-Gen	✓	✓	✓	×	×	×

表5. LOOPRAG 相较于 COLA-Gen 所取得的 pass@k 和加速比。

方法	LLMs	PolyBench		TSVC		LORE	
		pass@k	加速比	pass@k	加速比	pass@k	加速比
LOOPRAG	DeepSeek	70.00	**23.97**	**94.05**	32.66	**85.71**	**20.44**
	GPT-4	70.00	14.58	84.52	31.35	87.76	14.86
COLA-Gen	DeepSeek	**76.67**	9.82	**94.05**	**35.08**	**89.80**	17.85
	GPT-4	60.00	2.82	82.14	12.94	83.67	11.44

表5比较了使用 LOOPRAG 和 COLA-Gen 综合的示例代码生成的优化代码的性能。总体而言，LOOPRAG 在 pass@k 和加速比方面均优于 COLA-Gen。PolyBench、TSVC 和 LORE 的平均加速比分别提升了 3.81×、1.68× 和 1.22×。图10中的结果也凸显了参数驱动方法在数据集合成中的作用，这为高效执行代码优化奠定了基础。

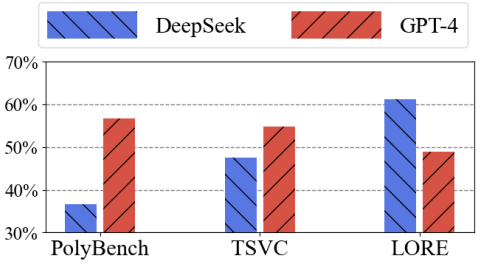

图10. LOOPRAG生成的更快的代码相较于COLA-Gen的百分比。

表 6.LOOPRAG使用循环感知算法相较于BM25和加权分数所取得的Pass@k和加速比。“加权分数”指仅使用加权分数的LOOPRAG（即LAScorew/oBM25）。

方法	LLMs	PolyBench		TSVC		LORE	
		Pass@k	加速比	Pass@k	加速比	Pass@k	加速比
循环感知	DeepSeek	70.00	**23.97**	**94.05**	32.66	85.71	20.44
	GPT-4	70.00	14.58	84.52	31.35	87.76	14.86
BM25	DeepSeek	**73.33**	16.50	88.10	28.67	89.8	11.14
	GPT-4	63.33	9.35	80.95	**42.67**	87.76	9.65
加权分数	DeepSeek	70.0	18.75	**94.05**	19.61	**91.84**	**21.79**
	GPT-4	60.00	13.66	85.71	26.54	77.55	14.88

6.4.2 检索算法。为评估循环感知算法在检索中的有效性，我们将其与 BM25 算法以及不使用 BM25 分数的循环感知算法（加权分数）进行比较，以考察我们的方法在提升代码优化能力方面的效果。

如表 6 所示，三种方法展现出相近的 pass@k。然而，循环感知算法在不同方面表现出更高的加速比，这与其设计方式相对应。例如，在 PolyBench 和 LORE 上，循环感知算法相较于 BM25 的平均加速比提升分别为 1.51× 和 1.69×。与加权分数方法相比，循环感知算法在 PolyBench 和 TSVC 上表现更高，在 PolyBench 和 TSVC 上分别实现了 1.17× 和 1.42× 的平均加速比提升。这些结果表明，通过利用循环特征，循环感知算法能够在相似性和多样性之间取得平衡，尽管其在 TSVC 上的加速比低于 BM25，在 LORE 上的加速比低于加权分数方法。

图 11 展示了 LOOPRAG 使用循环感知算法生成的更快代码占比，并将其与 BM25 和加权分数进行比较。平均而言，与 BM25 和加权分数相比，循环感知算法生成的加速比更高的优化代码约多 50%。这些结果凸显了该算法在提升代码优化性能方面的显著作用。与只关注单一方面的单项分数相比，它能提供信息更丰富的示例。因此，我们采用 LAScore，因为它性能均衡，且具备处理复杂场景的强大能力，能有效引导大语言模型生成效率更高的代码。

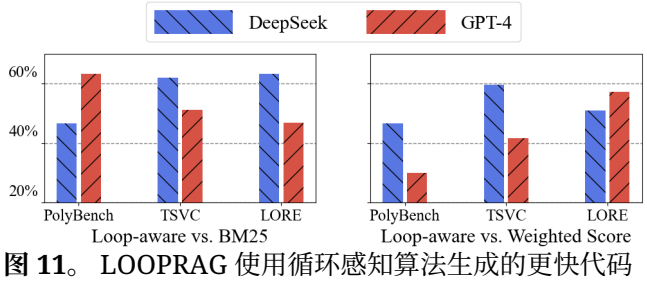

图 11。 LOOPRAG 使用循环感知算法生成的更快代码占比，与 BM25 和加权分数方法相比。

6.4.3 基于反馈的迭代生成。 为探究反馈机制和迭代生成的影响，我们使用第一轮和第二轮编译结果反馈测量 pass@k 的**提升**，并使用测试结果和性能排名反馈测量 pass@k 与加速比的提升。第一轮编译结果反馈对应步骤 2 中的生成，并与步骤 1 进行比较；第二轮则对应步骤 4，并与步骤 3 进行比较。测试结果和性能排名反馈对应步骤 4，并与步骤 2 进行比较。

表 7. 第一次和第二次编译后的 pass@k **提升**，以及 LOOPRAG 达到的测试结果和性能排名反馈。

反馈	LLMs	PolyBench	TSVC	LORE
		pass@k ↑		
第一轮编译	DeepSeek	14.44	**9.13**	**20.07**
	GPT-4	**21.67**	1.59	12.24
第二轮编译	DeepSeek	**3.33**	1.20	4.08
	GPT-4	**3.33**	**4.76**	**6.12**
测试结果和性能排名	DeepSeek	**5.00**	**6.24**	**5.34**
	GPT-4	1.67	4.14	5.10

表 7 展示了 LOOPRAG 中每轮迭代获得反馈后 pass@k 的提升情况。编译结果反馈成功地使 PolyBench、TSVC 和 LORE 上的平均 pass@k 分别提高了 21.39%（即 (14.44+3.33+21.67+3.33)/2）、8.34% 和 21.26%。与此同时，测试结果和性能排名反馈的效果较为适中，使 PolyBench、TSVC 和 LORE 上的平均 pass@k 分别提高了 3.34%、5.19% 和 5.22%。这些结果证明了这三种反馈在引导大语言模型生成正确代码方面的有效性。

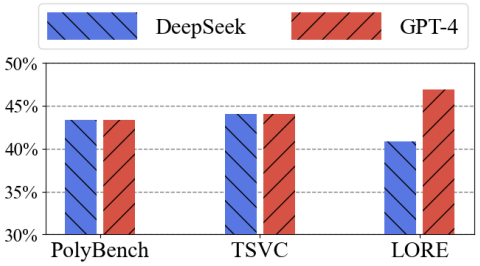

图12。 LOOPRAG 使用测试结果和性能排名反馈所生成的更快的代码的百分比。

如图12所示，测试结果和性能排名反馈结果分别使 PolyBench 、 TSVC 和 LORE 的代码平均提速 43.33% 、44.05% 和 43.88%。这对图7所示更快的代码占比的整体提升贡献显著，这证明了这两种反馈机制在增强大语言模型能力方面的有效性。

总的来说，这些结果揭示了基于反馈的迭代生成方法在引导大语言模型生成正确且高效代码方面的作用。

6.5 LOOPRAG 如何提升循环优化的质量?

为了解释 LOOPRAG 中的示例为何呈现出富有信息量的变换组合，以及 LOOPRAG 如何在示例的相似性与多样性之间取得平衡，我们在清单 2 和 3 中展示两个示例代码示例_*1* 和示例_*2*（优化后的版本见附录G）。这两段示例代码由循环感知算法检索，并用于示例演示，以引导大语言模型生成 PolyBench 中的优化代码*syrk*，如 §2.2 所示。与基础大语言模型 GPT-4 相比，加速比达到 71.60× 。

清单 2. 示例代码示例_*1*。

```
for i = 2到N for j = 0到M-1 S1: A[i-1][i] =A[i-2][i] +C[i][j] * 6; for k = 0到M-1 S2: A[k+1][k] =A[i][k] -C[k+1][i] * 4;
```

清单 3. 示例代码 示例_*2*。

```
对于 i = 0到L 对于 j = 0到i S1: A[i][j] =A[i][j] + 6; 对于 k = 0到L S2: A[i][k] = -A[k][i] +C语言[k] - 2;
```

将这两个示例代码与目标代码 *syrk*进行比较，可以发现循环属性存在许多差异，导致优化版本中的循环变换组合各不相同。例如，尽管这三段代码使用相同的数组 *A*，并表现出相同类型的依赖（WAW、WAR 和 RAW），但数组 *A* 的索引不同，导致依赖以不同模式出现。在清单 2 中，语句 *S1* 里的两个数组 *A* 在第一个维度上的索引不同，从而在迭代器 *i*的多次迭代之间，产生了其自身的循环携带依赖。然而，代码 *syrk* 中的两个数组*A* 具有相同的索引（图 2）。这一细微差异最终使示例代码示例_*1*的优化版本采用了循环倾斜。清单 3 中也存在类似情况，因此示例代码示例_*2*的优化版本采用了循环移位。这些差异使检索结果更加多样，有助于降低大语言模型获得潜在误导性指导的风险。

尽管示例代码之间的差异使示例演示中出现了各种类型的循环变换，但两个示例代码的优化版本中都采用了循环分块、循环融合和循环交换。例如，

这两个示例代码中的语句 *S1* 和 *S2* 位于同一个外层循环嵌套中，但处于不同的内层循环嵌套中，且循环边界相同。这种结构为循环融合创造了条件。此外，代码 *示例_1* 中的数组 *A* 和 *C* 的索引，以及代码 *示例_2* 中的数组 *A* 的索引，都与循环嵌套中迭代器的顺序不一致，因此适合进行循环交换。在这种情况下，这些源于循环属性的相似特征促成了上述有益的循环变换；大语言模型学会了这些变换，并将其应用于清单 1 中优化后的代码 *syrk*。

总的来说，LOOPRAG 检索到的这些示例代码，在循环属性方面兼具相似性与多样性。它们构成了 LOOPRAG 中的示例，其中优化版本的示例代码包含适当组合的循环变换，最终提升了循环优化的质量。

7 相关工作

循环数据集构建。大规模数据集是训练代码优化模型和评估编译器的宝贵资源。近期已有多种方法被提出，用于构建循环数据集。LORE [13] 维护了大量从基准测试套件、库和用于评估编译器的真实应用中提取的 C 语言 *for* 循环嵌套。ANGHABENCH [17]结合网络爬取与类型推断，从开源代码仓库生成 C 程序，以缩减代码规模。YARPGen [35] 是一种生成式模糊测试器，提出了三步循环生成策略，以触发循环优化并发现编译器缺陷。Baghdadi 等人 [4] 开发了一种随机代码生成器，用于创建合成数据，以训练代码优化的代价模型；Mezdour 等人 [38]则采用相同方法生成数据集，用于训练循环交换的深度学习模型。COLA-Gen [7]提出了一种循环生成器，能够以参数化方式创建用于代码优化的合成训练数据集。然而，现有循环生成方法在循环属性方面的多样性有限，也无法有效触发循环优化。因此，我们提出一种参数驱动方法来解决这一问题。

基于大语言模型的代码优化。由于对代码具有深入的语义理解，大语言模型在各种代码优化任务中展现出巨大潜力 [25]。Cummins 等人 [15, 16, 28] 旨在利用编译器生成的反馈来引导大语言模型优化 LLVM 汇编的代码大小。RALAD [55] 提出了一种检索增强的大语言模型方法，用于优化高级综合（HLS）程序。SBLLM [23] 使大语言模型能够通过进化搜索迭代地改进和优化代码。PCAOT [44] 利用指令提示和思维链，引导大语言模型使用 OpenMP 提升代码性能；而 LLM-Vectorizer [46] 则采用基于人工智能代理的方法，并结合测试驱动反馈，利用大语言模型生成向量化代码。然而，由于缺乏性能反馈或编译器中用于指导有效代码优化策略的内在机制（如代价模型或启发式方法），大语言模型往往会提出次优或错误的优化方案。我们的工作旨在弥合这一差距：通过提供信息丰富的示例和反馈机制，迭代引导大语言模型生成正确且高效的代码。

8 结论

本文提出了 LOOPRAG，这是一个旨在通过检索增强生成提升静态控制部分（SCoP）上循环变换优化效果的框架。我们的研究强调，提供信息丰富的示例对于指导大语言模型执行代码优化至关重要。为应对这一挑战，我们引入了一种参数驱动方法，用于合成作为示例来源的数据集。为了有效获取有价值的示例，我们提出了一种循环感知的代码检索算法。此外，我们还提出了一种基于反馈的迭代生成方法，以指导大语言模型生成正确且高效的代码。评估结果表明，LOOPRAG 的表现优于最先进的编译器、有代表性的大语言模型以及基于大语言模型的模型，在代码优化方面取得了显著提升。未来，我们计划纳入更多代码优化技术并改进生成步骤，以克服目前在静态控制部分（SCoP）上的局限，提升优化代码的可靠性，并实现整个流程的自动化。

参考文献

[1] Mayank Agarwal、Yikang Shen、Bailin Wang、Yoon Kim 和 Jie Chen。2024年。《结构化代码表示使代码语言模型能够以数据高效的方式进行适配》。arXiv:2401.10716 [cs.CL]https://arxiv.org/abs/2401 10716。Randy Allen 和 Ken Kennedy。[2] 2002年。面向现代架构的优化编译器：一种基于依赖的方法。Morgan Kaufmann出版社。https://cir.nii.ac.jp/crid/1130282272927219456[3] Sotiris Apostolakis、Ziyang Xu、Greg Chan、Simone Campanoni 和 David I August。2020年。观点：推测式自动并行化的一种合理方法。载于第二十五届编程语言与操作系统体系结构支持国际会议论文集。351–367。[4] Riyadh Baghdadi、Massinissa Merouani、Mohamed-Hicham LEGHET- TAS、Kamel Abdous、Taha Arbaoui、Karima BENATCHBA 和 Saman amarasinghe。2021年。用于自动代码优化的基于深度学习的成本模型。载于机器学习与系统会议论文集，A. Smola、A. Dimakis 和 I. Stoica（编），第3卷。181–193。https://proceedings.mlsys.org/paper_files/paper/2021/file/d9387b6d643efb25132be36f7b908d96-Paper.pdf [5] Parinaz Barakhshan 和 Rudolf Eigenmann。2023年。CaRV——通过捕获、重放、验证加速程序优化。载于国际高性能计算、网络、存储与分析会议 *SC '23* 研讨会论文集 （美国科罗拉多州丹佛）*(SC-W '23)*。计算机协会，纽约，美国，654–662。https://doi.org/10.1145/3624062.3624140[6] Cédric Bastoul、Albert Cohen、Sylvain Girbal、Saurabh Sharma 和 Olivier Temam。2004年。将多面体循环变换付诸

实践。载于并行计算中的语言与编译器，Lawrence Rauchwerger（编）。Springer Berlin Heidelberg，德国海德堡，209–225。https://doi.org/10.1007/978-3-540-24644-2_14[7] Maksim Berezov、Corinne Ancourt、Justyna Zawalska 和 Maryna Savchenko。2022年。COLA-Gen：用于基准测试自动代码生成的主动学习技术。载于第*13*届多核架构并行编程与运行时管理技术研讨会暨第*11*届多核嵌入式计算平台设计工具与架构研讨会（*PARMA-DITAM 2022*）（开放获取信息学系列（*OASIcs*），第*100*卷），Francesca Palumbo、João Bispo 和 Stefano Cherubin（编）。Dagstuhl城堡—莱布尼茨信息学中心，德国 Dagstuhl，3:1–3:14。https: //doi.org/ 10 4230 2022 3 。/OASIcs.PARMA-DITAM.。[8] Uday Bondhugula、Albert Hartono、J. Ramanujam 和 P. Sadayap- pan。2008年。一种实用的自动多面体并行化器和局部性优化器。*SIGPLAN* 通讯。43, 6（2008年6月），101–113。https: //doi.org/10.1145/1379022.1375595[9] Tom Brown、Benjamin Mann、Nick Ryder、Melanie Subbiah、Jared D Kaplan、Prafulla Dhariwal、Arvind Neelakantan、Pranav Shyam、Girish Sastry、Amanda Askell、Sandhini Agarwal、Ariel Herbert-Voss、Gretchen Krueger、Tom Henighan、Rewon Child、Aditya Ramesh、Daniel Ziegler、Jeffrey Wu、Clemens Winter、Chris Hesse、Mark Chen、Eric Sigler、Mateusz Litwin、Scott Gray、Benjamin Chess、Jack Clark、Christopher Berner、Sam McCandlish、Alec Radford、Ilya Sutskever 和 Dario Amodei。2020年。语言模型是少样本学习器。载于神经信息处理系统进展，H. Larochelle、M. Ranzato、R. Hadsell、M.F. Balcan 和 H. Lin（编），第33卷。Curran Associates, Inc.，1877–1901。https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf [10] D. Callahan、J. Dongarra 和 D. Levine。1988年。向量化编译器：测试套件与结果。载于超级计算 *'88*：*1988*年 *ACM/IEEE* 超级计算会议论文集，第*I*卷。98–105。https://doi. org/10.1109/SUPERC.1988.44642[11] Junjie Chen、Jibesh Patra、Michael Pradel、Yingfei Xiong、Hongyu Zhang、Dan Hao 和 Lu Zhang。2020年。编译器测试综述。*ACM*计算评论53, 1，文章4（2020年2月），36页。https://doi. org/ 10 1145 3363562 。/Zimin Chen、Sen Fang 和 Martin Monperrus。[12] 2024年。Supersonic：学习生成 C/C++ 源代码优化。*IEEE*软件工程汇刊50, 11（2024年），2849–2864。https: //doi.org/ 10 1109 2024 3423769 。/TSE.。Zhi Chen、Zhangxiaowen Gong、[13] Justin Josef Szaday、David C. Wong、David Padua、Alexandru Nicolau、Alexander V. Veidenbaum、Neftali Watkinson、Zehra Sura、Saeed Maleki、Josep Torrellas 和 Gerald DeJong。2017年。LORE：用于编译器评估的循环仓库。载于*2017* 年 *IEEE* 工作负载表征国际研讨会（*IISWC*）。219 228 10 1109 2017 8167779 –。[14] https://doi.org/。/IISWC.。Berkeley Churchill、Oded Padon、Rahul Sharma 和 Alex Aiken。2019年。用于等价性检查的语义程序对齐。载于第*40*届 *ACM SIGPLAN* 编程语言设计与实现会议论文集（美国亚利桑那州菲尼克斯）*(PLDI 2019)*。计算机协会，纽约，美国，1027–1040。https://doi.org/10.1145/3314221.3314596[15] Chris Cummins、Volker Seeker、Dejan Grubisic、Mostafa Elhoushi、Youwei Liang、Baptiste Roziere、Jonas Gehring、Fabian Gloeckle、Kim Hazelwood、Gabriel Synnaeve 和 Hugh Leather。2023年。用于编译器优化的大型语言模型。arXiv:2309.07062 [cs.PL] https://arxiv.org/abs/2309.07062[16] Chris Cummins、Volker Seeker、Dejan Grubisic、Baptiste Roziere、Jonas Gehring、Gabriel Synnaeve 和 Hugh Leather。2024年。Meta 大型语言模型编译器：编译器优化的基础模型。arXiv:2407.02524 [cs.PL] https://arxiv.org/abs/2407.02524

[17] Anderson Faustino da Silva、Bruno Conde Kind、José Wesley de Souza Magalhães、Jerônimo Nunes Rocha、Breno Campos Ferreira Guimarães 和 Fernando Magno Quinão Pereira。2021年。ANG-HABENCH：一个包含一百万个可编译 C 基准测试的套件，用于代码尺寸缩减。载于*2021 IEEE/ACM* 代码生成与优化国际研讨会（*CGO*）。378–390。https: //doi.org/ 10 1109 51591 2021 9370322 。/CGO。. L. Dagum 和 R. Menon。[18] 1998. OpenMP：共享内存编程的行业标准 API。*IEEE* 计算科学与工程 5, 1 1998 46 55 10 1109 99 660313 ( ), –。https://doi.org/。/。Alain Darte、Yves Robert 和 Frederic Vivien。[19] 2000. 调度与自动并行化（第1版）。Birkhauser Boston, Inc., 美国。https: //dl.acm.org/doi/abs/ 10 5555 556139 。/ DeepSeek-AI、Aixin Liu、Bei Feng、[20] Bin Wang、Bingxuan Wang、Bo Liu、Chenggang Zhao、Chengqi Dengr、Chong Ruan、Damai Dai、Daya Guo、Dejian Yang、Deli Chen、Dongjie Ji、Erhang Li、Fangyun Lin、Fuli Luo、Guangbo Hao、Guanting Chen、Guowei Li、H. Zhang、Hanwei Xu、Hao Yang、Haowei Zhang、Honghui Ding、Huajian Xin、Huazuo Gao、Hui Li、Hui Qu、J. L. Cai、Jian Liang、Jianzhong Guo、Jiaqi Ni、Jiashi Li、Jin Chen、Jingyang Yuan、Junjie Qiu、Junxiao Song、Kai Dong、Kaige Gao、Kang Guan、Lean Wang、Lecong Zhang、Lei Xu、Leyi Xia、Liang Zhao、Liyue Zhang、Meng Li、Miaojun Wang、Mingchuan Zhang、Minghua Zhang、Minghui Tang、Mingming Li、Ning Tian、Panpan Huang、Peiyi Wang、Peng Zhang、Qihao Zhu、Qinyu Chen、Qiushi Du、R. J. Chen、R. L. Jin、Ruiqi Ge、Ruizhe Pan、Runxin Xu、Ruyi Chen、S. S. Li、Shanghao Lu、Shangyan Zhou、Shanhuang Chen、Shaoqing Wu、Shengfeng Ye、Shirong Ma、Shiyu Wang、Shuang Zhou、Shuiping Yu、Shunfeng Zhou、Size Zheng、T. Wang、Tian Pei、Tian Yuan、Tianyu Sun、W. L. Xiao、Wangding Zeng、Wei An、Wen Liu、Wenfeng Liang、Wenjun Gao、Wentao Zhang、X. Q. Li、Xiangyue Jin、Xianzu Wang、Xiao Bi、Xiaodong Liu、Xiaohan Wang、Xiaojin Shen、Xiaokang Chen、Xiaosha Chen、Xiaotao Nie、Xiaowen Sun、Xiaoxiang Wang、Xin Liu、Xin Xie、Xingkai Yu、Xinnan Song、Xinyi Zhou、Xinyu Yang、Xuan Lu、Xuecheng Su、Y. Wu、Y. K. Li、Y. X. Wei、Y. X. Zhu、Yanhong Xu、Yanping Huang、Yao Li、Yao Zhao、Yaofeng Sun、Yaohui Li、Yaohui Wang、Yi Zheng、Yichao Zhang、Yiliang Xiong、Yilong Zhao、Ying He、Ying Tang、Yishi Piao、Yixin Dong、Yixuan Tan、Yiyuan Liu、Yongji Wang、Yongqiang Guo、Yuchen Zhu、Yuduan Wang、Yuheng Zou、Yukun Zha、Yunxian Ma、Yuting Yan、Yuxiang You、Yuxuan Liu、Z. Z. Ren、Zehui Ren、Zhangli Sha、Zhe Fu、Zhen Huang、Zhen Zhang、Zhenda Xie、Zhewen Hao、Zhihong Shao、Zhiniu Wen、Zhipeng Xu、Zhongyu Zhang、Zhuoshu Li、Zihan Wang、Zihui Gu、Zilin Li 和 Ziwei Xie。2024年。DeepSeek-V2：一种强大、经济且高效的混合专家语言模型。arXiv: 2405 04434 [ ] 2405 04434 . cs.CL https://arxiv.org/abs/。Luke Dramko、Claire Le Goues 和 Edward J. Schwartz。[21] 2025年。面向神经反编译器的快速、细粒度等价性检查。arXiv: 2501 04811 [ ] 2501 04811 . cs.LG [22] 2005. https://arxiv.org/abs/。Paul Feautrier。. 多面体模型中的自动并行化。数据并行编程模型：基础、*HPF* 实现与科学应用（2005），79–103。https://doi.org/10.1007/3- 540-61736-1 44_[23] Shuzheng Gao、Cuiyun Gao、Wenchao Gu 和 Michael Lyu。2025年。用于代码优化的基于搜索的大语言模型。载于*2025 IEEE/ACM* 第*47*届软件工程国际会议（*ICSE*）。IEEE 计算机协会，洛斯阿拉米托斯，加利福尼亚州，美国，254–266。https://doi.org/10.1109/ICSE55347.2025.00021[24] Sylvain Girbal、Nicolas Vasilache、Cédric Bastoul、Albert Cohen、David Parello、Marc Sigler 和 Olivier Temam。2006年。面向深度并行和存储层次结构的循环变换半自动组合。《国际并行编程杂志》34 (2006)，261–317。https://doi.org/10.1007/s10766-006-0012-3[25] Jingzhi Gong、Vardan Voskanyan、Paul Brookes、Fan Wu、Wei Jie、Jie Xu、Rafail Giavrimis、Mike Basios、Leslie Kanthan 和 Zheng Wang。2025年。用于代码优化的语言模型：综述、挑战

与未来方向。arXiv:2501.01277 [cs.SE] https://arxiv.org/abs/2501.01277[26] Clinton Gormley 和 Zachary Tong。2015年。*Elasticsearch*：权威指南（第1版）。O'Reilly Media, Inc. https://dl.acm.org/doi/abs/10. 5555/2904394[27] Tobias Grosser、Armin Groesslinger 和 Christian Lengauer。2012年。Polly——在低级中间表示上执行多面体优化。*Parallel Processing Letters* 22, 04 (2012)，1250010。[28] Dejan Grubisic、Chris Cummins、Volker Seeker 和 Hugh Leather。2024年。面向大语言模型的编译器生成反馈。arXiv: 2403 14714 [ ] 2403 14714 . cs.PL [29] https://arxiv.org/abs/。Shubhani Gupta、Aseem Saxena、Anmol Mahajan 和 Sorav Bansal。2018年。通过不变式草图和查询分解有效使用 SMT 求解器进行程序等价性检查。载于可满足性测试理论与应用——*SAT 2018*：第*21*届国际会议，*SAT 2018*，作为联合逻辑会议 *FloC 2018* 的一部分于*2018*年*7*月*9*日至*12*日在英国牛津举行，会议论文集 *21*。施普林格，365–382。https://doi.org/ 10 1007 978 3 319 94144 8_22 。/ - - - - [30] M. Kandemir 、 A. Choudhary 、 J. Ramanujam 和 P. Banerjee。1998 年。在集成框架中通过循环和数据变换提升局部性。载于第*31*届 *ACM/IEEE* 微体系结构国际研讨会论文集。285–296。https://doi.org/10.1109/MICRO.1998.742790[31] Sumith Kulal、Panupong Pasupat、Kartik Chandra、Mina Lee、Oded Padon、Alex Aiken 和 Percy S Liang。2019年。SPoC：基于搜索的伪代码转代码。载于神经信息处理系统进展，H. Wallach、H. Larochelle、A. Beygelzimer、F. d'Alché-Buc、E. Fox 和 R. Garnett（编），第32卷。Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2019/file/7298332f04ac004a0ca44cc69ecf6f6b-Paper.pdf[32] Monica S. Lam 和 Michael E. Wolf。2004年。一种数据局部性优化算法。*SIGPLAN* 通讯39, 4（2004年4月），442–459。https://doi.org/ 10.1145/989393.989437[33] Jiawei Liu、Chunqiu Steven Xia、Yuyao Wang 和 LINGMING ZHANG。2023年。ChatGPT 生成的代码真的正确吗？对用于代码生成的大语言模型进行严格评估。载于神经信息处理系统进展，A. Oh、T. Naumann、A. Globerson、K. Saenko、M. Hardt 和 S. Levine（编），第36卷。Curran Associates, Inc.，21558–21572。https://proceedings.neurips.cc/paper_files/paper/2023/file/43e9d647ccd3e4b7b5baab53f0368686-Paper-Conference.pdf[34] Tie-Yan Liu。2009年。信息检索中的排序学习。基础与趋势® 信息检索3, 3 (2009)，225–331。https://doi.org/ 10 1561 1500000016 。/ [35] Vsevolod Livinskii、Dmitry Babokin 和 John Regehr。2023年。对 C++和数据并行语言编译器中的循环优化进行模糊测试。*ACM* 程序设计语言论文集 7, PLDI，论文编号181（2023年6月），22页。https://doi.org/ 10 1145 3591295 。/ Nuno P. Lopes、Juneyoung Lee、[36] Chung-Kil Hur、Zhengyang Liu 和 John Regehr。2021年。Alive2：针对 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（线上，加拿大）（*PLDI 2021*）。计算机协会，纽约，纽约州，美国，65–79。https://doi.org/10.1145/3453483.3454030[37] Aman Madaan、Niket Tandon、Prakhar Gupta、Skyler Hallinan、Luyu Gao、Sarah Wiegreffe、Uri Alon、Nouha Dziri、Shrimai Prabh umoye、Yiming Yang、Shashank Gupta、Bodhisattwa Prasad Majumder、Katherine Hermann、Sean Welleck、Amir Yazdanbakhsh 和 Peter Clark。2023年。Self-Refine：通过自我反馈进行迭代式精炼。载于神经信息处理系统进展，A. Oh、T. Naumann、A. Globerson、K. Saenko、M. Hardt 和 S. Levine（编），第36卷。Curran Associates, Inc.，46534–46594。https://proceedings.neurips.cc/paper_files/paper/2023/file/91edff07232fb1b55a505a9e9f6c0ff3-Paper-Conference.pdf

[38] Lina Mezdour、Khadidja Kadem、Massinissa Merouani、Amina Selma Haichour、Saman Amarasinghe 和 Riyadh Baghdadi。2023年。《用于循环交换的深度学习模型》。载于第*32*届 *ACM SIGPLAN* 编译器构造国际会议论文集 （加拿大魁北克省蒙特利尔）（*CC 2023*）。计算机协会，美国纽约州纽约，50–60。https://doi.org/10.1145/3578360.3580257

[39] OpenAI，Josh Achiam、Steven Adler、Sandhini Agarwal、Lama Ah- mad、Ilge Akkaya、Florencia Leoni Aleman、Diogo Almeida、Janko Altenschmidt、Sam Altman、Shyamal Anadkat、Red Avila、Igor Babuschkin、Suchir Balaji、Valerie Balcom、Paul Baltescu、Haiming Bao、Mohammad Bavarian、Jeff Belgum、Irwan Bello、Jake Berdine、Gabriel Bernadett-Shapiro、Christopher Berner、Lenny Bogdonoff、Oleg Boiko、Madelaine Boyd、Anna-Luisa Brakman、Greg Brockman、Tim Brooks、Miles Brundage、Kevin Button、Trevor Cai、Rosie Camp- bell、Andrew Cann、Brittany Carey、Chelsea Carlson、Rory Carmichael、Brooke Chan、Che Chang、Fotis Chantzis、Derek Chen、Sully Chen、Ruby Chen、Jason Chen、Mark Chen、Ben Chess、Chester Cho、Casey Chu、Hyung Won Chung、Dave Cummings、Jeremiah Currier、Yunx- ing Dai、Cory Decareaux、Thomas Degry、Noah Deutsch、Damien Deville、Arka Dhar、David Dohan、Steve Dowling、Sheila Dunning、Adrien Ecoffet、Atty Eleti、Tyna Eloundou、David Farhi、Liam Fedus、Niko Felix、Simón Posada Fishman、Juston Forte、Isabella Fulford、Leo Gao、Elie Georges、Christian Gibson、Vik Goel、Tarun Gogineni、Gabriel Goh、Rapha Gontijo-Lopes、Jonathan Gordon、Morgan Graf- stein、Scott Gray、Ryan Greene、Joshua Gross、Shixiang Shane Gu、Yufei Guo、Chris Hallacy、Jesse Han、Jeff Harris、Yuchen He、Mike Heaton、Johannes Heidecke、Chris Hesse、Alan Hickey、Wade Hickey、Peter Hoeschele、Brandon Houghton、Kenny Hsu、Shengli Hu、Xin Hu、Joost Huizinga、Shantanu Jain、Shawn Jain、Joanne Jang、Angela Jiang、Roger Jiang、Haozhun Jin、Denny Jin、Shino Jomoto、Billie Jonn、Heewoo Jun、Tomer Kaftan、Łukasz Kaiser、Ali Kamali、Ingmar Kan- itscheider、Nitish Shirish Keskar、Tabarak Khan、Logan Kilpatrick、Jong Wook Kim、Christina Kim、Yongjik Kim、Jan Hendrik Kirchner、Jamie Kiros、Matt Knight、Daniel Kokotajlo、Łukasz Kondraciuk、An- drew Kondrich、Aris Konstantinidis、Kyle Kosic、Gretchen Krueger、Vishal Kuo、Michael Lampe、Ikai Lan、Teddy Lee、Jan Leike、Jade Leung、Daniel Levy、Chak Ming Li、Rachel Lim、Molly Lin、Stephanie Lin、Ma- teusz Litwin、Theresa Lopez、Ryan Lowe、Patricia Lue、Anna Makanju、Kim Malfacini、Sam Manning、Todor Markov、Yaniv Markovski、Bianca Martin、Katie Mayer、Andrew Mayne、Bob McGrew、Scott Mayer McK- inney、Christine McLeavey、Paul McMillan、Jake McNeil、David Medina、Aalok Mehta、Jacob Menick、Luke Metz、Andrey Mishchenko、Pamela Mishkin、Vinnie Monaco、Evan Morikawa、Daniel Mossing、Tong Mu、Mira Murati、Oleg Murk、David Mély、Ashvin Nair、Rei- ichiro Nakano、Rajeev Nayak、Arvind Neelakantan、Richard Ngo、Hyeonwoo Noh、Long Ouyang、Cullen O'Keefe、Jakub Pachocki、Alex Paino、Joe Palermo、Ashley Pantuliano、Giambattista Parascandolo、Joel Parish、Emy Parparita、Alex Passos、Mikhail Pavlov、Andrew Peng、Adam Perelman、Filipe de Avila Belbute Peres、Michael Petrov、Hen- rique Ponde de Oliveira Pinto、Michael、Pokorny、Michelle Pokrass、Vitchyr H. Pong、Tolly Powell、Alethea Power、Boris Power、Elizabeth Proehl、Raul Puri、Alec Radford、Jack Rae、Aditya Ramesh、Cameron Raymond、Francis Real、Kendra Rimbach、Carl Ross、Bob Rotsted、Henri Roussez、Nick Ryder、Mario Saltarelli、Ted Sanders、Shibani San- turkar、Girish Sastry、Heather Schmidt、David Schnurr、John Schulman、Daniel Selsam、Kyla Sheppard、Toki Sherbakov、Jessica Shieh、Sarah Shoker、Pranav Shyam、Szymon Sidor、Eric Sigler、Maddie Simens、Jordan Sitkin、Katarina Slama、Ian Sohl、Benjamin Sokolowsky、Yang Song、Natalie Staudacher、Felipe Petroski Such、Natalie Summers、Ilya Sutskever、Jie Tang、Nikolas Tezak、Madeleine B. Thompson、Phil Tillet、Amin Tootoonchian、Elizabeth Tseng、Preston Tuggle、Nick Turley、Jerry Tworek、Juan Felipe Cerón Uribe、Andrea Vallone、Arun Vi- jayvergiya、Chelsea Voss、Carroll Wainwright、Justin Jay Wang、Alvin Wang、Ben Wang、Jonathan Ward、Jason Wei、CJ Weinmann、Akila

Welihinda、Peter Welinder、Jiayi Weng、Lilian Weng、Matt Wiethoff、Dave Willner、Clemens Winter、Samuel Wolrich、Hannah Wong、Lau- ren Workman、Sherwin Wu、Jeff Wu、Michael Wu、Kai Xiao、Tao Xu、Sarah Yoo、Kevin Yu、Qiming Yuan、Wojciech Zaremba、Rowan Zellers、Chong Zhang、Marvin Zhang、Shengjia Zhao、Tianhao Zheng、Jun- tang Zhuang、William Zhuk 和 Barret Zoph。2024年。《GPT-4技术报告》。arXiv:2303.08774 [cs.CL] https://arxiv.org/abs/2303.08774[40] Kexin Pei、David Bieber、Kensen Shi、Charles Sutton 和 Pengcheng Yin。2023年。大语言模型能否推理程序不变量？载于第*40*届国际机器学习会议论文集（机器学习研究论文集，第*202*卷），由 Andreas Krause、Emma Brunskill、Kyunghyun Cho、Barbara Engel- hardt、Sivan Sabato 和 Jonathan Scarlett 编。PMLR，27496–27520。https://proceedings.mlr.press/v202/pei23a.html [41] Sebastian Pop、Albert Cohen、Cédric Bastoul、Sylvain Girbal、Georges- André Silber 和 Nicolas Vasilache。2006年。GRAPHITE：面向 GCC 的多面体分析与优化。载于*2006*年 *GCC* 开发者峰会论文集，第6卷。90–91。[42] Louis-Noel Pouchet 和 Tomofumi Yuki。2023年。PolyBench/C 4.2.1。https://polybench.sourceforge.net [43] Stephen Robertson 和 Hugo Zaragoza。2009年。概率相关性框架：BM25及其他方法。基础与趋势® 信息检索3，4（2009），333–389。https://doi.org/10.1561/ 1500000019[44] Miguel Romero Rosas、Miguel Torres Sanchez 和 Rudolf Eigenmann。2024年。应该让人工智能优化你的代码吗？当前大语言模型与经典优化编译器的比较研究。arXiv: 2406 12146 [ ] 2406 12146 . cs.AI [45] https://arxiv.org/abs/ 。2024 Aivin V. Solatorio。 。GISTEmbed：面向文本嵌入微调的引导式样本内训练负样本选择。arXiv: 2402 16829 [ ] 2402 16829 . cs.LG [46] https://arxiv.org/abs/ 。Jubi Taneja、Avery Laird、Cong Yan、Madan Musuvathi 和 Shuvendu K. Lahiri。2024年。LLM-Vectorizer：基于大语言模型的经验证循环向量化器。arXiv:2406.04693 [cs.SE] https://arxiv.org/abs/2406.04693[47] Nandan Thakur、Nils Reimers、Andreas Rücklé、Abhishek Srivastava 和 Iryna Gurevych。2021年。BEIR：用于信息检索模型零样本评估的异构基准。载于神经信息处理系统第三十五届会议数据集与基准赛道（第二轮）。https://openreview.net/forum?id=wCu6T5xFjeJ [48] Sven Verdoolaege。2010年。isl：面向多面体模型的整数集库。载于数学软件——*ICMS 2010*，由 Komei Fukuda、Joris Hoeven、Michael Joswig 和 Nobuki Takayama 编。《计算机科学讲义》，第6327卷。施普林格，299–302。https://doi.org/10. 1007/978-3-642-15582-6 49_[49] Sven Verdoolaege、Juan Carlos Juega、Albert Cohen、José Igna- cio Gómez、Christian Tenllado 和 Francky Catthoor。2013年。面向 CUDA 的多面体并行代码生成。*ACM*体系结构、代码优化汇刊9，4，文章54（2013年1月），23页。https://doi.org/10.1145/ 2400682.2400713[50] Sven Verdoolaege 和 Tobias Grosser。2012年。多面体提取工具。载于第二届多面体编译技术国际研讨会（*IMPACT'12*）。http://impact.gforge.inria.fr/impact2012/ 第二届多面体编译技术国际研讨会，IMAPCT 2012；会议日期：2012年1月23日至2012年1月23日。[51] Sven Verdoolaege、Serge Guelton、Tobias Grosser 和 Albert Cohen。2014年。调度树。载于*IMPACT*——与 *HiPEAC* 相关的第四届多面体编译技术研讨会。ACM，奥地利维也纳。 00914894 https://inria.hal.science/hal-[52]

Xuezhi Wang、Jason Wei、Dale Schuurmans、Quoc V Le、Ed H. Chi、Sharan Narang、Aakanksha Chowdhery 和 Denny Zhou。2023年。自一致性提升语言模型中的思维链推理。载于第十一届国际学习表征会议。https://openreview.net/forum?id=1PL1NIMMrw

[53] Michael Joseph Wolfe。1978年。提升程序固有并行性的技术。第929号。伊利诺伊大学厄巴纳-香槟分校计算机科学系。[54] 肖诗涛、刘征、张沛田、Niklas Muennighoff、连德福和聂建云。2024年。C-Pack：面向通用中文嵌入的资源打包。arXiv:2309.07597 [cs.CL] https://arxiv.org/abs/ 2309.07597[55] 徐昊成、胡浩天和黄思涛。2024年。利用检索增强大语言模型优化高层次综合设计。载于*2024*年 *IEEE* 大语言模型辅助设计研讨会（*LAD*）。1–5。https: //doi.org/ 10 1109 62341 2024 10691855 。 /LAD 。 。杨一清、[56] Corinne Ancourt 和 François Irigoin。1995年。用于循环变换的最小数据依赖抽象。载于并行计算的语言与编译器，Keshav Pingali、Utpal Banerjee、David Gelernter、Alex Nicolau 和 David Padua（编）。Springer Berlin Heidelberg，柏林，海德堡，201–216。https: //doi.org/ 10 1007 0025880 。 /BFb 张卓[57] 生、Aston Zhang、李沐和 Alex Smola。2023年。大语言模型中的自动思维链提示。载于第十一届国际学习表征会议。https://openreview.net/forum?id=5NTt8GFjUHkr[58] Oleksandr Zinenko、Sven Verdoolaege、Chandan Reddy、Jun Shirako、Tobias Grosser、Vivek Sarkar 和 Albert Cohen。2018年。在仿射调度中对并行性与时间/空间局部性的冲突需求进行建模。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，纽约州，美国，3–13。https://doi.org/10. 1145/3178372.3179507

数据集合成中的循环参数

在本附录中，我们将解释参数驱动方法中用于配置循环属性的十个循环参数的含义和取值范围：

1) **迭代器边界** 指定迭代器出现在循环边界中的概率，该概率在循环嵌套的每个后续层级减半。此参数的值从 {20%, 40%, 60%} 中随机选取。

2) **循环深度** 指定静态控制部分的最大循环深度。此参数的值从 2 到 4 中随机选取。 3) **语句索引** 指定循环嵌套每一层中的最大循环分支数，这决定了循环中语句的密度。此参数的值从 1 到 3 中随机选取。 4) **语句数量** 指定静态控制部分中的语句数量。此参数的值从 1 到 6 中随机选取。 5) **依赖距离** 指定依赖距离向量每个维度中边界的最大绝对值。此参数的值从 1 到 2 中随机选取。 6) **读依赖**指定每条语句中 WAR 和读后写依赖的最大数量。此参数的值从 1 到 3 中随机选取。 7) **写依赖** 指定每条语句存在写后写依赖的概率。此参数的值从 {20%, 40%, 60%} 中随机选取。 8) **数组列表** 指定每条语句中写操作和读操作可选数组的数量，表示数组的多样性。此参数的值从 1 到 3 中随机选取。 9) **读数组** 指定每条语句对数组进行读操作的最大次数。此参数的值从 {1, 3, 5} 中随机选取。 10) **数组索引** 指定数组索引矩阵中常数系数边界的最大绝对值。此参数的值从 1 到 2 中随机选取。

尽管在不同迭代中访问数组的相同索引可能导致依赖，但当数组相关参数（即**数组列表**、**读数组**和**数组索引**）的值被随机分配时，这种情况很少发生。因此，我们利用依赖相关参数（即**依赖距离**、**读依赖**和**写依赖**）为每条语句手动配置依赖，以补充数组访问自然产生的依赖，从而使示例代码更加多样化。此外，我们将依赖距离限制为常量，并生成迭代器系数设为 1 的数组索引。

算法*1*：根据循环参数进行示例代码合成 输入：*IteratorBound*、*LoopDepth*、*StmtIndex*、*NStmts*、*DepDistance*、*ReadDep*、*WriteDep*、*ArrayList*、*ReadArr*、*ArrIndexes*、*NameList*、*SizeList* 输出：代码 *1 SchedulesMtx = Rand(NStmts, LoopDepth, StmtIndex)*；*2 SchedulesMtx = ReOrder(SchedulesMtx)*；*3 ItBounds = GenItBounds(IteratorBound, SchedulesMtx)*；*4 for i = 0 to NStmts - 1 do 5 Names =NameList[ArrayList]*；*6 Indexes =GenIndex(ArrIndexes)*；*7 ArrsNoDep[i].append([Names, Indexes])*；*8 SourceIds = DepAssign(WriteDep,ReadDep,NStmts)*；*9 ArrsDep[i].append([SourceIds])*；*10 Arrs = PriorityAssign(ArrsNoDep, ArrsDep)*；*11 for i = 0to NStmts - 1 do 12 for Arr in Arrs[i] do 13 ID = SearchSource(Arr[SourceId])*；*14 ArrsSize[Arr[Name]] = SizeAssign(Arr, SizeList)*；*15 for i = 0to NStmts - 1 do 16 Iterators.append(ExtractIterator(SchedulesMtx[i]))*；*17 ScheduleTree.append(SchedulesMtx[i])*；*18 Arrs = CheckArrDep(Arrs)*；*19 Bounds = CreateBoundsData(ItBounds)*；*20 for i = 0to NStmts - 1 do 21 Arrs[i][Indexes] = ReGenerateIndex(DepDistance)*；*22 Bounds.append(GetBounds(Arrs[i]))*；*23 StmtsExpr.append(ConvertToExprs(Arrs[i]))*；*24 for ArrSize in ArrsSize do 25 Bounds.append(GetBounds(ArrSize))*；*26 for It in Iterators do 27 LB, UB = CalcBound(It, Bounds)*；*28 LoopsExpr.append(GenForLoop(LB, UB))*；*29 Structure =UpdateTree(ScheduleTree)*；*30 SCoP = GetLoop(Structure, LoopsExpr, StmtsExpr)*；*31 Code = GetCode(SCoP)*；*32 return Code*

B 数据集合成中的示例代码合成算法

算法1展示了 LOOPRAG 中根据循环参数生成示例代码的过程。该过程从十个参数以及两个为数组准备的列表开始，列表中包含名称和大小。首先，为每条语句配置具有随机值和维度的循环调度，然后根据偏序对其重新排序（第 1–2 行）。调度的构造会分配循环嵌套中迭代器的顺序，从而生成迭代器边界（第 3 行）。

名为 *ArrsNoDep*的数组作为名为 *ArrsDep*的数组的替代项生成，并带有数组名称

和索引（第 5—7 行）。名为 *ArrsDep* 的数组根据依赖相关参数（第 8—9 行）创建。如果该数组与写后写依赖相关，则生成 *ArrsDep* 作为目标数组，其 ID 为该写后写依赖中源数组所在语句的 ID。对于写后读依赖或读后写依赖， *ArrsDep* 对应于写后读依赖或读后写依赖中的数组读取，并包含与数组写入对应的语句 ID。

所有数组都按优先级分配，以便在必要时消除 *ArrsNoDep* 备选方案（第 10 行）。其余数组根据依赖关系，通过*ArrsDep* 中的语句ID建立关系，并从列表中随机获取数组大小（第 12–14 行）。随后检查这些数组相关依赖的有效性，并解决循环依赖等矛盾（第 18 行）。然后重新生成与有效依赖相关的数组索引（第 21 行）。

数组大小、数组索引和循环迭代器的值（分别见第 25 、 22 和 19 行）用于计算循环边界（第 27 行）。如果无解，生成过程就会中止。

收集迭代器和数组名称等所有纹理信息，用于语句表达式（第 23 行）和循环表达式（第 28 行）。根据调度树构建代码的静态控制部分（第 29 行），并通过添加初始化、输出提取以及 PolyBench 风格的主函数完成构建，最终生成完整程序（第 30–31 行）。此外，每个文件都附带头文件，其中包含其他功能模块的定义，例如数据类型宏和全局参数值。

C语言SCoP提取

LOOPRAG 和 PLuTo 中的 SCoP检测是使用 Clan 实现的。Clan 会在程序中识别 *#pragma scop* 和*#pragma endscop* 指令，并检查所包含的代码区域是否语法有效。

至于 Graphite 和 Polly，它们同样将 SCoP视为优化区域，但其 SCoP检测方法与 Clan 不同。与 Clan 主要进行语法检查不同，Graphite 和 Polly 都会实施更严格的规则，分析循环区域是否完全满足有效 SCoP的要求。具体而言，SCoP中的所有操作（包括函数调用）都必须无副作用——也就是说，只能包含不会修改 SCoP之外任何状态的表达式。

PolyBench 和 LORE 中的代码遵循标准的 SCoP设计，而每个 TSVC内核都会在外层循环中包含一个虚设函数调用。在这种情况下，Graphite 和 Polly 的 SCoP检测会失败，除非明确确保虚设 函数无副作用。

与 Polly 精确的静态控制部分检测不同，Clan 缺乏严格的验证规则，例如指针别名检查或函数副作用分析。相反，它只检查带注释的SCoP区域中的原始代码，并隐含地假设由*#pragma scop* 和*#pragma end-scop* 包围的所有函数都无副作用。因此，Clan 不会进行显式的函数分析（例如 TSVC内核中外层循环里的*dummy* 函数调用），而只是将此类函数视为静态控制部分中的语句。

我们使用 *__属性__((pure))* 限定符标注 *dummy* 函数，以指示 Polly 将该函数视为无副作用，并在分析时忽略其返回值。此标注确保对 *dummy* 的调用被视为无副作用，从而能够成功识别周围的循环结构为有效的静态控制部分（SCoP）。如果没有此限定符，函数调用就会被视为潜在的副作用来源，导致 Graphite 和 Polly 都放弃该 SCoP 候选项。

然而，此限定符对 Graphite 无效：应用*__属性__((pure))* 会导致死代码消除（DCE）移除外层计算循环。因此，为确保公平，我们不将 Graphite 纳入 TSVC 对比。

D 循环特征提取

LOOPRAG 提出了一种考虑循环属性进行检索的循环感知算法。我们选取并抽象循环调度和数组索引属性作为循环特征。在这些属性中，循环调度以二维+1 矩阵形式提取，而数组索引则表示为矩阵，其中包含各个维度中的迭代器和常数系数（图 13）。

循环调度被拆分为常量维度和迭代器维度，以表示该语句在循环嵌套各层级上的偏序，以及其外围循环嵌套中迭代器的顺序。对数组的读操作和写操作分别区分数组索引，以突出它们重要性的差异。每个索引都被拆分为常量列和迭代器列，然后根据依赖进行选择。循环无关依赖仅涉及数组索引中的常量列，而循环携带依赖则同时考虑两者。此外，移除索引矩阵中所有无意义的零列，以便匹配维度数量不同的数组索引。

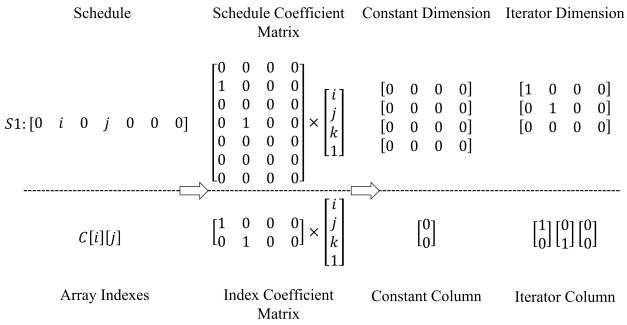

图 13。图 2 中原始代码 *syrk* 的特征提取过程示例。

E 提示词

我们通过以下提示词引导大语言模型应用循环变换优化、从示例中学习，并利用反馈信息。

E.1 基础提示词

作为编译器，请对下面的 *C* 程序使用保持语义的循环变换方法来提升其性能：

{target_code}
以下是一些生成规则：*1.* 提供一份优化代码。*2.* 不要在回复中包含原始 *C* 程序。*3.* 不要定义新函数。*4.* 已有变量无需重新定义。如果生成用于计算的新变量，请使用 *double* 类型。*5.* 请将代码放在 *Markdown* 代码块中。

此提示词用于 §6.2.2 中的基线大语言模型，以生成优化代码。{*target_code*} 表示目标代码。后几行中的规则也会添加到之后的所有提示词中，用于约束生成结果。它们可以大幅减少生成过程中出现的运行时错误，并确保将生成的静态控制部分补全为完整程序以供测试。double 类型的用法与基准测试中原始代码的设置相同。

E.2 带示例演示的提示

// 原始代码 *{ori_code}* // 优化代码 *{opt_code}*
请分析上述示例中使用了哪些保留语义的循环变换方法，并告诉我你学到了什么。请使用从示例中学到的适当方法来提升其性能：*{target_code}*

此提示用于生成步骤 1，作为首次生成优化代码的尝试。{*ori_code*} 和 {*opt_code*} 分别指示例代码及其优化版本。我们特意添加了“*analyze*”和“*learn*”这两个词，以增强大语言模型的优化能力。

E.3 编译结果反馈提示

此优化版本：*{last_code}* 对源代码进行了错误的变换，导致编译错误。编译器错误消息如下：*{error}* 请检查优化后的代码并重新生成。

此提示词用于生成步骤 2 和 4，分别对应第一轮和第二轮编译结果反馈，并附有之前聊天中的消息。{*last_code*}以及{*error*}分别表示上一步生成的优化后的代码，以及编译器在编译过程中提供的错误信息。

E.4 测试结果与性能排名反馈提示

可用示例 *[{idx}]*:
{feedback_*代码}* 失败示例 *[{idx}]*: *{feedback_*代码*}*

... 上述示例由大语言模型使用保义循环变换方法进行了优化。可用示例通过了编译、执行和等价性检查；失败示例则未通过。以下是原始代码：*{target_*代码*}* 性能排名结果（“>”表示优于）：*{idx}* > ... 失败：*{idx}*, ... 任务：分析可用示例为何成功，以及失败示例为何未能保证正确性。利用从排名示例中学到的影响最大的保义循环变换方法，提升原始代码的性能。

生成过程中的步骤 3 会在上述提示中，结合先前对话中的消息，同时利用测试结果和性能排名作为反馈。只有通过测试的代码才会纳入此提示，并同时附上排名结果以提供反馈。{*idx*}、{$feedback_code$}和{*target_code*}分别表示先前优化代码的 ID、先前优化代码的内容以及原始目标代码的内容。在排名中，“...”表示多个先前优化代码及原始目标代码的 ID。如果所有先前优化代码都未通过，提示将退回到提示 E.1。

F LOOPRAG 与基础大语言模型的基准测试加速比分析

为进一步分析和比较 LOOPRAG 与基础大语言模型所取得的性能提升，我们在图 14 中展示了一部分基准测试结果，说明了相较于 GCC 的加速比。在许多情况下，LOOPRAG 的加速比最高超过 256×，显著优于基础大语言模型；后者的加速比不超过 8×。

通过考察不同基准测试套件之间的差异，我们发现，TSVC 中有许多代码的加速比低于 PolyBench 和 LORE 中的代码。然而，与 GCC 相比，表2中 TSVC 的平均加速比指标高于其他基准测试套件。这表明，LOOPRAG 在 TSVC 中生成的若干优化代码实现了极高的加速比，从而扭曲了平均加速比指标。例如，代码

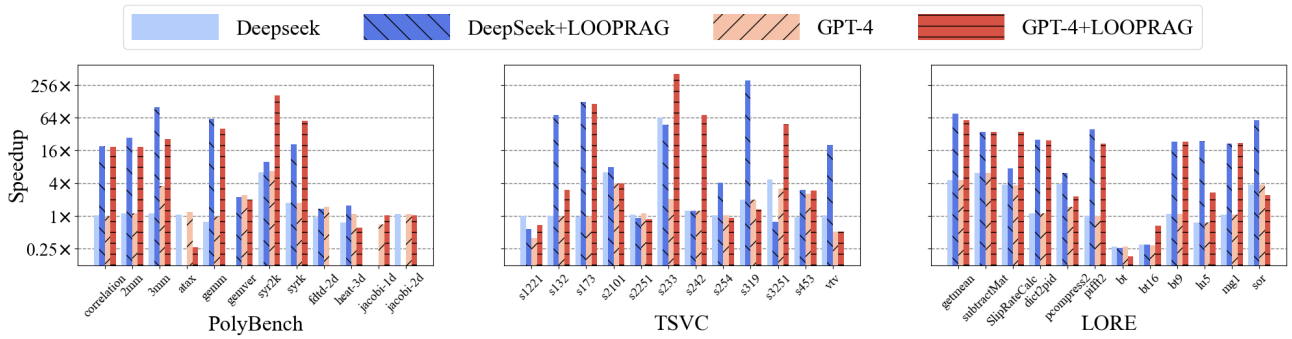

图 14. LOOPRAG 和基础大语言模型取得的加速比。

*s*233 和 *s*319 在 LOOPRAG 使用 DeepSeek 和 GPT-4 时分别达到 412。22× 和 307。62× 。这些优化后的代码利用循环分块和循环交换来提升数据局部性，并通过 OpenMP 指令实现并行性，从而在 TSVC 中取得极高的平均加速比。

然而，如 §6.2.2 所述，LOOPRAG 在 TSVC 上相较于基础大语言模型取得的平均加速比提升，低于其在 PolyBench 和 LORE 上的表现。图7中 TSVC 的更快的代码占比也并不突出。这意味着 TSVC 中的代码相对更容易由大语言模型分析和优化，因此 LOOPRAG 更难胜过基础大语言模型。因此，LOOPRAG 相较于基于大语言模型的方法所实现的加速比提升，主要归功于循环属性复杂的代码；这类代码更能体现循环变换的价值。

G 案例研究 G.1 优化版本案例研究

清单 4. 优化版本示例_*1*。

```
1 # pragma omp p a r a l l e l 2 f o r t 1 = 0到 N/ 3 2 3 f o r t
2 = 0到(t 1+N)/ 3 2 4 f o r t 3 = 32*t 1到 32*t 1+325 f o r
t 4 = 32*t 2到 32*t 2+326 S1 : A[t3 -1][t 3 ] =A[t3 -2][t
3 ] +C[t 3 ][t4 -t 3 ] * 6;7 i f t3 <M & t4 <N 8 S2 : A[t
3 +1][t 3 ] =A[t 4 ][t 3 ] -C[t 3 +1][t 4 ] * 4;
```

清单 5。优化后的版本 示例_*2*。

```
1 # pragma omp p a r a l l e l 2 f o r t 1 = 0to 2*L / 3 2 3 f o
r t 2 =max ( 0 , t1 -L / 3 2 ) t o min ( L / 3 2 , t 1 ) 4 f o r t
3 = 32*t 1 to 32*t 1+325 f o r t 4 = 32*t 2 to 32*t 1+326 i
f t3 -t 4 <t 4 7 S1 : A[t4 ][t3 -t4 ] =A[t4 ][t3 -t 4 ] + 6;8
S2 : A[t4 ][t3 -t4 ] = -A[t3 -t 4 ][t 4 ] +C语言[t3 -t 4 ] - 2;
```

为展示示例中包含的循环变换，我们给出了示例代码的优化版本示例_*1* 和示例_*2*，见清单 4 和 5。这些变换包括 §6.5 中提到的循环分块、循环融合、循环交换、循环倾斜和循环移动。

循环分块会划分并分隔迭代器，以创建诸如 /32 和 $t1$、$t2$、$t3$、$t4$ 这样的特征。循环融合会重新排列语句，使它们共享同一个循环嵌套。循环交换会修改外围循环嵌套的顺序。例如，清单 4 中 *S2* 的外围循环嵌套为$t1$、$t2$、$t3$、$t4$，条件为 $t3 < M \& t4 < N$，对应于“*k*、*i*”的顺序；与清单 2 中“*i*、*k*”的顺序相比，该顺序经过了交换。循环倾斜涉及清单 4 中的 $(t1+N)/32$ 和清单 5 中的 $max(0、t1-L/32)$ 等特征，用于重新分配循环嵌套中的迭代次数。清单 5 中以条件 $t3 - t4 < t4$ 的形式呈现循环移位，用于对齐不同语句的迭代。

这些优化版本中的循环变换为代码*syrk*提供了潜在的循环变换组合示例。循环倾斜和循环移位在这两个优化版本中引入的多样性，可能会被循环分块、循环融合和循环交换的组合所掩盖，因为这些变换在检索到的示例代码的优化版本中更为常见。最后，如图 2 所示，这些示例引导大语言模型生成高效的优化代码*syrk*。

G.2 LOOPRAG 优于基础大语言模型的案例研究

为展示 LOOPRAG 如何优于基础大语言模型，我们在清单 6、7 和 8 中给出了原始代码通用矩阵乘法，以及由基础大语言模型（DeepSeek）和 LOOPRAG 生成的优化代码。代码通用矩阵乘法 在文本特征方面与代码*syrk*（图 2）几乎完全相同。唯一的区别在于 *S2* 中最后一次数组读取的数组索引：代码*syrk*使用数组 $A[j][k]$，而代码通用矩阵乘法 使用数组 $B[k][j]$。

这一细微变化为搜索合适的循环变换组合创造了截然不同的环境。在图 2 中，迭代先遍历迭代器 j，然后是 $k$for $A[j][k]$，这会阻止连续

清单 6. 原始代码 通用矩阵乘法。

```
1 循环i = 0至 NI2 循环j = 0至 NJ3 S1: C语言[i ][j ] *=
beta; 4 循环k = 0至 NK5 循环j = 0至 NJ6 S2: C语言[
i ][j ] +=alpha *A[i ][k ] *B [k ][j ];
```

清单 7. 基础大语言模型生成的通用矩阵乘法 优化代码。

```
1 循 环 i = 0至 NI2 循 环 j = 0至 NJ3 S1: 双精度
temp =C语言[i ][j ] *beta; 4 循 环 k = 0至
NK5 S2: temp +=alpha *A[i ][k ] *B [k ][j ];
6 S3: C[i ][j ] = temp;
```

清单 8. LOOPRAG 生成的通用矩阵乘法 优化代码。

```
1 # pragma omp parallel 2 for t1 = 0 to NI / 32
3 for t2 = 0 to NJ / 32 4 for t3 = 32*t1 to 32*t1 +32
5 for t4 = 32*t2 to 32*t2 +32 6 S1 : C[t3 ][t4 ] *= beta ; 7 for t2 = 0 to NK / 32 8 for t3 = 32*t1 to 32*t
1 +32 9 for t4 = 32*t2 to 32*t2 +32 10 for j = 0 to NJ
11 S2 : C[t3 ][t4 ] += alpha *A[t3 ][j ] *B [t4 ][j ];
```

当数组 A 默认以行优先顺序存储时进行访问。然而，代码 通用矩阵乘法中不存在这个问题，因此数据局部性不同，且相同的循环变换组合不适用于实现最优效率。

如清单 7 所示，DeepSeek 对代码 通用矩阵乘法 应用了循环融合和循环交换，并引入了一个新的临时变量 *temp* ，以减少对 $C[i][j]$ 的访问次数。如图 14 所示，与原始代码相比，这使加速比达到 0.20×。相比之下，LOOPRAG 利用从优化后的示例代码版本中学到的循环分块和 *#pragma omp parallel* 指令，使性能较原始代码提升了 60.61× 。这比 DeepSeek 提高了 303.05×。尽管添加 *#pragma omp parallel* 后，DeepSeek 的加速比可以提升至 3.86× ，但仍显著低于 LOOPRAG 的性能。基础大语言模型 GPT-4 在此目标代码上的表现优于 DeepSeek，但加速比仅为 0.76× ，而 LOOPRAG 达到 44.56×，进一步凸显了 LOOPRAG 的成功。这些结果有力地支持了我们对图 1 的解释，并证明了 LOOPRAG 在引导大语言模型优化代码方面的有效性。

H 讨论

尽管 LOOPRAG 在代码优化方面优于工业编译器、基于大语言模型的方法和 PLuTo，但未来工作仍有很大的改进空间。

LOOPRAG 使用参数驱动方法合成示例代码。然而，现有示例代码仍缺少某些计算特征。例如，代码清单 9 给出了来自 PolyBench 的原始代码*jacobi-2d* ，其为 LOOPRAG 带来的加速比仅为 0.58× 。

代码清单 9.原始代码 *jacobi-2d*。

```
1 对于t = 0到T 2 对于i = 1到N 3 对于j = 1到N 4 S1:
B[i ][j ] =A[i ][j ] +A[i ][j -1] +A[i ][1+j ] +A[1+i ][
j ] +A[i -1][j ]; 5 对于i = 1到N 6 对于j = 1到N 7 S2:
A[i ][j ] =B[i ][j ] +B [i ][j -1] +B [i ][1+j ] +B[1+i
][j ] +B[i -1][j ];
```

清单 10. 优化代码 *jacobi-2d*。

```
1 for t1 = 0到T / 32 2 # pragma omp parallel 3 for t
2 = 0到N / 32 4 for t3 = 32*t2到 32*t2 +32 5 for t4 = 0
到N / 32 6 S1 : B[t3 ][t4 ] =A[t3 ][t4 ] +A[t3 ][t4 -1] +A[
t3 ][1+t4 ] +A[1+t3 ][t4 ] +A[t3 -1][t4 ]; 7 # pragma o
mp parallel 8 for t2 = 0到N / 32 9 for t3 = 32*t2到 32*
t2 +32 10 for t4 = 0到N / 32 11 S2 : A[t3 ][t4 ] =B[t3 ][t
4 ] +B [t3 ][t4 -1] +B [t3 ][1+t4 ] +B[1+t3 ][t4 ] +B[t
3 -1][j ];
```

较低的加速比归因于缺乏恰当的循环变换。在代码清单 9 中，对数组 $S1$和 $S2$的读操作呈现出特定的数组索引模式，涉及 $[i][j]$、$[i+1][j]$、 $[i-1][j]$、$[i][j+1]$和 $[i][j-1]$。这是 *stencil* 计算的典型特征，可通过循环倾斜进行优化，以实现波前并行。然而，LOOPRAG 在其优化代码中仅应用了循环分块。PolyBench 中的*fdtd-2d*、*heat-3d* 和*jacobi-1d* 代码中的 *stencil* 计算也存在类似情况，如图 14 所示。

与另外两个基准测试套件相比，这种计算特征相对复杂且少见，但在现实应用中应用广泛，例如图像处理。在这种情况下，我们需要完善合成过程中的循环属性配置，改进检索过程中的循环特征提取，或引入针对上述计算特征定制的模板，以生成更多样化的示例代码。此外，LOOPRAG 缺乏可靠的语义等价性保证，依赖于

大量测试，而非可证明的正确性。此外，这一过程可能耗时较长；即使将温度设为 0，大语言模型的输出仍存在显著差异。

这种可靠性不足，使 LOOPRAG 无法作为全自动代码优化工具发挥作用。总体而言，我们需要在数据集合成过程中更好地平衡多样性与合法性，在检索过程中平衡多样性与相似性，并在生成过程中采用更高效的方法来验证正确性和性能。