Co通过多阶段学习进行编译器自动调优

MINGXUAN ZHU, 高可信软件技术重点实验室（北京大学），中华人民共和国教育部，中国
郝丹，高可信软件技术重点实验室（北京大学），中华人民共和国教育部，中国JUNJIE C
HEN，智能与计算学部，天津大学，中国

arXiv:2304.14908v1 [cs.PL] 27 Apr 2023

GCC 和 LLVM 等广泛使用的编译器通常包含数百种优化，这些优化由优化标志控制。编译过程中可启用或禁用这些标志，以提升编译器程序的运行时性能（例如，缩短执行时间）。由于优化标志数量众多且组合复杂，编译器用户难以手动调优编译器优化标志。文献中已有多种自动调优技术，它们通过比较编译后的程序在不同优化标志组合下的实际运行时性能，为其调优优化标志。然而，由于搜索空间庞大且实际运行成本高昂，这些技术普遍面临效率问题。为降低实际运行成本，本文提出一种轻量级学习方法，仅使用少量实际运行时性能数据，即可预测编译后的程序在各种优化标志组合下的运行时性能。此外，为缩小搜索空间，我们设计了一种新颖的粒子群算法，利用预测模型调优编译器优化标志。为评估所提出的 CompTuner 的性能，我们使用两种广泛使用的基准测试集 cBench 和 PolyBench，对两种热门 C 编译器 GCC 和 LLVM 进行了广泛的实验研究。实验结果表明，CompTuner 的表现显著优于五种对比技术，其中包括最先进的技术 BOCA。

CCS 概念：• **软件及其工程 → 编译器**。

其他关键词和短语：编译器、编译器自动调优、多阶段学习、粒子群优化

ACM 引用格式：
Mingxuan Zhu、郝丹和 Junjie Chen。2023年。通过多阶段学习进行编译器自动调优。*ACM* 软件工程与方法学汇刊 1，1，第1篇文章（2023年10月），共29页。https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 引言

编译器在软件开发中发挥着重要作用，尤其是在程序执行方面。GCC 或 LLVM 等广泛使用的编译器通常包含数百种优化，例如**函数内联**、**无用代码删除**，这些优化由优化标志控制。在编译过程中，这些优化可以启用或禁用，从而生成运行时性能各异的编译后的程序

作者地址：Mingxuan Zhu，zhumingxuan@stu.pku.edu.cn，高可信软件技术重点实验室（北京大学），中华人民共和国教育部，北京市颐和园路5号，中国，100871；郝丹，高可信软件技术重点实验室（北京大学），中华人民共和国教育部，北京市颐和园路5号，中国，100871，haodan@pku.edu.cn；Junjie Chen，智能与计算学部，天津大学，天津市雅观路135号，300350，中国，junjiechen@tju.edu.cn。

允许为个人或课堂用途免费制作本作品的全部或部分数字副本或纸质副本，前提是这些副本不得用于营利或商业用途，不得为此进行复制或分发，并且副本首页须附上本声明和完整引文。对于本作品中由 ACM 以外的其他方拥有版权的部分，必须尊重其版权。注明出处后，可以摘录。以其他方式复制、重新发布、张贴到服务器或分发至列表，均须事先获得具体许可和/或支付费用。请联系 permissions@acm.org 申请许可。© 2023年 美国计算机协会。1049-331X/2023/10-ART1 $15.00 https://doi.org/10.1145/nnnnnnn.nnnnnnn

[39, 50, 57]。例如，编译器优化标志**-floop-unroll-and-jam** 可以执行外层循环展开和内层循环融合，以优化程序中的循环语句。也就是说，编译器优化标志会显著影响编译后的程序的运行时性能。

另一方面，同一种优化应用于不同程序时，并不总能带来相同的性能提升。尽管 GCC 和 LLVM 等广泛使用的编译器推荐了 -O1、-O2 和 -O3 等优化设置，但这些设置并不能保证对任何程序都能提升运行时性能。

因此，有必要针对不同程序选择**特定的** 编译器优化设置，尤其是那些运行时间较长且优化标志对其影响显著的程序。具体而言，编译器优化设置（即编译器调优）包括选择一组合适的编译器优化（也称为“标志选择” [14, 23] 或“阶段选择” [30]），并确定应用这些优化的顺序（也称为“阶段排序” [5,29]）。本文聚焦于前者，即标志选择问题。鉴于优化数量众多且组合呈指数级增长，编译器用户很难理解每个编译器标志对程序的影响，也难以手动为程序选择编译器优化设置（亦可视为优化序列）。

为减少设置编译器优化所需的人力，文献中提出了多种专注于**标志选择** 的编译器自动调优技术，可分为两类 [4, 6], : 基于监督学习的技术和基于无监督学习的技术。具体而言，基于无监督学习的技术最初是为解决编译器自动调优问题而提出的。对于正在编译的程序，这类技术通常通过某些搜索策略（例如基于爬山法的算法 [2] 和遗传算法 [23, 28, 37, 48]）探索其优化序列搜索空间，并根据采用相应优化序列编译后的程序实际运行时性能来选择优化序列。在搜索过程中，这些技术通常会生成大量优化序列，而目标程序采用相应优化序列编译并执行后，才能收集其运行时性能。也就是说，这些技术非常耗时。为解决成本问题，人们提出了基于监督学习的技术：这类技术首先构建模型来预测优化序列的运行时性能，然后根据预测的运行时性能而非实际性能，采用某些搜索策略寻找理想的优化序列。具体而言，这些技术利用大量数据（即优化序列，以及程序采用相应优化序列编译后的实际运行时性能） [7, 8, 11, 14, 15, 40, 56]构建预测模型，因此仍面临成本问题。此外，由于优化标志的组合空间庞大，要找到理想的优化序列始终颇具挑战，现有编译器自动调优技术（包括基于监督学习和基于无监督学习的技术）仍存在性能问题。例如，我们的实验研究表明，当前最先进的基于无监督学习的技术（即基于遗传算法的技术 [23]）仅在 40 个程序中的 2 个上取得了理想的优化结果；而当前最先进的监督学习技术 BOCA [14][1]，在程序 **telecom_adpcm_c**上达到随机优化的优化后运行时性能，仍几乎需要多出三倍的时间。

[1]BOCA 是 2021年 ICSE 上发表的最新编译器自动调优技术。它通过预测模型识别影响显著的优化标志，以缩小搜索空间，并迭代地选择具有最高 **期望改进** 的优化序列来调优预测模型。

为缓解现有编译器自动调优技术的时间开销和性能问题，本文提出一种基于多阶段学习的编译器自动调优技术**Comp-Tuner**：首先通过多阶段学习构建预测模型，再通过新一轮学习搜索理想的优化序列。具体而言，CompTuner 首先通过多阶段学习构建预测模型，每个阶段仅使用少量经过仔细筛选的数据（即优化序列及其对应的实际运行时性能）。这一轻量级学习过程在不牺牲预测模型准确率的前提下，缓解了现有监督学习技术的成本问题。随后，Comp- Tuner 通过改进的粒子群优化算法 [31, 43, 53], 搜索运行时性能良好的优化序列；该算法在考虑性能多样性的同时，兼顾局部开发与全局探索。所提出的 CompTuner 在预测模型构建和搜索过程两方面均不同于最新的 BOCA 工作。具体而言，BOCA 根据筛选出的最佳性能序列构建预测模型，而 CompTuner 则考虑具有性能多样性的序列来构建预测模型。在搜索过程中，BOCA 通过识别影响显著的标志来缩小搜索空间，而 CompTuner 则采用改进的粒子群算法进行搜索。

为了评估我们方法的性能，我们使用两个广泛应用的编译器 GCC 8.3.0 和 LLVM 9.0.0，针对两个广泛使用的基准测试套件 cBench 和 PolyBench 中的 20 个程序开展了实验。实验结果表明，CompTuner 生成的优化序列始终优于 GCC 和 LLVM 默认的 -O3 优化。此外，在编译器自动调优时间受限的情况下，CompTuner 生成的优化序列显著优于所有五种具有代表性的编译器自动调优技术。此外，我们提出的 CompTuner 在许多情况下都取得了最佳优化结果（即 GCC 上的 8 个程序和 LLVM 上的 8 个程序）。此外，我们手动分析了 BOCA 和 CompTuner 选出的部分优化标志，以及通过搜索过程获得的最佳配置，发现 BOCA 在不同试验中会为同一程序推荐不同的优化标志，而 CompTuner 选出的优化标志与最佳配置中的优化标志则有很大重合。

此外，消融研究表明，CompTuner 构建的预测模型具有较高的准确率（使用 60 个训练数据时，最低预测误差率仅为 1.49%），从而使 CompTuner 获得了良好的加速性能。此外，改进后的粒子群优化算法通过选择具有多样性的优化序列，确实提升了 CompTuner 的性能。在我们选择的 6 个实验案例中，CompTuner 在 4 个案例中取得了更高的加速比，而变体技术[2] 仅在 2 个案例中表现更佳。

本文的贡献总结如下。

- **一种新颖的基于多阶段学习的编译器自动调优方法 CompTuner**，该方法使用少量精选实例构建预测模型，并通过改进的粒子群优化推荐编译器优化序列。
- **一项广泛的实验**，使用 GCC 和 LLVM 对两个编译器优化基准测试 cBench 和 PolyBench 进行评估，结果表明 CompTuner 有效且前景广阔。
- **一个可复现软件包**可在以下位置获取：*https*：*//github.com/TOSEM*202210/*compTuner*

本文其余部分安排如下。第 2 节介绍方法的具体细节。第 3 节介绍实验设置，第 4 节介绍实验结果与分析。第 5 节介绍有效性威胁。第 6 节从三个方面进行讨论。第 7 节简要回顾相关工作。最后，第 8 节总结全文。

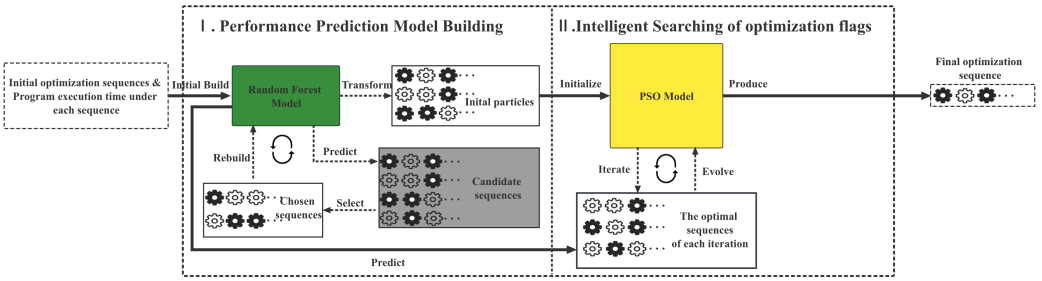


图 1. CompTuner 的工作流程。

2 技术

编译器的作用是将一种编程语言编写的程序转换为另一种编程语言编写的程序，通常会生成可执行程序。为了减小编译代码大小或提升运行时性能，GCC 或 LLVM 等传统编译器通常包含数百个编译器优化标志。例如，GCC 的优化标志 **-floop-interchange**、**-floop-unroll-and-jam** 和 **-fmove-loop-invariants** 用于优化程序中的循环部分，而 **-fcse-follow-jumps**和 **-fguess-branch-probability** 用于优化程序中的分支部分。具体而言，一个优化标志指定了针对某种程序结构的一类程序转换，其取值通常为 0/1[3]（即 0 表示禁用相应的优化标志，1 表示启用相应的优化标志）。

为便于阐述，我们将本文研究的问题形式化如下。给定一个编译器，其优化集合可表示为 $O = \{o_1, o_2, \ldots\ldots, o_n\}$，其中 n 表示优化的数量，o_i 的值（其中 $1 \le i \le n$）表示相应的优化标志是否启用或禁用（即 o_i 设为 1 或 0）。这些优化标志的一种特定设置可用一个 0/1 序列表示，称为优化序列 [14]。整个优化空间的大小（即包含所有优化序列）为 2^n，数值极其庞大。例如，GCC 8.3.0 有 106 项优化，其优化空间大小为 2^{106}。另一方面，启用某个优化标志并不总能改善编译后的程序的运行时行为，而且同一优化标志对不同的编译后程序可能产生不同影响。由于优化空间庞大，且各个优化标志的性能表现各异，手动进行编译器调优是不可能的。因此，本文旨在实现编译器自动调优，即自动为编译后的程序确定特定的优化序列，以提升运行时性能。具体而言，本文关注缩短执行时间这一最常用的编译器优化目标 [14], ，并将缩小可执行代码等其他编译器优化目标留作未来工作。

2.1 方法概述

现有的编译器自动调优技术通常通过比较编译后的程序在不同优化序列下的实际执行时间（即，为 O中的优化标志设置不同的值），来寻找理想的优化序列，但会因搜索空间庞大、执行时间过长而面临性能问题。为解决这一性能问题，在本文中，

[2]也就是说，一种变体技术使用影响显著的标志来缩小搜索空间，而不是使用粒子群优化。[3]少数优化标志的取值并非 0 或 1。例如，**-falign-functions** 可设为 24 或 32，表示函数的对齐位置。与以往所有工作一样，本文重点关注取值仅为 0 或 1 的优化标志。

我们提出了一种基于多阶段学习的方法 CompTuner，通过两个学习阶段缓解这两项性能问题（即执行时间过长和搜索空间庞大）。

图1展示了 CompTuner 的总体框架，该框架由预测模型构建和智能搜索组成。

- 预测模型构建 旨在仅使用少量训练实例，学习预测模型，以预测具有不同优化序列的编译后程序的运行时性能。具体而言，我们提出了一种轻量级学习方法，通过对随机生成的优化序列进行初始学习来构建预测模型，并基于具有多样性的优化序列进行多个阶段的学习，以提高预测准确率。更多详情请参阅第2.2节。

- 智能搜索 旨在通过改进的粒子群优化算法 [31, 43, 53]，为目标程序搜索所需的优化序列。具体而言，我们将编译器自动调优问题转化为粒子群优化问题，其中优化序列被视为一个粒子；随后，基于生成序列的性能相似性，细化粒子群优化算法中的参数，从而扩展搜索空间。更多详情请参阅第2.3节。

2.2 性能预测模型构建

为提高编译器自动调优的效率，所提出的 CompTuner 使用优化序列预测编译后的程序的运行时性能，而不是进行实际执行。

直观而言，预计需要通过大量实例（即，编译后的程序在各种优化序列下的实际运行时性能）才能学习得到高准确率的预测模型，但为如此多的实例进行标注（即，使用各种优化序列执行编译后的程序）无疑非常耗时。为解决这一问题，我们提出了一种轻量级学习方法，通过多个学习阶段构建预测模型：初始学习使用少量随机生成的实例构建预测模型，增强学习则通过多个学习阶段，利用经过仔细选择的实例（具有性能多样性）提高预测模型的准确率。更具体地说，此学习过程中使用的实例是优化序列，其标签是编译后的程序采用相应优化序列时的实际运行时性能。特别地，训练数据的特征是编译器优化标志，每个标志被设置为 0 或 1，分别表示相应的优化标志关闭或开启。请注意，我们不会为所有程序构建统一的预测模型，而是为每个程序构建单独的预测模型[4]。

2.2.1 初始学习。在初始学习阶段，CompTuner 使用少量随机生成的数据构建预测模型。具体而言，CompTuner 首先随机生成少量优化序列，并收集编译后的程序在每个优化序列下的运行时性能（即图1中“初始构建”操作的输入数据），将其作为初始学习的训练数据。请注意，在我们的实验中，初始学习所用的训练数据数量设为 2。随后，CompTuner 使用监督学习算法（本文采用随机森林）基于这些数据构建预测模型，因为随机森林算法通过合并多棵决策树的决策来进行预测，并且在处理具有高维特征的数据时表现良好，而编译器优化序列约包含一百个优化标志。

[4]在评估中，我们为基准测试集中的每个程序构建一个预测模型，并搜索其编译器优化序列。

2.2.2 增强学习。由于初始学习仅使用极少量数据构建预测模型，其预测结果可能不够准确。为解决这一问题，CompTuner 会进一步选择一些运行时性能多样性更高的优化序列，并通过多轮学习进一步提高预测模型的准确率。由于 CompTuner 使用少量训练数据构建预测模型，因此这是一种轻量级学习方法。

在增强学习中，CompTuner 会随机生成一组候选优化序列，并选择运行时性能各不相同的优化序列，用于训练预测模型。对于带有优化序列的目标程序，CompTuner 生成的随机森林模型中的多棵树会对其运行时性能给出不同的预测值，因此，CompTuner 会根据贝叶斯优化 [10, 11, 14, 22, 52, 55]合并不同树得出的预测性能。也就是说，CompTuner 会合并这些预测值的均值和方差，得到该优化序列的整体性能（记为 EI），如公式 1 所示。图1中的“Select”操作根据每个候选序列的EI值执行。

$$EI(sequence) = (f' - \mu(sequence))\Phi(Z) + \sigma(sequence)\phi(Z)$$
$$where\ Z = \frac{f' - \mu(sequence)}{\sigma(sequence)} \tag{1}$$

其中，f' 是用于构建预测模型的优化序列的最短执行时间。对于任意 $sequence$，$\mu(sequence)$ 和 $\sigma(sequence)$ 分别表示随机森林模型中各树预测结果的均值和方差。$\Phi(Z)$ 是累积分布，$\phi(Z)$ 是标准高斯分布在序列性能上的概率密度。

为了提高预测模型的准确率，在每一轮增强学习中，Comp-Tuner 会选择性能各异的优化序列，即兼顾运行时性能的好坏和随机性。基于这一思路，在随机生成的序列中，除了选择 EI 值最大的优化序列外，CompTuner 还会检查该序列的准确率。如果准确率未达到标准（实验中设为 0.95），CompTuner 就会以较高概率选择另一个具有较小 EI 值的优化序列。具体而言，CompTuner 首先按 EI 值降序排列所有生成的序列，然后为每个优化序列分配一个概率，确保 EI 值较小的优化序列具有较高的概率。最后，CompTuner 从所有生成的序列中随机选择一个优化序列，这可能是性能较低的优化序列 [32, 41, 54]。

利用选定的优化序列（包括其实际运行时性能），Comp-Tuner 会训练另一个预测模型，如图1中的“重建”操作所示。Comp-Tuner 会重复这一增强学习过程，直到预测模型足够准确。也就是说，CompTuner 通过初始学习和多个阶段的增强学习，使预测模型达到所需的准确率。在评估中，学习过程的终止标准（即所需的预测准确率）设为 0.96[5]。通过这种构建—调优流程，CompTuner 构建出更准确的预测模型，并为智能搜索过程选择更优的起始数据。

2.2.3 算法详解。该预测模型构建算法的更多细节见算法 1。

[5]对于某些目标程序而言，可能很难达到这一准确率。在这种情况下，为避免使用过多数据训练预测模型，当训练数据超过 50 条时，我们也会停止训练过程。

性能预测模型构建算法的输入参数包括： $IniSize$ 表示初始模型的实例数量， Acc 表示预测准确率阈值， N 表示最终模型的实例数量。第 2–3 行初始化两个向量 $Seqs$ 和 $PerofSeqs$，它们表示用于构建预测模型的实例。第 5–12 行通过随机生成 0-1序列来生成初始性能预测模型的实例。具体而言，第 6 行通过实际执行获取每个序列的性能。第 13 行使用随机森林模型构建预测模型。

第15至39行通过选择新实例来调整预测模型。具体而言，第17至24行选择一个新序列来调整预测模型。第18至19行获取每个候选序列的性能，并选择性能最佳的序列。第18行使用预测模型预测候选序列。第25至35行根据性能多样性选择另一个序列。第25行通过比较所选序列的实际性能与预测性能来获取其准确率。第27至35行检查准确率，并决定是否选择新序列。第28至31行通过提高性能较差序列被选中的概率来选择新序列。第37至38行根据所选序列的平均预测准确率，决定是否停止该算法。第36行利用新实例重新训练预测模型。

2.3 优化标志的智能搜索

为了获得适用于目标程序的理想编译器优化序列，CompTuner 将构建预测模型时所用的数据（即通过图1中的“Transform”操作获得的数据）作为后续搜索过程的起点，并设计了一种粒子群优化算法。为避免局部最优问题，我们进一步通过细化影响探索速度的粒子群优化算法参数来扩展搜索空间，从而得到改进的粒子群优化算法。下文将先介绍如何通过粒子群优化算法解决编译器自动调优问题，再介绍如何改进该算法。

2.3.1 粒子群优化。粒子群优化算法 [31, 43, 53] 是一种具有代表性的搜索算法，它通过模拟鸟类的觅食行为来完成优化过程，并因在平衡局部最优与全局最优方面表现卓越而得到广泛应用。在我们改进的粒子群优化算法中，我们将每个优化序列 O 视为一个**粒子**，其初始运动速度向量由 V_i 表示，初始位置向量（表示每个优化序列的启用/禁用选择）由 X_i表示；其后续运动速度向量则根据公式 2 更新。

$$V_i^{(t+1)} = \omega \times V_i^{(t)} + c_1 \times rand_1^{(t)} \times (pBest_i^{(t)} - X_i^{(t)}) + c_2 \times rand_2^{(t)} \times (gBest^{(t)} - X_i^{(t)}) \tag{2}$$

其中， ω 为惯性权重，用于调节解空间的搜索范围。 c_1， c_2 为加速常数，用于调节探索的步长。 c_1 增大表示局部搜索能力增强，而 c_2 增大表示全局搜索能力增强。 $pBest$ 是每个粒子的历史最优位置向量， $gBest$ 是所有粒子的全局历史位置向量。 $rand_1$ 和 $rand_2$ 是在每次迭代中随机生成的 0 到 1 之间的数值，用于扰动粒子的速度更新。具体而言， $X_{(t)\,i}$是粒子在第 t_{th}次迭代中的位置， $V_{(t)\,i}$是粒子在第 t_{th} 次迭代中的运动速度。借助这些变量，CompTuner 会更新每个粒子在第 $t + 1_{th}$ 次迭代中的运动速度向量，并通过更新后的速度向量更新粒子的位置，如公式 3 和 4 所示。

算法 1性能预测模型构建

过程 构建($IniSize, Acc, N$)
  $Seqs \leftarrow []$
  $PerofSeqs \leftarrow []$
  // 第 5-13 行构建初始性能预测模型
  while $Sizeof(Seqs) < IniSize$ **do**
    $seq \leftarrow random(0,1)^{number_of_flags}$
    如果 seq 不在 $Seqs$ **则**
      $per \leftarrow Performance(seq)$
      $Seqs.append(seq)$
      $PerofSeqs.append(per)$
    结束 if
  结束 while
  $model \leftarrow RandomForestModel(Seqs, \ PerofSeqs)$
  // 行数 15-39 用于调优性能预测模型
  当 $Sizeof(Seqs) < N$**时**
    // 行数 17-24 从候选项中选择期望改进最高的序列
    $candidates \leftarrow Set(random(0,1)^{number_of_flags})$
    $model.predict(candidates)$
    $seq \leftarrow selectFromcandidatesByEI$
    如果 seq 不在 $Seqs$ **则**
      $per \leftarrow Performance(seq)$
      $Seqs.append(seq)$
      $PerofSeqs.append(per)$
    结束条件判断
    $acc \leftarrow Accuracy(seq)$
    // 行数 27-35 根据候选项的性能分布选择序列
    如果 $acc < Acc$ **则**
      $seq_new \leftarrow selectFromcandidatesByValueDistribution$
      当 seq_new 在 $Seqs$ **时**
        $seq_new \leftarrow selectFromcandidatesByValueDistribution$
      结束循环
      $per \leftarrow Performance(seq_new)$
      $Seqs.append(seq_new)$
      $PerofSeqs.append(per)$
    结束 if
    $model \leftarrow RandomForestModel(Seqs, \ PerofSeqs)$
    如果 $AverageAcc(Seqs) > Acc$ **则** 跳出
    结束 if
  结束 while
结束过程

$$s_{i,j}^{(t)} = \frac{1}{1 + e^{-v_(i,j)^{(t)}}} \tag{3}$$

$$x_{i,j}^{(t)} = \begin{cases} 1, rand() < s_{i,j}^{(t)} \\ 0, rand() \geq s_{i,j}^{(t)} \end{cases} \tag{4}$$

由于粒子的 position vector 由 0 和 1 组成，CompTuner 引入临时向量 S_i 来调整每个粒子的 position vector。在第 t_{th} 次迭代中，通过 Sigmoid函数将更新后的运动速度向量 $S_i^{(t)}$ 转换为临时向量的取值，如公式 3 所示，从而使临时向量的取值范围变为 $[0, 1]$。其中，每个粒子的临时向量取值为 $S\ (t)\ i$。

CompTuner 通过临时向量，按照公式 4 更新每个粒子的 position vector。对于粒子 i，如果临时向量 S_i 的第 j_{th} 个元素 $s_{i,\ j}$ 大于一个随机值（其取值范围为 $[0, 1]$），则 position vector X_i 的第 j_{th} 个元素 $x_{i,\ j}$ 更新为 1；否则更新为 0。这样，CompTuner 就完成了本次迭代中每个粒子的速度向量和 position vector 的更新，如图 1 中的“演化”操作所示。

在每次迭代中，CompTuner 使用预测模型（在第2.2节中生成）预测生成的优化序列（即粒子的位置向量）的运行时性能（本文中指执行时间），而不是进行实际执行，以减少编译器自动调优所耗费的时间，如图1中的“预测”操作所示。具体而言，在每次迭代中，CompTuner 根据优化序列（即每个粒子的位置向量）的预测运行时性能，更新每个粒子的最优位置 $gBest_i$ 以及所有粒子的最优位置 $pBest$。

2.3.2 粒子群优化的改进。为避免陷入局部优化，CompTuner 进一步通过不同方式利用性能良好和性能较差的序列，扩展优化序列的搜索空间。具体而言，CompTuner 根据预测性能，从每次迭代生成的所有序列中确定最优序列，并根据各序列的性能是否与最优序列相近，将这些序列分为两组（即性能良好的序列组和性能较差的序列组）。此外，CompTuner 根据序列相似度判断某个优化序列的性能是否与最优序列相近，因为相似优化序列的加速效果也具有一定的接近性。具体而言，CompTuner 使用公式5（即两个向量 X_i和 X_j [61]的余弦相似度）计算某个序列（记为 X_i）与最优序列（记为 X_j）之间的相似度，并用 $x_{i、s}$和 $x_{j、s}$表示它们的编译器优化标志（0或1）。

$$sim(X_i, X_j) = \frac{\sum_{s=1}^{n}(x_{i,s} \times x_{j,s})}{\sqrt{\sum_{s=1}^{n}(x_{i,s})^2} \times \sqrt{\sum_{t=1}^{n}(x_{j,s})^2}} \tag{5}$$

基于该公式，CompTuner 将生成的序列分为两组：与最优序列相似度较高的序列（即性能较好的序列集合，记为 $part_1$），以及与最优序列相似度较低的序列（即性能较差的序列集合，记为 $part_2$）。此外，如果本次迭代中生成的最优序列的运行时性能优于已有最优序列的运行时性能，CompTuner 会增大 c_1 的值，以增强 $part_1$ 中粒子的局部搜索能力，尝试从当前最优值中找到更好的结果；同时增大 c_2的值，以增强 $part_2$ 中粒子的全局搜索能力，尝试在更大的搜索空间中获得理想结果。

算法 2智能搜索

```
1: 过程 搜索(IniPars, seq_initial, s_initial)
2:     bestPerofEach ← PerofEachParofIniPars
3:     bestSeqofEach ← SeqofEachParofIniPars
4:     bestPerofAll ← s_initial
5:     bestSeqofAll ← seq_initial
6:     velofEach ← RandomlyGenerated
7:                                  // 第 8-9 行更新粒子以生成新序列
8:     对于 i ← 1 to M 执行
9:         如果 i = 1则
10:            NewPars ← Evolution(IniPars, velofEach)
11:                 // 第 13-22 行预测粒子的位置向量，并更新每个粒子
12:        else
13:            prePer ← []
14:            对于 j ← 1 to sizeof(IniPars) 执行
15:                prePer[j] ← model.predict(NewPars[j])
16:                如果 prePer[j] ≤ bestPerofEach[j] 则
17:                    bestPerofEach[j] ← prePer[j]
18:                    bestSeqofEach[j] ← NewPars[j]
19:                else
20:                    继续
21:                结束 if
22:            结束循环
23:                                 // 行数 23-33 更新粒子以生成新的序列
24:            CurrentBestPer ← best(bestPerofAll)
25:            如果 CurrentBestPer > bestPerfofAll那么
26:                bestPerofAll ← CurrentBestPer
27:                bestSeqofAll ← best(bestSeqofEach)
28:                NewPars ← Evolution(NewPars)
29:            else
30:                NewPars_1, NewPars_2 ← divide(NewPars)
31:                NewPars_1 ← Evolution(NewPars_1)
32:                NewPars_2 ← Evolution(NewPars_2)
33:                NewPars ← Combine(NewPars_1, NewPars_2)
34:            结束 if
35:        结束 if
36:    结束 for
37:    (seq_final, s_final) ← (bestSeqofAll, bestPerofAll)
38:    返回 (seq_final, s_final)
39: 结束过程
```

2.3.3 算法详情。该改进型粒子群优化算法的更多细节见算法 2。

第 2-6 行为搜索过程初始化粒子。第 2-5 行通过 $InitialPars$（即用于构建预测模型的数据集）初始化每个粒子的最优位置向量（即序列）和性能（即目标程序的执行时间），并初始化所有粒子的最优位置向量和性能：$seq_{initial}$（即性能最佳的序列

（用于构建预测模型））和 $s_{initial}$（即构建预测模型时的最佳性能）。第 6 行为每个粒子随机生成一个速度向量。

第 7–37 行对整个粒子群进行迭代更新，其中每个粒子的位置向量在搜索过程中（即生成新序列时）不断变化，而新序列则由生成的预测模型进行预测。CompTuner 持续选取性能最佳的序列，直至达到终止条件。第 9–10 行根据公式 3 和 4，使用随机生成的速度向量，在进化的第一轮中更新每个粒子的位置向量。在后续迭代中，第 12–22 行更新每个粒子的最佳位置和性能。第 15 行预测上一轮迭代中更新的每个粒子的新位置向量，随后第 16–21 行更新每个粒子的最优位置向量（即编译器优化序列）和最优性能。

第 24–35 行在新一轮进化中再次更新每个粒子的位置向量。为避免搜索过程陷入局部最优，CompTuner 提供了两种更新方式。第 24 行首先获取本轮所有粒子的最优性能。如果该性能优于此前所有粒子的最优性能，则第 25–28 行根据公式 2、3 和 4，更新所有粒子的最优性能和最优位置，以及粒子的位置向量。反之，第 29–33 行根据本轮的最优序列，按照公式 5 将所有粒子分为两组，并以不同方式更新两组粒子（即调整公式 2 中 c_1 和 c_2 的值），最后合并两组中的粒子。第 34–35 行获取所需的编译器序列，以及编译后的程序在该优化序列下的执行时间。

3 实验设置

为评估 CompTuner 的性能，本实验旨在回答两个研究问题。

- **RQ1：有效性**。 与最先进的编译器自动调优技术相比，CompTuner 的表现如何?
- **RQ2：消融分析**。 CompTuner 的各个组件分别如何影响性能?

3.1 数据集

在本实验中，我们选择了两种广泛使用的 C 编译器 GCC 8.3.0 [24] 和 LLVM 9.0.0 [33] 作为目标编译器，它们已被众多研究采用[18–20, 46][6]。与最先进的研究工作一样 [14], ，我们使用来自两个广泛使用的基准测试套件 cBench [13] 和 PolyBench [44]的相同程序作为目标程序。

表1展示了这些编译后的程序的统计数据。前十个程序来自 cBench，其余程序来自 PolyBench。也就是说，在本实验中，我们有 20 个编译后的程序，涵盖了广泛的实际功能，代码行数从 200 行到 26,000 行不等。

[6]为验证 CompTuner 在最新编译器上的有效性，我们还使用 GCC 12.2.0 和 LLVM 15.0.0 进行了实验，并在第 6 节中讨论了实验结果。

表1. 程序统计信息

ID	程序	功能	行数
P1	相关性	计算相关性	248
P2	协方差	计算协方差	218
P3	symm	对称矩阵乘法	231
P4	2mm	两次矩阵乘法	252
P5	3mm	三次矩阵乘法	267
P6	Cholesky 分解	Cholesky 分解	212
P7	lu	LU分解	210
P8	Nussinov算法	预测 RNA折叠	569
P9	三维热方程	三维空间中的热方程	211
P10	二维Jacobi算法	Jacobi式模板计算	200
C1	汽车位计数_	位操作测试	954
C2	汽车susan e_ _	边缘图像识别	2,129
C3	汽车 SUSAN C_ _	角点图像识别	2,129
C4	汽车 SUSAN S_ _	图像平滑	2,129
C5	consumer_tiff2rgba	TIFF 图像转换	22,321
C6	消费者_jpeg_c	图像压缩	26,950
C7	office_rsynth	文本转语音合成	5,412
C8	security_sha	安全哈希算法	297
C9	bzip2e	文件压缩	7,200
C10	电信_自适应差分脉冲编码调制_c	脉冲编码调制	389

3.2 对比技术

现有的编译器自动调优技术可分为基于无监督学习的技术和基于监督学习的技术 [4, 6]。

基于无监督学习的技术通过特定的搜索策略，在优化空间中实际执行目标程序，从而选择性能良好的编译器优化序列。我们考虑两种具有代表性的基于无监督学习的技术：随机迭代优化 [15]（简称**RIO**，这是一种简单的基于无监督学习的技术）和基于遗传算法的技术 [23]（简称**遗传算法**，这是最先进的基于无监督学习的技术）。此外，本研究还考虑了**OpenTuner**[7],，这是 GitHub 上一个通用的调优框架，并非专为编译器自动调优设计。

基于监督学习的技术从一些已有的优化序列中学习性能知识，然后通过某些方法生成新的编译器优化序列并对其进行预测，最后选出理想的编译优化序列。我们考虑两种具有代表性的基于监督学习的技术：**BOCA** [14]（这是最先进的编译器自动调优技术）和**TPE** [9]（这实际上是一种通用但先进的基于贝叶斯优化的技术，并非专为编译器自动调优设计）。

为减轻所提出的技术和对比技术产生的随机性的影响，我们对每种技术重复执行五次。

[7]https://github.com/jansel/opentuner

3.3 实现

我们基于 NumPy [38] 和 scikit-learn [51]，使用 Python 实现了 CompTuner。我们采用 scikit-learn 中随机森林模型的实现，并使用默认参数设置；同时根据一些试验为所提出的粒子群优化算法设置参数值。具体而言，参数 c_1、 c_2和 w 分别设置为 2、2 和 0.6。

在实现对比技术时，如果它们提供了可复现的软件包（即 TPE、BOCA 和 OpenTuner 技术），我们就直接使用其代码；否则，我们会严格按照论文中的描述重新实现这些技术（即 GA 和 RIO）。

实验在一台工作站上进行，该工作站配备 Intel(R) Xeon(R) Gold 5218R CPU @ 2.10GHz、377.8G 内存，运行 Ubuntu 16.04.6 LTS 操作系统。

3.4 测量

与以往工作一样 [14], ，在本实验中，我们通过与编译器的默认优化进行比较，来衡量编译器自动调优技术在运行时性能方面的提升。具体而言，GCC 和 LLVM 提供 -O1/O2/O3/Ofast 优化设置；在本研究中，我们使用它们的 -O3 优化设置作为编译器的默认优化，因为这是这些编译器中级别最高的默认优化设置，并且被现有的编译器调优工作广泛采用 [14]。

为了获得编译器自动调优技术带来的运行时性能提升，对于任意 C 程序 p，我们分别使用编译器自动调优技术选出的优化序列和默认编译器优化选项 -O3 进行编译 p，得到两个编译后的程序 p_r和 p_d。随后运行这两个程序 p_r和 p_d，并通过用 p_r 的执行时间除以 p_d 的执行时间，计算 p （由编译器自动调优技术实现的）运行时性能加速**加速比**。我们将所得结果用作评估编译器自动调优技术的指标。我们使用 Linux 的**time**命令获取 p_r 和 p_d 的执行时间。

4 结果与分析

4.1 总体有效性

4.1.1 不同技术的结果。为了研究我们提出的 CompTuner 对目标程序的优化效果，我们将 CompTuner 与五种最先进的编译器自动调优技术（即 RIO [15], 、遗传算法 [23], 、TPE [9] 、BOCA [14] 和 OpenTuner [3]）进行比较。表2和表3展示了比较结果。

最先进的工作 BOCA [14] 使用迭代次数作为终止条件（即 60 次迭代），但迭代次数与实际中的调优选择并不相符。为模拟编译器自动调优的实际应用，我们将 CompTuner 的时间限制设为 6,000 秒，并让 CompTuner 对每个目标程序在 6,000 秒内运行，记录：(1) 生成的优化序列所实现的运行时性能加速比（记为 $performance_c$），以及 (2) 生成该优化序列所用的时间（记为 $time_c$）。相应结果以表格中 CompTuner 的**加速比**值给出，其中括号前的值表示 $performance_c$，括号内的值表示 $time_c$。请注意， $time_c$ 不超过给定的 6,000 秒。为获取 $time_c$，我们使用 Python 库的 **time()** 功能记录 CompTuner 消耗的时间。如果耗时达到 6,000 秒，我们就停止调优过程。我们记录此过程中获得的最佳加速结果，以及 $time_c$达到该结果所用的时间。对于对比技术，我们在 $1.5 * time_c$ 的时间限制下重复调优过程。随后，我们根据对比技术能否生成具有相同运行时

表 2. GCC 上各程序相较于 -O3 的性能提升

技术编号		加速比	ID	加速比	ID	加速比	ID	加速比	ID	加速比
CompTuner		**1.077(3107)**		**1.080(4067)**		1.042(2573)		1.071(3720)		**1.041(2976)**
RIO		✖		✖		1.042(4172)		✖		✖
GA	P1	✖	P2	✖	P3	✖	P4	✖	P5	1.041(3160)
TPE		✖		✖		1.046(3775)		**1.072(3112)**		✖
BOCA		✖		✖		**1.075(1923)**		1.071(3726)		1.046(3639)
OpenTuner		✖		✖		✖		1.075(4691)		✖
CompTuner		1.013(4726)		**1.073(5549)**		1.029(3661)		1.025(2976)		**1.055(2192)**
RIO		**1.016(3018)**		✖		**1.029(3264)**		✖		✖
GA	P6	1.013(3862)	P7	✖	P8	✖	P9	1.025(3684)	P10	✖
TPE		✖		✖		✖		**1.027(2637)**		✖
BOCA		1.014(4971)		✖		1.030(4082)		1.028(3420)		1.055(3026)
OpenTuner		✖		1.075(6792)		1.033(4970)		✖		✖
CompTuner		1.382(4642)		1.424(4905)		**1.327(3907)**		1.118(3018)		1.036(5527)
RIO		✖		✖		✖		1.119(3976)		✖
GA	C1	✖	C2	✖	C3	✖	C4	✖	C5	**1.038(5028)**
TPE		✖		✖		1.327(4527)		**1.142(2998)**		✖
BOCA		**1.397(3952)**		**1.498(4850)**		1.327(4382)		1.127(3192)		✖
OpenTuner		✖		1.431(5782)		✖		1.224(4102)		✖
CompTuner		1.018(4869)		**1.451(5019)**		1.038(4892)		**1.020(3628)**		1.010(2324)
RIO		✖		✖		✖		✖		1.011(2971)
GA	C6	**1.018(3960)**	C7	✖	C8	✖	C9	✖	C10	✖
TPE		✖		1.452(5927)		✖		1.023(3785)		1.014(2672)
BOCA		1.025(4027)		✖		**1.040(4285)**		✖		**1.017(1852)**
OpenTuner		✖		✖		1.039(5562)		1.026(5179)		✖

性能加速比 $performance_c$，探究其性能。具体而言，我们在更长的时间限制内运行每种对比技术，并记录 (1) 生成的优化序列中是否有任何一个能够达到运行时性能加速比 $performance_c$，以及 (2) 生成该优化序列所用的时间。相应结果以表格中 RIO、GA、TPE、BOCA 和 OpenTuner 的**加速比** 值给出，其中 ✖ 表示对比技术未能生成具有如此高运行时性能加速比的优化序列，括号内的值表示生成相应序列所用的时间。如果对比技术生成了具有如此高运行时性能加速比的优化序列，我们会列出其运行时性能加速比和所用时间。为减轻随机性的影响，每种技术都重复运行五次，因此表格中列出的是中位数结果。完整结果见可复现软件包。此外，表格中的时间越短，表示相应的自动调优技术越有效。请注意，对于每个目标程序，我们都用粗体强调所有受研究自动调优技术中的最佳结果。

根据表2和表3，CompTuner 在 GCC 上的运行时性能加速范围为 1.010 至 1.451，在 LLVM 上的范围为 1.013 至 1.361，这表明：(1) CompTuner 生成的编译器优化序列始终优于 GCC 和 LLVM 提供的默认 -O3 优化；(2) CompTuner 始终通过生成的

表 3. LLVM 上程序相较于 -O3 的性能提升

技术编号		加速比	ID	加速比	ID	加速比	ID	加速比	ID	加速比
CompTuner		**1.042(2052)**		1.025(2439)		1.041(2182)		**1.078(3085)**		**1.066(2994)**
RIO		✖		1.027(2271)		✖		✖		✖
GA	P1	✖	P2	✖	P3	1.041(2267)	P4	✖	P5	✖
TPE		✖		✖		1.063(2379)		1.079(4673)		✖
BOCA		1.042(2892)		1.030(2189)		**1.043(2074)**		1.078(3562)		✖
OpenTuner		✖		**1.025(2174)**		1.043(2991)		✖		1.066(4157)
CompTuner		1.013(5963)		1.023(5728)		**1.116(3625)**		1.014(2230)		1.036(2874)
RIO		1.015(6124)		**1.040(4419)**		✖		✖		✖
GA	P6	✖	P7	✖	P8	✖	P9	✖	P10	1.036(3627)
TPE		✖		✖		✖		✖		1.038(3135)
BOCA		**1.015(5732)**		✖		1.124(3824)		**1.015(2152)**		**1.039(2610)**
OpenTuner		✖		1.027(4993)		✖		1.014(2962)		✖
CompTuner		**1.030(3327)**		1.069(4126)		**1.204(3294)**		1.099(2965)		1.019(5524)
RIO		✖		✖		✖		✖		✖
GA	C1	✖	C2	1.069(4362)	C3	✖	C4	1.099(3642)	C5	**1.019(3057)**
TPE		✖		✖		1.204(3894)		✖		✖
BOCA		✖		**1.072(3920)**		1.209(3362)		✖		✖
OpenTuner		✖		✖		✖		**1.108(2711)**		1.038（4717）
CompTuner		**1.093(4170)**		1.025(2749)		1.020(3998)		**1.036(3557)**		1.361(2758)
RIO		✖		✖		1.023(4872)		1.038(3913)		1.371(3724)
GA	C6	✖	C7	✖	C8	✖	C9	1.044(3721)	C10	✖
TPE		1.101(4535)		1.025(3889)		✖		1.038(3659)		**1.370(2159)**
BOCA		1.093(5598)		**1.029(2597)**		**1.032（2892）**		✖		1.372(2352)
OpenTuner		✖		1.028(3917)		1.022(4537)		✖		✖

编译器优化序列加速目标程序，但对不同目标程序的加速效果各不相同。此外，CompTuner 生成具有良好运行时性能的优化序列所用的时间（即括号内的数值 $time_c$）通常少于给定的 6,000 秒时间限制，这表明 CompTuner 未来仍有改进空间。

许多对比基线（即 RIO、遗传算法、TPE、BOCA 和 OpenTuner）的结果都标记为 ✖，这表示即使给予更长的时间限制，这些对比基线生成的优化序列也无法达到与 CompTuner 相同的运行时性能加速。此外，在 GCC 上的 2 个程序（即 P1、P2）和 LLVM 上的 1 个程序（即 C1 ）中，即使给予更长的时间限制，所有基线都无法生成具有与 CompTuner 相同运行时性能加速的优化序列。在五种基线中，BOCA 的表现要好得多，因为 RIO、遗传算法、TPE、BOCA 和 OpenTuner 分别有 29、29、24、12 和 24
✖ 分别依据此表。对于一种比较技术，更多的 ✖表示该技术即使在更多案例上获得更多时间，也无法达到 CompTuner 所取得的优化结果。

4.1.2 统计分析。为进一步考察 CompTuner 是否优于作为对比的基线方法，我们从两个方面对实验结果进行统计分析。首先，我们探究

最佳技术，分析某项技术在多少个案例中取得了最佳优化结果。其次，我们对 CompTuner 和当前最先进的技术 BOCA 进行一系列统计分析（包括显著性分析）。

所有技术之间的比较。根据表2和表3，我们针对每个程序确定取得最佳优化结果的编译器自动调优技术（以粗体标示）。然后，对于每种编译器自动调优技术，我们统计其取得最佳优化结果的程序数量，结果列于表4。该表最后三列给出了 GCC、LLVM 以及两种编译器的统计结果。

从该表可以看出，在20个目标程序中，我们提出的 CompTuner 分别在 GCC/LLVM 的 8/8 个程序上取得了最佳优化结果；相比之下，RIO、遗传算法、TPE、BOCA 和 OpenTuner 分别仅在 GCC/LLVM 的 2/1、2/1、3/1、5/7 和 0/2 个程序上取得最佳结果。这一观察证实，在大多数情况下，CompTuner 在生成性能良好的优化序列方面优于各项基线方法。

总而言之，我们所提出的技术 CompTuner 生成的优化序列优于 GCC 和 LLVM 默认的 -O3 优化。此外，在大多数情况下，它的表现优于所比较的编译器自动调优技术，包括最新的 BOCA，表明 CompTuner 前景可期。

表 4. 最佳性能结果数量

技术	GCC	LLVM	All
CompTuner	**8**	**8**	**16**
RIO	2	1	3
GA	2	1	3
TPE	3	1	4
BOCA	5	7	12
OpenTuner	0	2	2

CompTuner 与 BOCA 的比较。由于 BOCA 是最新的编译器自动调优工作，我们对 CompTuner 和 BOCA 进行了全面比较。

从表2和表3可以看出，在20个目标程序中，对于 GCC/LLVM 编译器，我们提出的 CompTuner 分别在14/11个程序上取得了优于 BOCA 的结果，而 BOCA 对于 GCC/LLVM 编译器分别仅在6/9个程序上取得了更好的结果。即使 CompTuner 和 BOCA 都未取得最佳结果，CompTuner 通常也优于 BOCA（例如 GCC 上的 P6）。也就是说，CompTuner 在大多数程序上都优于 BOCA。我们进一步研究了 CompTuner 优于 BOCA 的25个程序（GCC 上的14个程序和 LLVM 上的11个程序），比较 CompTuner 和 BOCA 在取得相同加速比时所用的时间[8]。平均而言，CompTuner 在 GCC 的14个程序上节省了20.56%的时间，在 LLVM 的11个程序上节省了24.87%的时间。也就是说，在这些程序上，CompTuner 以远少于 BOCA 的时间取得了相同的优化结果，这也表明 CompTuner 的性能优于 BOCA。

此外，我们对表 2 和 3[9] 中的 40 个案例进行显著性分析，以比较 CompTuner 和 BOCA 的加速效果。通过对这 40 个

[8]如果 BOCA 在规定时间内未达到相同的优化结果，我们将时间设为 6000 秒。[9]与之前相同，如果 BOCA 未能在规定时间内达到 CompTuner 的优化结果，则将其时间设为 6000 秒。

案例的耗时执行配对 t 检验，我们得到 p值 0.002986，这表明 CompTuner 生成的优化标志设置显著优于 BOCA 生成的设置。

我们手动考察了 CompTuner 和 BOCA 推荐的一些程序的优化标志设置，以了解前者为何优于后者。具体而言，对于每个程序，我们分别运行 CompTuner 和 BOCA 五次，手动分析它们推荐的优化标志设置。我们发现，CompTuner 五次推荐的优化标志设置大体上相近，而 BOCA 五次推荐的设置则截然不同。例如，对于 GCC 上的程序 P7，CompTuner 在所有五次实验中始终启用一组包含 30 个优化标志的设置，而 BOCA 始终只启用一组包含 9 个优化标志的设置。换言之，BOCA 在五次实验中为 P7 推荐的优化序列差异很大。也就是说，BOCA 表现不如 CompTuner，因为它的预测模型未能成功推荐影响显著的标志。

4.1.3 CompTuner 的优化序列。为了进一步考察 CompTuner 是否为程序选择了合适的编译器优化标志，我们对 CompTuner 生成的优化序列进行了一项小规模研究。

P1 和 C1 的优化序列。

这里以程序 P1 和 C1 为例，手动分析 CompTuner[10] 生成的优化序列。

对于程序 P1，CompTuner 选择了 81 个优化标志，包括 7 个循环相关优化标志、6 个分支相关优化标志，以及针对其他程序结构的 68 个标志。具体而言，我们发现 CompTuner 选择了许多循环相关标志（例如 -floop-unroll-and-jam、-floop-interchange）。这一结果符合我们的预期，因为程序 P1 由五个双重循环结构和一个三重循环结构组成，这些结构占据了程序的大部分内容。在其余未选择的标志中，CompTuner 没有选择 -fcrossjumping 标志，因为官方 GCC文档将该标志解释为“生成的代码性能可能优于启用交叉跳转之前的代码，也可能不如之前”。

对于程序 C1，CompTuner 选择了 85 个优化标志，包括 3 个循环相关标志、9 个分支相关标志，以及针对其他程序结构的 73 个标志。与 P1 相比，C1 包含更多分支结构，因此 CompTuner 为 C1 选择了大多数分支相关标志（例如 -fcse-follow-jumps、-fguess-branch-probability）。此外，由于 C1 包含许多计算，CompTuner 也选择了相应的标志（例如 -fipa-icf、-ftree-bit-ccp）。另外，对于 C1，CompTuner 没有选择 -fgcse 标志，因为官方 GCC文档将该标志解释为“添加 -fno-gcse 后，运行时性能可能会更好”。

优化序列比较。

此外，通过与最佳优化序列（即最佳优化配置）进行比较，了解 CompTuner 生成的优化序列效果如何也很有意义，因此我们在本节开展了一项小规模研究。由于 GCC 编译器包含 100 多个优化标志[11]，我们很难针对全部优化标志进行研究。作为替代，我们选取 PolyBench 和 cBench 的前三个程序，即程序 P1、P2、P3、C1、C2、C3，并构建包含较少优化标志的编译器。具体来说，对于每个受研究的编译器（即 GCC 和 LLVM），我们构建两组编译器优化标志，每组包含 10 个随机选取的编译器优化标志。表5展示了这四组编译器优化标志。对于每组包含 10 个优化标志的集合，我们通过搜索过程找出其最佳

[10]我们在网站上公布了 CompTuner 为所有程序选择的序列。[11]搜索空间比 2^{100}, 更大，因此即使为一个程序找到最佳优化配置，成本也非常高。

配置，并将最佳配置与 CompTuner 生成的优化序列进行比较。

表5. 配备选定优化标志的编译器

编译器	标志
GCC1	-fssa-phiopt, -fsched-interblock, -ftree-循环-distribution, -ftree-loop-vectorize, -fcse-follow-jumps, 跳转线程优化, 内联小函数, -fipa-cp, 树级死代码消除, 树级死存储消除
GCC2	-fdelayed-branch, -fdse, -fipa-reference, -ftree-sink, fipa-vrp, -fipa-bit-cp, -floop-unroll-and-jam、-floop-interchange、-fpeel-loops、-fgcse
LLVM1	-targetlibinfo, 基于类型的别名分析, -basicaa, -globalopt, -ipsccp, -无用参数消除, -instcombine, -基础调用图, -异常处理裁剪, -inline
LLVM2	-functionattrs, -argpromotion, -domtree, -早期公共子表达式消除, -lazy-value-info, -jump-threading, -correlated-propagation, -simplifycfg, -instcombine, -tailcallelim

由于篇幅有限，论文中没有展示 CompTuner 选出的具体优化序列和最佳配置，而是将其发布在网站上。作为替代，我们分析了它们所选优化序列之间的差异，以及运行时性能加速方面的差异。分析结果见表6：第二列展示使用 CompTuner 选出的优化序列编译的程序，其运行时性能与最佳配置的运行时性能之比；第三列展示 CompTuner 选中但最佳配置未选中的标志；第四列展示最佳配置选中但 CompTuner 未选中的标志；最后一列展示 CompTuner 与最佳配置中设置相同的优化标志所占比例。

从第二列可以看出，CompTuner 的性能通常接近 1，这表明 CompTuner 选出的优化标志所实现的运行时性能加速与最佳配置相当。从最后一列可以看出，准确率始终高于 0.7，且大多高于 0.8，这表明 CompTuner 推荐的优化配置与最佳配置非常接近。第三列和第四列也能得出这一结论：CompTuner 最多只会选中 2 个不必要的标志，或最多漏掉 2 个标志。

此外，我们还观察到，CompTuner 的运行时性能加速（即第二列所示）通常会随着其准确率（即最后一列所示）的提高而增加。这一现象符合预期，因为准确率较高的优化序列往往与最佳配置有很大重合。此外，在 GCC2 上比较 P1 和 C3 时也有一些例外，我们猜测原因是某些优化标志（例如 -ftree-sink）对运行时性能的影响可能远大于其他标志。

4.2 消融分析

在本节中，我们进行消融分析，以研究 CompTuner 的两个组成部分各自的贡献：预测模型（见第2.2节）和智能搜索过程（见第2.3节）。

表6。 CompTuner 与最佳配置的比较 方案

程序	性能	不必要的标志	缺失的标志	准确率
		GCC1		
P1	0.953	-fsched-interblock	跳转线程优化	0.8
P2	0.936	-fcse-follow-jumps	内联小函数, 树级死存储消除	0.7
P3	0.964	-	-fsched-interblock, -ftree-loop-vectorize	0.8
C1	0.981	-树级死代码消除	-	0.9
C2	0.905	-fssa-phiopt	内联小函数, -fipa-cp	0.7
C3	0.947	-ftree-dse	-ftree-loop-distribution	0.8
		GCC2		
P1	0.966	-	-ftree-sink	0.9
P2	0.949	-fdse	-floop-interchange	0.8
P3	0.922	-ftree-sink	-fipa-reference, -fgcse	0.7
C1	0.913	-fipa-bit-cp, fipa-vrp	-fgcse	0.7
C2	0.935	-fipa-bit-cp	-fpeel-loops	0.8
C3	0.971	-fdse, -floop-unroll-and-jam	-	0.8
		LLVM1		
P1	1.000	-	-	1.0
P2	0.976	-基于类型的别名分析	-	0.9
P3	0.947	-basicaa	-ipsccp	0.8
C1	0.980	-	-ipsccp	0.9
C2	0.977	-	-基础调用图, -异常处理裁剪	0.8
C3	0.935	-无用参数消除	-tbaa	0.8
		LLVM2		
P1	0.984	-早期公共子表达式消除	-	0.9
P2	0.962	-domtree	-jump-threading	0.8
P3	0.973	-instcombine,	-lazy-value-info	0.8
C1	0.991	-	-functionattrs	0.9
C2	0.946	-domtree	-jump-threading, -simplifycfg	0.7
C3	0.959	-argpromotion	-correlated-propagation	0.8

4.2.1 预测模型分析。 为了考察 CompTuner 中提出的轻量级学习方法是否设计合理，我们仿照 BOCA，仅使用运行时性能良好的优化序列构建预测模型，从而得到 CompTuner 的一个变体（记为 $CompTuner_{high}$）。此外，BOCA 是一种最先进的基于监督学习的技术，同样会构建预测模型。然而，与 CompTuner 不同，BOCA 通过选择期望改进最高的优化序列来构建预测模型，因此本研究也将 BOCA 构建的预测模型纳入比较。总而言之，本研究比较 CompTuner、 $CompTuner_{high}$ 和 BOCA 的预测模型，评估其预测准确率和运行时性能加速效果。此外，所有这些方法都会从一些随机选取的优化序列开始构建预测模型。为了缓解

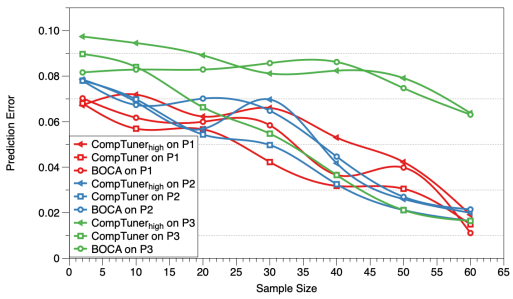


图 2. PolyBench 的预测误差率

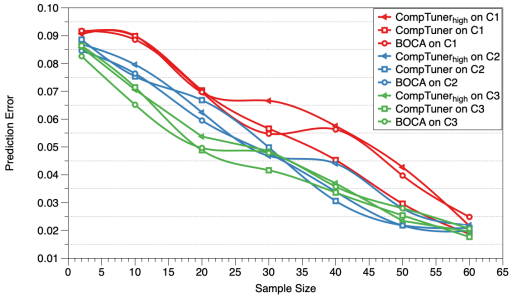


图 3. cBench 的预测误差率

为研究这种随机选择的影响，我们将相同的随机选取的优化序列作为三种方法的输入，并在构建预测模型时逐步增加所选优化序列的数量，以观察每种对比技术的预测准确率。

对于每个目标程序，我们随机生成 100 个编译器优化序列，并记录它们的实际运行时性能作为真实值。对于每种对比技术（即 Comp-Tuner、 $CompTuner_{high}$ 和 BOCA），我们首先使用不同数量的训练数据（即数量分别设为 2、10、20、30、40、50 和 60）构建预测模型，然后使用这些预测模型预测随机生成的优化序列的运行时性能，并记录每个预测模型的准确率。结果如图 2 和图 3 所示，其中横轴表示训练集的大小，纵轴表示预测误差率，即预测运行时性能与实际运行时性能之差的绝对值除以实际运行时性能。较低的预测误差率意味着预测模型性能较好。具体而言，这项消融研究使用 GCC，并选取 PolyBench 和 cBench 中的前三个目标程序作为代表，分别为 P1、P2、P3 和 C1、C2、C3。为区分这些程序，图中以红色表示 P1 和 C1，以蓝色表示 P2 和 C2，以绿色表示 P3 和 C3。

从这些图中可以看出，尽管这些技术使用相同的优化序列作为输入来构建预测模型，但它们对应模型的预测误差率却各不相同。在大多数程序中（即当输入规模不小于 20 时，六个程序中有四个），CompTuner 的预测误差率小于 $CompTuner_{high}$ 和 BOCA。特别是在 P3 中，CompTuner 的预测误差率仅为 0.02，而 $CompTuner_{high}$ 和 BOCA 的预测误差率均大于 0.06。尽管在模型构建之初（即输入的优化序列数量非常少时），CompTuner 的预测误差率有时（例如在 P1 和 P3 中）略高于 $CompTuner_{high}$ 或 BOCA，但当使用 20 个优化序列时，前者的预测误差率便会接近后者。此外，当使用超过 40 个输入数据时，CompTuner 的预测误差率通常小于 $CompTuner_{high}$ 和 BOCA，这表明只要提供数量合理的训练数据，CompTuner 始终能够生成具有较高准确率的预测模型。

此外，与 $CompTuner_{high}$相比，CompTuner 的曲线较为接近，表明其模型在各种目标程序上的预测准确率较为稳定。另一方面，从这些图中可以观察到，当使用更多训练数据时，采用 CompTuner、BOCA 或 $CompTuner_{high}$ 构建的预测模型的预测误差率均有所下降。然而，当 BOCA 和 $CompTuner_{high}$ 使用更多训练数据来调优模型时，预测误差率反而会上升（例如，$CompTuner_{high}$ 在 P2 上的情况，以及 BOCA 在 P3 上的情况），这表明这两种

这些技术在选择更多数据来调优模型时，无法确保模型的准确率。

接着，我们按照第4.1节的设置，对比 CompTuner 与 $CompTuner_{high}$的运行时性能加速效果（两者的唯一区别在于预测模型的构建方式）。请注意，我们不再比较 CompTuner 与 BOCA 的运行时性能加速效果，因为已在第4.1节中进行了比较。对比结果见表7中 CompTuner 行和 $CompTuner_{high}$ 行。与 $CompTuner_{high}$相比，CompTuner 在六个目标程序中的五个上取得了更好的优化结果，平均提升 13.29%[12]。尤其是，CompTuner 在 P2 上相较于 $CompTuner_{high}$ 提升了 33.17%。这些对比结果表明，CompTuner 的模型构建方式不仅能构建出更准确的预测模型，还能为程序带来更好的加速效果。

表7. 消融研究结果

技术	P1	P2	P3
CompTuner	**1.052**	1.059	**1.029**
$CompTuner_{high}$	1.039	1.025	1.027
$CompTuner_{impact}$	1.051	**1.067**	1.023
技术	C1	C2	C3
CompTuner	**1.299**	1.365	**1.274**
$CompTuner_{high}$	1.291	**1.416**	1.266
$CompTuner_{impact}$	1.286	1.393	1.259

4.2.2 智能搜索分析。为了探究 CompTuner 中改进的粒子群优化算法是否设计合理，我们将这一优化选择过程替换为 BOCA 的选择过程。BOCA 使用所有影响显著的编译器优化标志组合以及部分影响较小的编译器优化标志组合，生成若干编译器优化序列，由此得到 CompTuner 的一个变体（记为 $CompTuner_{impact}$）。比较结果也列于表7中。

从表中可以看出，CompTuner 在大多数程序（即六个目标程序中的四个）上取得了更好的优化结果，这证实了 CompTuner 中智能搜索的有效性。我们还注意到，$CompTuner_{impact}$ 在程序 P2（六个目标程序中规模最小的程序）上取得了比 CompTuner 更好的优化结果，因此我们推测，CompTuner 在规模更大的程序上效果更为明显。也就是说，将影响显著的标志与 CompTuner 相结合，或许是未来一个有前景的方向。

为验证我们的假设，我们选取了一些较小的程序（即 P1、P2 和 P3）以及较大的程序（即 C4 和 C5），并比较 CompTuner 和 $CompTuner_{impact}$ 的性能，如表8所示。从表中可以看出，CompTuner 相较于 $CompTuner_{impact}$ 在较大程序上的优势更为明显。具体而言，P1、P2 和 P3 分别代表行数为 200+ 的程序，而 C4 和 C5 分别代表行数为 2000+ 的程序。根据实验结果，在所选的五个程序中，$CompTuner_{impact}$ 仅在较小的程序 P2（218 行）上优于 CompTuner。此外，CompTuner 在较大的程序 C5（22,321 行）上表现更为突出。

[12]该百分比是根据表7中五个程序的平均结果计算得出的 .

表8. CompTuner 与 $CompTuner_{impact}$的比较

技术	P1	P2	P3	C4	C5
CompTuner	**1.052**	1.059	**1.029**	**1.116**	**1.035**
$CompTuner_{impact}$	1.051	**1.067**	1.023	1.109	1.021
差异	0.001	-0.008	0.006	0.070	0.014

5 有效性威胁

内部威胁主要来自编译器自动调优方法的实现。为降低这一威胁，我们使用所比较方法的可复现软件包；如果没有可用的可复现软件包，则严格按照相关论文重新实现这些方法。此外，作者还会审查实现，以进一步降低这一威胁。

外部威胁主要来自目标程序和编译器。为降低这一威胁，我们采用了与以往工作相同的基准测试集和主流编译器 [14, 23]。

构念威胁在于评估中使用的指标。GCC 和 LLVM 提供了几种推荐的优化设置，例如 -O1、-O2、-O3 和 -Ofast。据报道，-Ofast 在 -O3 的基础上增加了一些非常规优化，这些优化是通过违反某些国际标准实现的。-O1 或 -O2 不会启用许多优化标志。因此，在文献中通常不推荐使用 -Ofast，现有的编译器调优工作也通常以 -O3 为比较基准。与先前的工作一样 [14],，我们也使用相对于编译器默认 -O3 优化的运行时性能加速。

6 讨论

本节中，我们开展了三项小型研究，以探究编译器自动调优时间、编译器版本和多阶段模型构建的影响。

6.1 调优时间的影响

编译器自动调优时间可能会影响自动调优技术的性能，因此我们通过改变调优时间限制来比较编译器自动调优技术的性能。具体而言，我们在 6 个程序（即 P1、P2、P3、C1、C2 和 C3）上重复第4.1节中的实验，将时间限制设为 2,000 秒，而第4.1节中使用的时间限制为 6,000 秒。结果见表 9。

表 9. 在 2,000 秒内，GCC 和 LLVM 上的程序相较于 -O3 的性能提升

技术	P1	P2	P3	C1	C2	C3
			GCC			
CompTuner	**1.051(1923)**	**1.047(1662)**	1.041(1825)	1.215(1520)	1.197(1394)	**1.263(1882)**
RIO	✖	✖	✖	✖	**1.197(1215)**	✖
GA	✖	✖	1.041(1858)	✖	✖	✖
TPE	✖	✖	✖	1.240(1637)	1.199(1558)	1.280(1904)
BOCA	✖	1.049(1780)	**1.046(1532)**	**1.264(1482)**	✖	1.276(1964)
OpenTuner	✖	✖	✖	✖	✖	✖
			LLVM			
CompTuner	**1.038(1725)**	**1.022(1554)**	1.037(1475)	**1.020(1632)**	1.046(1913)	**1.103(1844)**
RIO	✖	✖	1.038(1926)	✖	✖	✖
GA	✖	1.024(1824)	✖	1.021(1849)	✖	✖
TPE	✖	1.040（1998）	**1.037（1216）**	✖	✖	✖
BOCA	✖	1.023(1794)	✖	1.022(1962)	**1.052(1565)**	✖
OpenTuner	✖	✖	✖	✖	1.046(1924)	✖

从该表可以看出，在 2,000 秒的评估时间内，CompTuner 在大多数程序上的表现优于对比技术。具体而言，CompTuner 在 GCC/LLVM 的程序中分别有 3/4 个取得了最佳优化结果，而对比技术 RIO、GA、TPE、BOCA 和 OpenTuner 在 GCC/LLVM 的程序中取得最佳结果的数量分别仅为 1/0、0/0、0/1、2/1 和 0/0。与 6,000 秒内的结果（见表 2 和表 3）相比，这些自动调优技术的性能提升从 2,000 秒到 6,000 秒有所增加。这一现象符合预期，因为更长的调优时间可能会提升模型的性能。不过，从 2,000 秒到 6,000 秒，CompTuner 相较于对比技术的优势始终存在。

6.2 编译器版本的影响

为降低编译器版本差异对有效性造成的威胁，我们进一步在最新版本的 GCC 和 LLVM（即 GCC 12.2.0 和 LLVM 15.0.0）上，将 CompTuner 与 BOCA [14] 进行比较，并使用与第 6 节相同的程序进行评估。结果见表10。

表10. CompTuner 与 BOCA 在 GCC 12.2.0 和 LLVM 15.0.0 上的比较

技术	P1	P2	P3	C1	C2	C3
			GCC 12.2.0			
CompTuner	**1.014(4517)**	**1.028(5262)**	**1.008(3884)**	**1.122(5753)**	1.095(4031)	**1.169(3983)**
BOCA	✖	✖	1.014(5105)	1.126(6814)	**1.174(3372)**	✖
			LLVM 15.0.0			
CompTuner	**1.021(4362)**	**1.003(3527)**	**1.032(3380)**	**1.025(4171)**	**1.041(5514)**	1.096(4793)
BOCA	✖	1.016(4107)	✖	✖	✖	**1.114(4623)**

从该表可以看出，CompTuner 的所有数据均大于 1，这表明我们提出的 CompTuner 在最新版本的 GCC 和 LLVM 上取得了优于 -O3 优化的结果。CompTuner 在 GCC 12.2.0 和 LLVM 15.0.0 上分别在 5/5 个程序中取得了最佳优化结果，而作为对比的 BOCA 仅分别在 1/1 个程序中取得了最佳结果。请注意，对于 GCC 上的 P3 和 LLVM 上的 P2，我们将 CompTuner 的结果标记为最佳结果（以粗体显示），因为 BOCA 需要更长时间才能达到与 CompTuner 相同的运行时性能加速。换言之，在 P3 上，BOCA 无法在 3,884 秒内于 GCC 中生成加速比为 1.008 的优化序列；在 P2 上，BOCA 也无法在 3,527 秒内于 LLVM 中生成加速比为 1.003 的优化序列。

总而言之，在最新版本的 GCC 和 LLVM 上，我们提出的 CompTuner 在大多数程序上的表现仍优于最先进的 BOCA。此外，结合对旧版 GCC 和 LLVM 的观察结果（即表 2 和表 3 中的结果），与旧版本编译器相比，CompTuner 在最新版本上的表现远胜 BOCA。

6.3 多阶段模型构建的影响

我们提出的 CompTuner 通过多阶段学习而非单阶段学习来构建预测模型，旨在提高预测准确率。为验证多阶段模型构建是否优于单阶段学习，我们将 CompTuner 的多阶段学习替换为单阶段学习，实现了一个变体，称为 $CompTuner_{one-time}$，并在与第 6 节相同的 6 个程序上比较 CompTuner 和 $CompTuner_{one-time}$ 的性能。为公平起见，$CompTuner_{one-time}$ 使用从候选序列中随机选出的 50 个优化序列来构建预测模型，而 CompTuner 使用通过多阶段选择选出的另外 50 个优化序列来构建模型。结果见表 11。

为了验证我们技术中性能预测模型构建部分的有效性，我们将该技术的性能与 $CompTuner_{one-time}$ 进行比较。$CompTuner_{one-time}$ 通过从候选序列中单阶段选择来构建性能预测模型。[13]。我们选择 GCC 上的 6 个程序来比较这两种技术。

表 11. CompTuner 与 $CompTuner_{one-time}$ 的比较

技术	P1	P2	P3	C1	C2	C3
预测误差率						
CompTuner	**0.031**	**0.024**	0.041	**0.030**	0.032	**0.025**
$CompTuner_{one-time}$	0.039	0.026	**0.035**	0.044	**0.014**	0.031
加速比						
技术	P1	P2	P3	C1	C2	C3
CompTuner	**1.047**	**1.053**	**1.026**	**1.285**	1.348	**1.257**
$CompTuner_{one-time}$	1.028	1.034	1.014	1.246	**1.437**	1.201

从表 11 可以看出，CompTuner 在 6 个程序中的 5 个上表现优于 $CompTuner_{one-time}$，并在 6 个程序中的 4 个上构建了更准确的预测模型。此外，误差率较低的预测模型通常会产生运行时性能较好的优化序列。然而，在程序P3上，$CompTuner_{one-time}$构建的预测模型准确率很高，但其运行时

[13]我们为两种性能预测模型构建方法各选取了50个样本。

性能低于 CompTuner。我们怀疑原因在于，CompTuner 为改进后的粒子群过程选择了更好的起始点。

7 相关工作

本文以编译器自动调优为目标，因此与编译器自动调优的研究最为相关。此外，本文也与一般的软件调优相关。因此，本节将首先在第 7.1 节简要回顾软件调优方面的现有工作，然后在第 7.2 节回顾编译器自动调优方面的工作。

7.1 软件调优

调优是工程和研究领域中的一个重要问题。许多系统和模型通常都设有大量可配置参数，可通过调优来满足不同需求。例如，MySQL 等数据管理系统的参数值可以通过调优来提升性能并确保安全性，Ngnix 等 Web 服务系统的参数值可以通过调优来确保其性能和稳定性 [26],嵌入式系统的参数值组合可以通过调优来减少运行时能耗 [47, 58], 深度学习模型有许多超参数，可以通过调优使模型获得更好的性能（例如，更高的预测准确率） [36, 49], 一些软件系统有许多参数，其参数值可以通过调优来提升其在不同硬件环境中的运行时性能 [60]。

调优广泛应用于各种软件和系统中，但它们面临的共同难题是如何高效探索庞大的参数空间。为了解决这一问题，研究人员提出了许多调优技术，可分为基于搜索的技术和基于学习的技术。具体而言，除了基于随机的方法 [15] 和基于遗传算法的方法 [23, 28],，基于模拟退火算法的技术 [21] 也得到了广泛应用；这些方法通过实际执行目标系统并采用特定的搜索策略，来寻找最优参数设置。然而，由于软件的复杂性，实际执行会消耗大量时间和计算资源，导致优化效率低下。基于学习的技术利用数据构建预测模型，进而预测参数组合的性能，而非通过实际执行进行评估。为了快速找到更优的参数组合，研究人员提出了许多采样技术，以在庞大的参数空间中近似求得最优参数组合。例如，Henard 等人 [27] 提出了一种多目标优化技术，通过约束条件缩小搜索空间；Nair 等人 [35] 和 Victoria 等人 [59] 提出了基于贝叶斯优化的技术，通过使用采集函数评估样本的性能来生成最优样本；Dogru 等人 [17] 提出了一种基于强化学习的技术，该技术利用奖励函数评估采样的未来收益，从而逐步选择搜索空间中的参数组合。此外，Chris 等人 [16] 提出了一种深度神经网络，它从原始代码中学习启发式方法，构建合适的代码表示，并学习如何实现最佳优化。

7.2 编译器自动调优

编译器自动调优旨在推荐优化设置，以提升程序运行时性能。通常，编译器自动调优包括标志选择和阶段排序。阶段排序旨在确定所选优化标志的应用顺序。例如，Ashouri 等人 [5] 提出了一种基于学习的技术，通过将 LLVM 的优化遍划分为不同的簇，并利用预训练预测模型预测完整序列的加速比。Huang 等人 [29] 提出了一种基于强化学习的技术，根据程序特征和先前应用的优化遍来预测下一个最佳标志。由于本文关注的是

标志选择而非阶段排序，因此我们将现有的标志选择技术 [4, 6]分为基于无监督学习的方法和基于监督学习的方法，并按照这一分类对现有技术进行综述。

基于无监督学习的技术。 基于无监督学习的技术从随机生成的样本开始优化过程，并使用一些搜索策略生成新的优化序列。Chen 等人 [15] 提出了一种随机迭代优化技术，旨在通过随机生成新序列来生成所需的优化序列，通常被视为编译器自动调优技术的基准。为追踪搜索过程，人们提出了一些技术（例如，基于爬山算法的技术 [2] 和基于遗传算法的技术 [23, 28, 37, 48]），通过进化来指导编译器自动调优。此外，Perez 等人 [42] 提出了 Irace，根据性能分布生成新的编译器序列。基于无监督学习的技术通过使用大量生成的优化序列实际编译并执行目标程序，来搜索并获得所需的优化序列；这一过程非常耗时。

基于监督学习的技术。基于监督学习的技术利用优化序列（及其对应的编译后程序运行时性能）构建预测模型，然后通过一些预定义的启发式方法探索优化空间。具体而言，Agakov 等人 [1] 随机选取若干优化序列来构建预测模型，并使用马尔可夫方法生成新的优化序列。随后，Ashouri 等人 [8] 提出了一种基于贝叶斯网络的技术，该技术首先构建预测模型，然后通过贝叶斯过程生成新序列，并逐步逼近所需解。Ameer 等人 [25] 提出提取程序的循环结构，对这些循环进行嵌入，然后通过深度强化学习动态确定所有循环的标志序列。最近，Chen 等人 [14] 提出了另一种基于贝叶斯网络的技术 BOCA，该技术首先构建预测模型，以区分影响显著的标志与影响较小的标志，然后通过贝叶斯过程生成新序列。

我们提出的 CompTuner 也是一种基于监督学习的技术，但它与现有的基于监督学习的技术有两点不同。首先，CompTuner 通过多个阶段的学习来构建预测模型，以缓解现有的成本问题。其次，CompTuner 使用改进的粒子群优化算法选择优化序列，通过同时考虑全局最优和局部最优来缓解性能问题。

此外，基于监督学习的技术通常使用高斯模型作为预测模型。由于编译器编译标志的数量过多，这些技术所构建的模型通常存在准确率问题，而我们提出的 CompTuner 通过改进的粒子群优化算法解决了这一问题。

最相关的工作是 **BOCA** [14]。具体而言，BOCA 首先通过迭代选择序列来构建预测模型，用于预测某个优化标志是否具有显著影响；随后分别探索影响显著的标志的搜索空间和影响较小的标志的搜索空间。CompTuner 也是一种基于监督学习的技术，与 BOCA 类似，因为二者都会使用少量数据迭代构建预测模型。然而，二者预测模型的用途不同。BOCA 的模型主要用于识别某个优化标志是否具有显著影响，而 CompTuner 的模型则用于预测优化序列的运行时性能。此外，在构建预测模型时，BOCA 只使用运行时性能良好的序列，而我们提出的 CompTuner 使用多样化的序列。

8 结论

为减少针对特定程序进行编译器调优所需的人力，已有文献提出了多种编译器自动调优技术，但它们都存在成本和性能

方面的问题。本文提出 CompTuner，一种基于多阶段学习的编译器自动调优技术。为降低成本，CompTuner 通过少量数据迭代构建预测模型，以预测优化序列的运行时性能，而非实际执行。为改善性能，CompTuner 设计了一种改进的粒子群优化算法，通过调整加速常数来平衡局部最优与全局最优。根据在 GCC 和 LLVM 上的实验结果，在大多数目标程序上，CompTuner 都取得了最佳运行时性能，显著优于最先进的编译器自动调优技术。此外，消融分析也证明了 CompTuner 各组成部分的贡献。

参考文献

[1] Felix V. Agakov、Edwin V. Bonilla、John Cavazos、Björn Franke、Grigori Fursin、Michael F. P. O'Boyle、John Thomson、Marc Toussaint 和 Christopher K. I. Williams。2006年。使用机器学习聚焦迭代优化。载于第四届 IEEE/ACM 国际代码生成与优化研讨会（CGO 2006），2006年3月26日至29日，美国纽约州纽约市。IEEE计算机学会，295–305。https://doi.org/10.1109/CGO.2006.37[2] Lelac Almagor、Keith D Cooper、Alexander Grosul、Timothy J Harvey、Steven W Reeves、Devika Subramanian、Linda Torczon 和 Todd Waterman。2004年。寻找有效的编译序列。ACM SIGPLAN 通讯 39，7（2004年），231–239。[3] Jason Ansel、Shoaib Kamil、Kalyan Veeramachaneni、Jonathan Ragan-Kelley、Jeffrey Bosboom、Una-May O'Reilly 和 Saman P. Amarasinghe。2014年。OpenTuner：一个可扩展的程序自动调优框架。载于国际并行体系结构与编译会议，PACT '14，加拿大阿尔伯塔省埃德蒙顿，2014年8月24日至27日，José Nel- son Amaral 和 Josep Torrellas（编）。ACM，303–316。[4] Amir Hossein Ashouri。2016年。使用机器学习技术进行编译器自动调优。ASHOURI_博士_论文_2016. pdf，页码（2016年），3–4。[5] Amir H Ashouri、Andrea Bignoli、Gianluca Palermo、Cristina Silvano、Sameer Kulkarni 和 John Cavazos。2017年。Micomp：利用优化子序列和机器学习缓解编译器阶段排序问题。ACM体系结构与代码优化汇刊（TACO）14，3（2017年），1–28。[6] Amir H Ashouri、William Killian、John Cavazos、Gianluca Palermo 和 Cristina Silvano。2018年。一篇关于使用机器学习进行编译器自动调优的综述。ACM计算调查（CSUR）51，5（2018年），1–42。[7] Amir Hossein Ashouri、Giovanni Mariani、Gianluca Palermo 和 Cristina Silvano。2014年。一种面向嵌入式处理器编译器自动调优的贝叶斯网络方法。载于2014年第12届IEEE实时多媒体嵌入式系统研讨会（ESTIMedia）。IEEE，90–97。[8] Amir H Ashouri、Gianluca Palermo、John Cavazos 和 Cristina Silvano。2018年。使用机器学习自动调优编译器。施普林格。[9] James Bergstra、Rémi Bardenet、Yoshua Bengio 和 Balázs Kégl。2011年。超参数优化算法。神经信息处理系统进展 24（2011年），2546–2554。[10] Eric Brochu、Vlad M. Cora 和 Nando de Freitas。2010年。昂贵代价函数贝叶斯优化教程：应用于主动用户建模和分层强化学习。CoRR abs/1012.2599（ 2010 1012 2599）。arXiv：。[11] John Cavazos、Christophe Dubach、Felix V. Agakov、Edwin V. Bonilla、Michael F. P. O'Boyle、Grigori Fursin 和 Olivier Temam。2006年。自动构建性能模型，以快速探索新硬件设计的软件。载于2006年国际嵌入式系统编译器、体系结构与综合会议（CASES 2006）论文集，韩国首尔，2006年10月22日至25日，Seongsoo Hong、Wayne H. Wolf、Krisztián Flautner 和 Taewhan Kim（编）。ACM，24–34。[12] John Cavazos、Grigori Fursin、Felix Agakov、Edwin Bonilla、Michael FP O'Boyle 和 Olivier Temam。2007年。利用性能计数器快速选择优质编译器优化。载于国际代码生成与优化研讨会（CGO'07）。IEEE，185–197。[13] "c Bench"。2022年。https://ctuning.org/wiki/index.php/CTools:CBench。[14] Junjie Chen、Ningxin Xu、Peiqi Chen 和 Hongyu Zhang。2021年。通过贝叶斯优化实现高效的编译器自动调优。载于2021年第43届IEEE/ACM国际软件工程会议（ICSE）。IEEE，1198–1209。[15] Yang Chen、Shuangde Fang、Yuanjie Huang、Lieven Eeckhout、Grigori Fursin、Olivier Temam 和 Chengyong Wu。2012年。解构迭代优化。ACM体系结构与代码优化汇刊（TACO）9，3（2012年），1–30。[16] Chris Cummins、Pavlos Petoumenos、Zheng Wang 和 Hugh Leather。2017年。端到端深度学习优化启发式方法。载于第26届国际并行体系结构与编译技术会议，PACT 2017，

美国俄勒冈州波特兰，2017年9月9日至13日。IEEE计算机学会，219–232。[17] Oguzhan Dogru、Kirubakaran Velswamy、Fadi Ibrahim、Yuqi Wu、Arun Senthil Sundaramoorthy、Biao Huang、Shu Xu、Mark Nixon 和 Noel Bell。2022年。用于自主 PID 调优的强化学习方法。Comput. Chem. Eng. 161（2022年），107760。[18] Qi Du 和 Hui Huang。2020年。ARMv8平台上快速傅里叶变换的实现与优化研究。载于IOP会议系列：材料科学与工程，第768卷。IOP Publishing，072114。[19] Qi Du、Hui Huang 和 Chun Huang。2020年。屋顶线模型引导的编译优化参数选择方法。载于《数据科学：第六届计算机科学家、工程师和教育工作者先驱国际会议》，ICPCSEE 2020，中国太原，2020年9月18日至21日，论文集，第I部分 6。施普林格，268–282。[20] Alexis Engelke 和 Martin Schulz。2020年。使用 LLVM 在运行时进行稳健的实用二元优化。载于2020年IEEE/ACM第六届HPC LLVM编译器基础设施研讨会（LLVM-HPC）及百亿亿次级计算层次化并行研讨会（HiPar）。IEEE，56–64。[21] Matteo Fischetti 和 Matteo Stringher。2019年。使用模拟退火进行嵌入式超参数调优。CoRR abs/ 1906 01504 2019 1906 01504 1906 01504 。（）。arXiv：。[22] Peter I Frazier。。贝叶斯优化教程。arXiv预印本 arXiv:1807.02811（2018年）。[23] Unai Garciarena 和 Roberto Santana。2016年。通过学习并利用标志间的交互来进化优化编译器标志选择。载于2016年遗传与进化计算会议伴随会议论文集。1159–1166。[24] “GCC”。2022年。https://gcc.gnu.org。[25] Ameer Haj-Ali、Nesreen K. Ahmed、Theodore L. Willke、Yakun Sophia Shao、Krste Asanovic 和 Ion Stoica。2020年。NeuroVectorizer：利用深度强化学习实现端到端向量化。载于CGO '20：第18届ACM/IEEE国际代码生成与优化研讨会，美国加利福尼亚州圣迭戈，2020年2月。ACM，242–255。[26] Haochen He、Zhouyang Jia、Shanshan Li、Yue Yu、Chenglong Zhou、Qing Liao、Ji Wang 和 Xiangke Liao。2022年。面向多意图的性能调优配置选择。载于第44届IEEE/ACM国际软件工程会议，ICSE 2022，美国宾夕法尼亚州匹兹堡，2022年5月25日至27日。ACM，1431–1442。https://doi.org/10.1145/3510003.3510094[27] Christopher Henard、Mike Papadakis、Mark Harman 和 Yves Le Traon。2015年。结合多目标搜索与约束求解来配置大型软件产品线。载于2015年第37届IEEE/ACM国际软件工程会议，第1卷。517–528。[28] Kenneth Hoste 和 Lieven Eeckhout。2008年。Cole：编译器优化级别探索。载于第6届IEEE/ACM代码生成与优化年度国际研讨会论文集。165–174。[29] Qijing Huang、Ameer Haj-Ali、William Moses、John Xiang、Ion Stoica、Krste Asanovic 和 John Wawrzynek。2019年。Autophase：利用深度强化学习进行高层次综合的编译器阶段排序。载于2019年IEEE第27届现场可编程定制计算机年度国际研讨会（FCCM）。IEEE，308–308。[30] Michael R. Jantz 和 Prasad A. Kulkarni。2013年。动态即时编译期间优化阶段选择的性能潜力。载于ACM SIGPLAN/SIGOPS虚拟执行环境国际会议（与ASPLOS 2013同期举行），VEE '13，美国得克萨斯州休斯敦，2013年3月16日至17日，Steve Muir、Gernot Heiser 和 Stephen M. Blackburn（编）。ACM，131–142。[31] James Kennedy 和 Russell Eberhart。1995年。粒子群优化。载于ICNN'95——国际神经网络会议论文集，第4卷。IEEE，1942–1948。[32] Adam Lipowski 和 Dorota Lipowska。2012年。通过随机接受实现轮盘赌选择。Physica A：统计力学及其应用 391，6（2012年），2193–2196。[33] “LLVM”。2022年。https://llvm.org。[34] Antoine Monsifrot、François Bodin 和 Rene Quiniou。2002年。一种自动生成编译器启发式方法的机器学习方法。载于人工智能国际会议：方法、系统与应用。施普林格，41–50。[35] Vivek Nair、Zhe Yu、Tim Menzies、Norbert Siegmund 和 Sven Apel。2020年。使用FLASH寻找更快的配置。IEEE软件工程汇刊 46，7（2020年），794–811。[36] Sajjad Nematzadeh、Farzad Kiani、Mahsa Torkamanian-Afshar 和 Nizamettin Aydin。2022年。使用元启发式算法调优机器学习算法和深度神经网络的超参数：一项关于生物医学与生物学案例的生物信息学研究。Comput. Biol. Chem. 97 2022 107619 10 1016 2021 107619 [37] 2019 （），。https://doi.org/。/j.compbiolchem.。Youcong Ni、Rui Wu、Xin Du、Peng Ye、Wangbiao Li 和 Ruliang Xiao。。基于频繁模式挖掘的GCC编译时能耗优化进化算法。J. Softw 30，5（2019年），1269–1287。[38] “NumPy”。2022年。https://numpy.org。[39] David A Padua 和 Michael J Wolfe。1986 29，12 1986 1184 1201 。面向超级计算机的高级编译器优化。Commun. ACM（），–。

[40] Eunjung Park、John Cavazos、Louis-Noël Pouchet、Cédric Bastoul、Albert Cohen 和 Ponnuswamy Sadayappan。2013。多面体优化空间中的预测建模。《国际并行编程杂志》41，5（2013），704–750。[41] Juliana Alves Pereira、Mathieu Acher、Hugo Martin 和 Jean-Marc Jézéquel。2020。采样对可配置系统性能预测的影响：案例研究。载于 ICPE 2020-第11届 ACM/SPEC国际性能工程会议。ACM，1–13。[42] Leslie Pérez Cáceres、Federico Pagnozzi、Alberto Franzin 和 Thomas Stützle。2017。使用 Irace 自动配置 GCC。载于国际人工进化会议（人工进化）。施普林格，202–216。[43] Riccardo Poli、James Kennedy 和 Tim Blackwell。2007。粒子群优化。《群体智能》1，1（2007），33–57。[44] “PolyBench”。2022。https://web.cse.ohio-state.edu/~pouchet.2/software/polybench/。[45] Suresh Purini 和 Lakshya Jain。2013。寻找覆盖程序空间的优良优化序列。《ACM体系结构与代码优化汇刊》（TACO）9，4（2013），1–23。[46] Junqiao Qiu、Lin Jiang 和 Zhijia Zhao。2020。通过有原则的逐位推测，攻克顺序位流处理难题。载于第二十五届国际编程语言与操作系统体系结构支持会议论文集。607–621。[47] Piyanun Ruangurai 和 Chaiyaporn Silawatchananai。2019。基于粒子群优化的虚拟参考迭代调谐在嵌入式系统中的实现。载于第七届国际控制、机电一体化与自动化会议，ICCMA 2019， 6 8, 2019 365 369 10 1109 46720 2019 8988608 [48] 2012 代尔夫特，荷兰，11月-。电气电子工程师学会，–。https://doi.org/。/ICCMA。。Thayalan Sandran、Nordin Zakaria 和 Anindya Jyoti Pal。。一种基于编译时长和执行时长、用于编译器标志选择的遗传算法参数优化调谐方法。载于国际问题求解软计算会议论文集（SocProS 2011），2011年12月20日至22日。施普林格，599–610。[49] Chompunuch Sarasaen、Soumick Chatterjee、Mario Breitkopf、Georg Rose、Andreas Nürnberger 和 Oliver Speck。2021。利用先验知识微调深度学习模型参数，以提升动态磁共振成像的超分辨率效果。CoRR abs/ 2102 02711 2021 2102 02711 2102 02711。（）。arXiv：。https://arxiv.org/abs/。[50] Paul B Schneck。。编译器优化技术综述。载于 ACM年度会议论文集。106–113。[51] “scikit-learn”。2022。https://scikit-learn.org/stable/。[52] Bobak Shahriari、Kevin Swersky、Ziyu Wang、Ryan P Adams 和 Nando De Freitas。2015。将人类排除在循环之外：贝叶斯优化综述。《IEEE会刊》104，1（2015），148–175。[53] Yuhui Shi 和 Russell C Eberhart。1999。粒子群优化的实证研究。载于1999年进化计算大会论文集（CEC99）（目录号99TH8406），第3卷。电气电子工程师学会，1945–1950。[54] Norbert Siegmund、Alexander Grebhahn、Sven Apel 和 Christian Kästner。2015。高度可配置系统的性能影响模型。载于2015年第10届软件工程基础联合会议论文集。284–294。[55] Jasper Snoek、Hugo Larochelle 和 Ryan P Adams。2012。机器学习算法的实用贝叶斯优化。载于《神经信息处理系统进展》25（2012）。[56] Mark Stephenson、Saman Amarasinghe、Martin Martin 和 Una-May O’Reilly。2003。元优化：利用机器学习改进编译器启发式方法。《ACM SIGPLAN 通讯》38，5（2003），77–90。[57] Ananta Tiwari、Chun Chen、Jacqueline Chame、Mary Hall 和 Jeffrey K Hollingsworth。2009。用于编译器优化的可扩展自动调优框架。载于2009年IEEE并行与分布式处理国际研讨会。电气电子工程师学会，1–12。[58] Ruben Vazquez、Ann Gordon-Ross 和 Greg Stitt。2019。嵌入式系统中用于缓存调优的能耗预测。载于第37届IEEE计算机设计国际会议（ICCD 2019），阿布扎比，阿拉伯联合酋长国，11月 17 20, 2019 630 637 10 1109 46524 2019 00091 -。电气电子工程师学会，–。[59] 2021 https://doi.org/。/ICCD。。Helen Victoria 和 Ganesh Maragatham。。使用贝叶斯优化自动调优超参数。《演化系统》12（2021年3月），12–30。[60] Yu Wang、Shu Jiang、Weiman Lin、Yu Cao、Longtao Lin、Jiangtao Hu、Jinghao Miao 和 Qi Luo。2021。用于大规模系统部署中自动驾驶车辆控制的基于学习的自动参数调优框架。载于2021年美国控制会议（ACC 2021），美国路易斯安那州新奥尔良，2021年5月25日至28日。电气电子工程师学会，2919–2926。https: //doi.org/ 10 23919 50511 2021 9482827。/ACC。。Jun Ye。。直觉模糊集的余弦相似性度量及其应用。《数学与计算机建模》53，1-2（2011），91–97。[61] 2011