Magellan：借助 AlphaEvolve 自主发现新型编译器优化启发式方法

Hongzheng Chen[1,3] Alexander Novikov[2] Ngân (NV) V˜u[2] Hanna Alam[1] Zhiru Zhang[3]
Aiden Grossman[1] Mircea Trofin[1] Amir Yazdanbakhsh[2]1 谷歌 [2] GoogleDeepMind [3] 康奈尔大学

arXiv:2601.21096v1 [cs.AI] 28 Jan 2026

摘要

现代编译器依靠人工设计的启发式方法来指导优化流程。这些由人设计的规则往往难以适应现代软件和硬件的复杂性，并带来沉重的维护负担。为应对这一挑战，我们提出了 Magellan，这是一种智能体框架，通过合成可执行的 C++ 决策逻辑来演进编译器流程本身。Magellan 将大语言模型编码智能体与进化搜索和自动调优结合起来，形成一个闭环：生成启发式方法，在用户提供的宏基准测试上进行评估，并不断改进，从而生成可直接集成到现有编译器中的精简启发式方法。在多个生产级优化任务中，Magellan 发现的策略可媲美或超越专家基线。在 LLVM 函数内联任务中，Magellan 合成的新启发式方法在二进制文件大小缩减和端到端性能方面均优于数十年来的人工工程成果。在寄存器分配任务中，它学会了一条简洁的活跃区间处理优先级规则，在大规模工作负载上可媲美复杂的人类设计策略。我们还报告了 XLA 问题上的初步结果，展示了该方法在 LLVM 之外的可移植性，同时减少了工程投入。

1 引言

编译器是最成熟且经过系统化工程设计的软件系统之一，其许多优化问题都是 NP 困难 [1],，使得在生产规模下求得精确解并不现实。因此，现代编译器高度依赖手工设计的启发式方法 [2–6],，这些方法编码了专家对复杂成本效益权衡的直觉。尽管这些启发式方法推动了数十年的发展，但它们在设计、调优和维护方面固有地十分困难。随着软硬件生态系统变得更加异构，这一挑战正日益加剧：长期沿用的规则所依据的假设不断失效，人们也需要持续投入精力，才能使编译器性能保持竞争力。

大语言模型（LLM）的近期进展重新激发了人们对自动化启发式方法设计的兴趣。大语言模型与进化式、反馈驱动的搜索相结合，用于发现适用于定义明确的优化任务的算法和启发式方法 [7–11]。在编译器领域，大多数基于大语言模型的工作分为两个方向：要么（1）通过直接合成目标代码来取代代码生成 [12, 13],，要么（2）搜索优化序列，以促使现有编译器取得更好的结果 [14–17]。相比之下，我们的目标有所不同。我们并不打算为每个程序生成独立的遍历序列，也不打算绕过编译器来生成代码。相反，我们以编译器的**优化决策逻辑**本身为目标，通过**演进编译器遍历代码**，生成精简、可部署的启发式方法，以便将其集成到上游并在各类应用程序中重复使用。

我们认为，这种遍级进化在手工工程与神经网络（NN）策略集成之间提供了一条切实可行的折中路径。[18, 19] 中介绍的以往基于 NN 的策略发现方法表明，学习得到的策略可以达到或超过专家启发式方法的性能，但通常需要将神经网络模型集成到编译器基础设施中 [20–22]。对于新的遍、新的目标或新兴加速器，复现这种集成可能代价高昂且耗时。相比之下，我们的方法将启发式方法设计视为一个程序综合问题，直接在可执行的 C++实现上进行搜索，使编译器能够在真实的宏观基准测试上对其进行编译和评估。因此，所发现的启发式方法保留了与人工编写的编译器代码相同或极为相近的部署与维护特性。

本文介绍了 Magellan，这是一种自主框架，通过将由大语言模型驱动的编码智能体与进化搜索和自动调优相结合，来发现编译器优化启发式方法。Magellan 面向真实世界的大规模生产级编译器优化而设计：用户可以直接使用端到端的应用程序性能指标作为奖励信号，而 Magellan 则会迭代地提出、评估并改进可直接集成到编译器中的优化遍实现。我们发现，即使对于函数内联这类基础且经过深入研究的优化， [23],Magellan 也能合成出新的启发式方法，超越数十年来人工设计所取得的成果，这表明借助可扩展的自动化搜索进行探索，成熟的编译器通道仍蕴藏着尚未发掘的优化潜力。

我们通过对启发式方法驱动的 LLVM 优化遍（具体包括函数内联和寄存器分配）进行深入案例研究，展示了 Magellan 的通用性和实际影响，并将评估扩展到 XLA 中的其他任务。总的来说，Magellan 大幅简化了编译器中的自动启发式方法发现：它将这一过程转化为对人类可读、可部署的编译器代码进行直接搜索，既提升了效率，也带来了相较于专家基线的可衡量改进。我们的主要贡献如下：

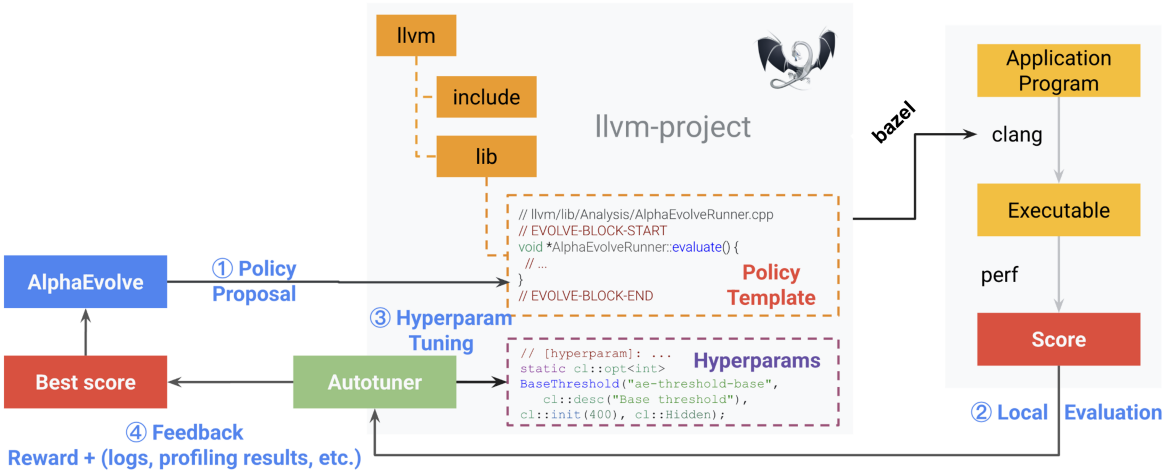

图 1. Magellan 概览。我们以 LLVM 为示例进行说明，但 Magellan 也能轻松应用于其他编译器。

- 我们推出了 Magellan，这是一款由 AlphaEvolve [10] 驱动的端到端编译器优化智能体，通过合成并评估可执行的 C++ 启发式方法来演进编译器优化遍，并可通过简单的集成流程接入现有编译器。
- 我们在 Magellan 中引入了一种分层搜索策略，通过结合大语言模型提出启发式模板，并由自动调优器在搜索过程中优化超参数，从而提高演化效率。
- 我们展示了 LLVM 和 XLA 中真实编译器优化任务的案例研究，结果表明，与专家制定的基线相比，代码尺寸有所缩减，运行时也获得了加速。

2 Magellan 设计

图 1 展示了 Magellan 的四阶段工作流程：该流程将 AlphaEvolve 与 LLVM 编译器相结合，以自动发现并改进优化启发式方法。LLVM 可替换为 XLA 等其他基础设施，以适配不同的编译器遍和后端。Magellan 的核心思想是一种分层搜索策略，将高层策略设计与底层参数调优分离：大语言模型提出带有符号超参数的启发式模板，随后由外部自动调优器填入这些参数值。系统以闭环流程运行，包含**（1）策略提议**、**（2）局部评估**、**（3）超参数调优**和**（4）反馈生成**，形成一个逐步改进编译器决策策略的迭代循环。

2.1 策略提议

在第一阶段，Magellan 使用配备大语言模型的 AlphaEvolve 修改一个包含待改进策略的源文件。与策略相关的区域通过 EVOLVE-BLOCK 注释标记，这是一种由 AlphaEvolve 提供的、用于界定可编辑代码的机制。每个候选方案都为某个编译器优化过程定义一项决策策略，并符合 LLVM 标准 API 接口。为了构建搜索空间，我们提示大语言模型生成一个策略模板，其中可调常量被参数化为显式的编译器标志，使模型能够专注于高层决策逻辑，而不必执着于具体的数值阈值。随后，系统会自动将合成的模板插入 LLVM 代码库进行评估。

2.2 局部评估

在生成带有默认超参数的候选启发式方法后，编译器会使用第一阶段所述的修改后源文件重新编译，并在具有代表性的宏基准测试上评估所得编译器，以获得端到端奖励信号。在我们当前的实现中，我们重新构建编译器，并在基准测试套件上执行所得工具链，报告二进制大小（例如通过 llvm-size）或运行时性能（例如通过 perf stat）等指标。该设置使评估立足于真实工作负载而非合成目标，同时也使我们能够量化不同配置下取得进展所需的采样复杂度。重要的是，Magellan 并不局限于完整的端到端执行：当此类评估开销过高时，同一循环可以转而依赖代理信号（例如静态成本模型或更快的近似方法）作为奖励，以保真度换取可扩展性。

2.3 超参数调优

随后，系统会将得到的性能得分传递给黑盒自动调优器 [24], ，由其根据以往的评估结果提出新的超参数配置。在这一阶段，程序模板保持不变，同时迭代调整超参数，以寻求更优的

性能。经过固定轮数的调优后，系统会将性能最佳的配置返回给 AlphaEvolve，作为进一步优化策略的反馈。

2.4 反馈整合

在最后阶段，评估结果（包括最佳得分、调优日志和性能分析跟踪记录）会返回给 AlphaEvolve。随后，AlphaEvolve 会在策略模板上执行进化搜索，结合这些信号与经验奖励来选择有潜力的候选方案，并为下一轮迭代生成新的变体。然后，系统会评估新的候选方案并重复这一循环，通过连续多轮的选择与变异，逐步完善策略模板。

3 案例研究

我们通过将 Magellan 应用于多个关键的编译器优化问题，展示其实际应用价值。除非另有说明，所有实验均使用 Gemini-2.5-Pro [25]作为基础模型。默认情况下不进行自动调优。如果进行自动调优，则使用 Vizier [24]作为自动调优器。我们的实验基于谷歌内部的 clang/LLVM 工具链开展，该工具链紧密跟踪 LLVM 主代码库的最新版本，而不是上游发布的快照；2025 年夏季期间的提交代表了本研究所使用的版本。

3.1 面向二进制大小的函数内联

函数内联是一种经典的编译器优化，它会将函数调用替换为相应被调用函数的函数体 [6]。这种转换消除了调用开销，并带来了更多优化机会，例如常量传播、死代码消除，以及跨调用边界的寄存器分配。然而，内联也可能增大代码大小并降低指令缓存的性能，因此决定何时内联以及内联哪些函数并非易事。事实上，已知全局最优的内联方案属于 NP 困难问题 [23], ，因此现代编译器依赖启发式成本模型，在性能收益、代码膨胀和资源限制之间取得平衡。

具体而言，我们选择以缩减二进制大小为目标的函数内联作为 Magellan 的试点问题，因为它能提供清晰且确定的奖励信号，也便于我们将 Magellan 与作为神经网络策略的精确替代方案进行对比 [18]。这项优化十分重要，应用于谷歌的边缘部署场景，例如 Fuchsia OS、Android 版 Chrome，以及云基础设施。我们考虑了两种设置，示例如图 2 所示：

(a) **部分启发式**：通过扩展 MLModelRunner::evaluateUntyped（用于嵌入神经网络的接口）定义的基于特征的策略 [18]。在这种情况下，

```
1 (a) 部分启发式方法定义。 2 //
llvm/lib/Analysis/AlphaEvolveRunner.cpp 3 //
EVOLVE-BLOCK-START 4 void*AERunner::evaluateUntyped(){ 5
std::unique_ptr<InlineAdvice>6 int64_tCallerUsers =7 *getTensor<
int64_t>(FeatureIndex::caller_users); 8 int64_tCalleeUsers =9
*getTensor<int64_t>(FeatureIndex::callee_users); 10 // ... 11 } 12 //
EVOLVE-BLOCK-END 13 14 (b) 完整启发式方法定义。 15 //
llvm/lib/Analysis/AlphaEvolveRunner.cpp 16 //
EVOLVE-BLOCK-START 17
AEInlineAdvisor::getAdviceImpl(CallBase&CB){ 18
boolIsInliningRecommended =false; 19 Function *Callee =
CB.getCalledFunction(); 20 Function *Caller =CB.getCaller(); 21 // ...
22 } 23 //EVOLVE-BLOCK-END
```

图 2。 函数内联的可进化模块。

Magellan 需要组合 38 个预定义特征[1] （例如调用点和调用者/被调用者用法），以形成新的策略。

（b）**完整启发式方法**：通过 MLInlineAdvisor::getAdviceImpl 实现的 API 级决策逻辑。Magellan 仅接收一个 CallBase 对象，必须推断应调用哪些 LLVM API 来构建完整策略。请注意，从 CallBase 开始，LLVM 的 IR 接口允许遍历周围的上下文，包括外围的基本块和函数、包含该调用点的任意循环，最终还可遍历整个编译单元。

在这两种设置中，启发式方法都会返回一个二元决策，指示是否应内联该调用点。在调用启发式方法之前，现有的 MLInlineAdvisor 基础设施会检查内联决策是否合法。这确保了任何能够成功集成到编译器中的策略都能始终保证编译器输出正确。

我们使用内部搜索应用评估 Magellan，并从始终返回 false 的朴素策略开始。基线是上游 LLVM 内联启发式[2]。在第一种设置中，Magellan 使用从 MLInlineAdvisor [18], 中提取的特征组合启发式方法，这些特征旨在作为神经网络的输入。不同于先前使用的基于神经网络的训练技术 [18],，我们从最终的链接后的二进制文件中计算奖励，而不是编译编译单元的随机样本，并使用生成的目标文件中的 .text 节大小。在这种基于特征的设置中，早期改进十分迅速：在几十次迭代内，搜索结果便持续优于基线，并收敛至稳定平台期，实现了 4.27% 的大小

[1]https://github.com/llvm/llvm-project/blob/c878baf1d9a259cf0788ffa1cc5c9d065adcb4c5/llvm/include/llvm/Analysis/ InlineModelFeatureMaps.h[2] https://github.com/llvm/llvm-project/blob/main/llvm/lib/Analysis/InlineCost.cpp

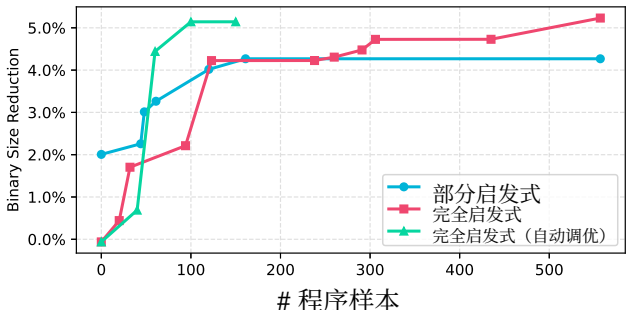

图 3. 内部搜索基准上二进制大小缩减的演化曲线。每个程序样本都对应一个特定的策略模板，以及自动调优器提出的一组超参数。两种方法的初始点有所不同，因为预定义特征会在搜索开始前影响内联决策。

相较于 LLVM 的上游启发式方法，在经过 1.5 天的顺序探索后，大小减少了。

在第二种设置中，Magellan 直接使用 LLVM API 生成完整的内联启发式方法。如图 3 所示，由于没有预定义特征来引导决策，API 级搜索的初始点较弱，导致早期需要经历更长的试错阶段。然而，它在基于特征的策略进入平台期后仍持续改进，展现了更具表达力的搜索空间所带来的优势。在为期 1.5 天的搜索窗口结束时，基于 API 的方法较人工设计的启发式方法提升了 5.23%，表现超过了基线和基于特征的设置。值得注意的是，两种方法的演化曲线不仅最终性能不同，学习动态也有所差异：基于特征的搜索收敛更快，但较早达到饱和；而基于 API 的搜索探索得更积极，最终取得了更大的提升。我们将在第 3.1.3 节详细分析合成策略。

3.1.1 关注点分离的有效性。 为评估将高层策略设计与底层参数调优分离所带来的收益，我们为完整启发式设置增加了一个外部自动调优器，该自动调优器仅搜索超参数，同时保持策略模板固定。在每次外层迭代中，自动调优器提出一批 10 个候选超参数配置，并由 Magellan 选出表现最佳的配置用于下一次迭代。如图 3 所示，与让大语言模型同时搜索策略逻辑和超参数相比，这种解耦策略显著加快了收敛速度，并获得了更高的最终收益。尽管未经调优的完整启发式设置已经优于基于特征的设置，加入自动调优后，通过在大语言模型极少参与的情况下探索更丰富的数值设计空间，进一步提升了二进制大小缩减。仅在 10 次外层迭代（100 个程序样本）内，

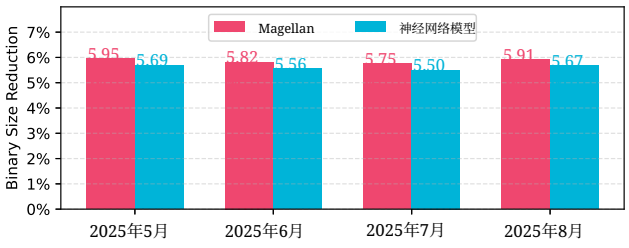

图 4.函数内联的时间泛化。

Magellan 在大约 5 小时内实现了超过 5% 的大小缩减。这一改进源于更高的采样效率就其构造而言：一旦策略模板能够编译通过，自动调优器便可探索多种超参数设置，而无需重新生成策略，尤其无需反复重新编译编译器。因此，当大语言模型直接提出完整启发式时，无效率超过 65%，而采用自动调优后降至仅 13%，从而使浪费的样本减少了 5×。综合来看，这些结果凸显了将高层策略设计与底层参数调优分离的有效性与实用性。

3.1.2 时间泛化与领域泛化。 为了了解生成的策略的泛化能力，我们在不同时间戳和不同应用程序上评估同一策略。

由于谷歌使用单体式源代码仓库 [26], , 策略是在特定时间戳训练的，而底层应用程序和代码库会随时间演进。我们采用图 3 中 Magellan 生成的最佳策略，通过测量该策略在连续数月收集的应用程序基准测试的四个快照上的性能，评估其时间泛化能力。我们将演进后的启发式方法与当前使用的神经网络[3], 进行比较；该网络约在 2 年前训练，先采用进化策略 [27]在大型内部代码库上进行训练，随后通过离线模仿学习 [28]在相关内部二进制文件上进一步调优。尽管编译器内部机制和工作负载特征持续变化，Magellan 仍始终取得更优或相当的结果，使代码大小减少 5.75% 至 5.95%，并略胜于神经网络。这些结果表明，Magellan 的 C++ 策略能够跨时间稳健泛化，无需持续重新训练或微调。

图 5 将分析扩展至领域泛化，评估 Magellan 在十多个使用生产环境优化标志编译的生产环境二进制文件上的表现。每个二进制文件都代表一个不同的应用领域，包括搜索基础设施和嵌入式工作负载。在此，Magellan 实现了 8.79% 的平均体积缩减率，与

[3]https://github.com/google/ml-compiler-opt/releases/tag/inlining-Oz- v1.2。LLVM 使用现有的树内 MLGO 功能来实现这一点，本文不再对此作详细介绍。

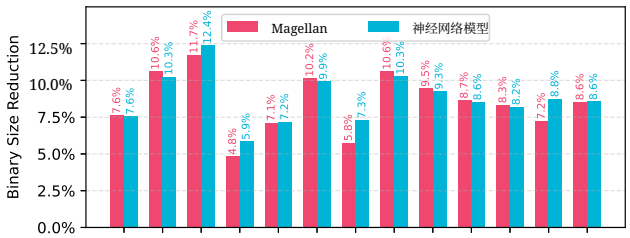

图 5. 函数内联的领域泛化。

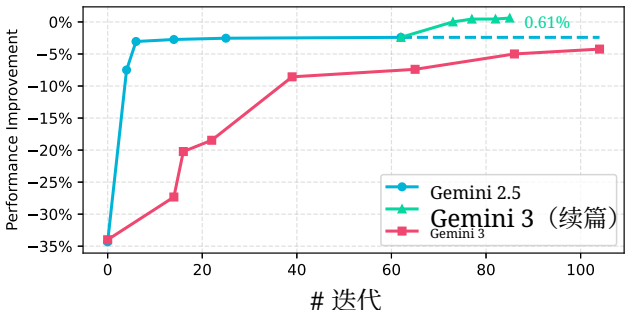

图 6. clang 基准测试上性能改进的演化曲线。

神经模型的 8.52% 相近，并且在大多数单个二进制文件上表现更优。在异构代码库中观察到的稳定表现表明，Magellan 发现的启发式方法捕捉到了广泛适用的结构模式，而非过度拟合特定基准测试。此外，这种泛化能力无需重新训练模型或重新设计特征。综上，这些结果验证了 Magellan 能够生成兼具时间鲁棒性和领域泛化能力的启发式方法，满足在不断演进的编译器工具链中实际部署的一项标准。

3.1.3 代码可维护性。对 Magellan 合成的启发式方法进行详细分析后发现，与 LLVM 手动设计的基线相比，生成的代码精简得多，同时优化质量也更高。演化得到的内联策略仅包含 143 行可执行逻辑，如附录 A 所示；去除注释和空行后，比 2,115 行的手动实现短约 15×，却能轻松集成到 LLVM 中，并持续实现更大幅度的代码体积缩减。其简洁性源于消除了冗余的分支条件和特殊情况逻辑，只保留了在内联收益与代码膨胀之间取得平衡所需的关键决策。不过，我们强调，代码简洁并不等同于普遍意义上的可维护性：尚不清楚范围与默认启发式方法相当的多目标策略（例如同时优化代码大小和性能）能否保持同样简洁。

此外，Magellan 通过认识到有策略地内联可以缩小二进制文件，而不只是避免代码膨胀，从而实现了更大幅度的代码体积缩减。其关键洞见是积极内联单次使用函数，使调用点和整个函数体都能消失，并优先处理只读函数，以便在内联后进行大规模死代码消除。生成的策略还采用加权复杂度模型，对控制流指令施加更重的惩罚，并对常量参数和精确类型匹配给予奖励，因为它们表明存在简化机会。与 LLVM 保守的策略相比

这种倾向于少做内联以缩小体积的策略认识到，内联合适的函数，尤其是仅调用一次且没有副作用的函数，往往能通过释放下游优化的空间，使最终生成的二进制文件更小。

3.2 通过函数内联提升性能

我们进一步在更具挑战性的任务上评估 Magellan：以 LLVM 上游启发式方法为基线，通过函数内联提升端到端性能。与二进制文件大小缩减一样，奖励信号较为稀疏，但性能反馈不仅带有噪声，获取成本也高得多，因为除了重新编译之外，还需要运行有代表性的宏基准测试；对于具有平坦配置文件和多样化热点路径的大型生产工作负载，微基准测试往往无法很好地反映其表现 [29]。为部分控制这项开销，我们使用 clang 本身作为宏基准测试，认为它具有代表性，同时评估起来切实可行。基准测试二进制文件采用相同的配置文件引导优化（PGO）配置文件构建，该配置文件通过 IR 插桩获取；基线组和实验组均使用分布式 ThinLTO [30] 和 -O3 优化级别。我们从一个朴素策略开始，该策略对所有内联决策均返回 false；在所有实验中，自动调优器每次迭代会提出 40 组超参数配置。以 Gemini-2.5-Pro 为基础模型时，Magellan 从较差的起点逐步恢复并发现了有意义的改进，但搜索最终陷入停滞：经过 60 次迭代后，性能回退约 2.4%，且未能超越基线。我们还从相同的朴素 false 策略出发评估了 Gemini-3-Pro [31]，但经过 100 次迭代（约一周的顺序评估）后，它同样未能突破 0%，这表明在这种搜索设置下，即使使用更强的大语言模型，从头开始也不足以突破性能上限。

为判断这一平台期究竟反映了难以逾越的性能上限，还是搜索限制，我们进行了进一步实验：将 Gemini-2.5-Pro 生成的性能最佳启发式方法作为 Gemini-3-Pro 的初始化种子，同时保持相同的自动调优配置，即每轮提出 40 个超参数提案。该

延续策略将搜索空间限制在一个结构良好的启发式方法附近，并提供归纳提示，使 Gemini-3-Pro 得以突破 0% 的平台期。如图 6 所示，延续运行的 Gemini-3-Pro 持续取得高于 0% 的正加速比，最终超越手动调优的基线 0.61%。演化策略通过采用轻量级成本模型和调优良好的启发式方法，胜过 LLVM 的上游内联器：它会跳过零开销指令，对常量参数、循环嵌套和向量代码给予大幅奖励，并针对热点调用点大幅提高阈值，从而无需昂贵的分析开销，就能内联最关键的情形（附录 B）。据我们所知，尽管此前在 LLVM 中部署的神经网络策略也展现出类似突破性能上限的表现 [32],，但迄今为止，这类结果仅在微基准测试而非大型工作负载上得到验证，且未借助（或以之为基线）PGO 和（Thin）LTO 等最先进的性能优化技术 [30]。这些发现表明，大型模型结合局部超参数调优能够开辟性能空间中的新区域，凸显了在稀疏且噪声较大的目标下实现自动化性能优化的一条有前景的路径。

3.3 其他问题的初步结果

除了函数内联，我们还在 LLVM 和 XLA 中评估了 Magellan 在其他优化任务上的表现。LLVM 中的另一项任务是寄存器分配。在这一设置下，Magellan 针对 RegAllocGreedy[4] 中用于活跃范围处理的优先队列策略。在大规模生产搜索工作负载上，Magellan 收敛到一个出人意料地简单的解决方案：常值策略。这表明，在此配置中，仅调整活跃范围插入的顺序就足以实现出色的性能。尽管演化策略十分简单，其性能仍可媲美精心设计的人工启发式方法；经过相对较少的迭代，性能便从 -0.55% 提升至 -0.15%，表明其迅速收敛至近乎最优的行为。

为了展示 Magellan 在 LLVM 之外的适用性，我们进一步在 XLA 编译器栈中对其进行评估，重点关注图重写和自动分片。图重写任务基于 Enzyme-JAX [33, 34], , 它使用等式饱和 [35] 穷尽地应用重写规则，并在 e-图中表示许多等价的计算图。饱和之后，关键挑战是提取：在成本模型下选择单个优化后的图。e-图将等价表达式分组为 e-类，每个 e-类包含多个 e-节点，其子节点引用其他 e-类。一次有效的提取从指定的根开始，为每个可达的 e-类选择一个等价节点，确保选择某个等价节点时也会从其每个子 e-类中恰好选择一个代表，并避免环。该问题是 NP 困难并且通常用整数线性规划或启发式方法 [36, 37]来解决。在我们的设定中，每个等价节点被赋予一个开销，目标是最小化提取程序的总开销。Magellan 合成新的提取启发式方法来指导这些选择决策，相较人工设计的策略提升了 7%，并展示了其应对基于等价饱和的图优化中固有组合权衡的能力。

第二项任务针对 XLA 中用于 TPU 分布式执行的自动分片 [38–40] ，遵循 ASPLOS'25 IOPDDL竞赛的设定 [41]。该竞赛将问题定义为计算图上的约束组合优化任务：每个节点都必须从离散候选集中选择一种分片策略，目标是在确保时变内存使用量不超过全局预算的同时，最小化计算、设备间通信和重新分片的总开销。此任务的输入是源自真实模型（如扩散模型和 Gemma）的图实例，其中节点提供多种有效策略，边则体现不同策略选择之间的通信影响。我们遵循竞赛方案，使用相同的公开基准实例进行训练，并使用私有实例进行评估（竞赛划分中有 5 个公开实例和 20 个私有实例），同时采用与竞赛相同的目标和评分函数。初始化时，我们将官方问题规范[5]输入 Gemini-2.5-Pro，以获得初始解决方案；随后应用 Magellan 演化启发式策略，使其在遵守内存约束的同时选择分片决策，以降低全局开销。经过一周的演化，Magellan 的性能已可媲美竞赛中的顶尖提交方案，且无需人工设计成本模型。尽管竞赛组织者仍在推进将其集成到完整 XLA 流程中，但这些结果表明，Magellan 可应用于大规模机器学习编译器优化问题，且适用范围不局限于 LLVM。

4 相关工作

近年来，大语言模型（LLM）与进化搜索和反馈驱动搜索相结合，用于解决指标明确的传统优化问题。FunSearch [7] 表明，将大语言模型与进化程序搜索相结合，通过反复改进程序，可以为帽集和装箱问题等传统组合优化问题找到更优解。后续研究将大语言模型纳入进化框架，利用反思、交叉和变异来改进候选启发式方法 [8, 9]。AlphaEvolve [10]将这一范式拓展到整个代码库的进化，而非孤立的函数，并展示了其在多种科学与工程应用中的算法改进效果。

[4]https://github.com/llvm/llvm-project/blob/main/llvm/lib/CodeGen/RegAllocGreedy.cpp

[5]https://github.com/asplos-contest/2025/blob/main/IOPDDL.md

ShinkaEvolve [11]进一步推进了这一方法，采用多种技术来提高样本效率并改善探索行为。

这些由大语言模型引导的进化框架取得了成功，激发了人们对系统和编译器优化问题的兴趣，这类问题通常涉及组合搜索空间。为推动这一方向的发展，HeuriGym [42]作为首批专为大语言模型评估而设计的综合性系统与编译器任务套件之一应运而生。人工智能驱动的研究系统（ADRS） [43], 从更广泛的角度探讨了由大语言模型驱动的科学工作流程，并强调大语言模型有望在各个系统领域实现复杂研究流程的自动化。SATLUTION [44] 等针对特定任务的求解器则将大语言模型应用于 SAT 求解。

在由大语言模型赋能的编译器与程序生成领域，先前的工作主要沿着两个方向展开。第一种做法是以大语言模型作为**代码生成器** 来取代编译器，根据高级描述直接生成目标代码或加速器内核 [12, 13, 45]。尽管这一方向前景可期，但通常仅限于相对较小的内核，而且绕过了现有编译器基础设施所提供的成熟正确性与兼容性保证。相比之下，我们旨在复用编译器流水线，以保留这些特性，并面向大型生产级应用。第二种方向与我们的目标更为互补：大语言模型不生成代码，而是提出优化**序列** 或转换方案，引导现有编译器取得更好的结果 [14]。这类方法通常将大语言模型的提议与蒙特卡洛树搜索 [15]等结构化搜索以及针对特定任务的微调 [16, 17] 相结合，以改进阶段排序和启发式选择。然而，这些方法通常需要针对每个输入应用程序生成序列，而我们的方法只需承担一次性开销，直接进化出可部署的编译器遍次，并可在多个程序间重复使用。

与生成式方法相辅相成，编译器基础设施中还有大量关于基于机器学习的启发式合成的研究。诸如 CompilerGym [19] 这样的强化学习环境，使智能体能够与真实编译器交互，根据反馈学习优化序列，从而避免在推理时针对每个程序进行生成。具体而言，在 LLVM 中，机器学习引导的优化（MLGO）设施 [18] 允许使用训练好的神经策略，替代内联或寄存器分配等决策中的手工启发式方法 [46], , 这表明学习得到的启发式方法无需显式程序合成，也能超越基线编译器的性能。

我们在 Magellan 上的工作与这些由机器学习驱动的编译器研究方向一致，但更强调**可维护且易于部署的启发式合成**，通过将高层策略生成与底层数值调优分离来实现这一点。Magellan 合成紧凑、可部署的编译器通道，能够在不同应用程序之间泛化；它结合了大语言模型的表达能力与自动调优器的效率，在搜索开销与真实世界编译器技术栈中的实际部署之间取得平衡。

5 讨论与未来工作

Magellan 的有效性源于将大语言模型驱动的创造力与针对策略模板和超参数的系统化进化搜索相结合。大语言模型提出一种 C++ 启发式模板，该模板使用 LLVM 的 API，并将其参数公开为编译器标志；与此同时，进化搜索以系统化的方式探索稀疏奖励景观。加入自动调优器来调优这些公开参数，进一步提升了采样效率。除了原始性能提升之外，一项关键成果是**生产力的大幅提升**：Magellan 能够快速生成与人工经过数十年开发的启发式方法相匹配、且往往超越它们的方案，并使用与人工开发者相同的宏基准测试来获得端到端奖励信号。它生成简洁、易读的启发式代码，可像人工编写的编译器代码一样交付和维护，从而大幅降低在生产环境中部署数据驱动的编译器优化的门槛。

尽管有这些优势，仍有一些挑战和未解问题亟待解决。当前搜索循环受到编译和评估每个候选方案所需开销的限制，尤其是对于需要评估有代表性的宏基准测试的性能目标。尚不清楚大语言模型驱动的遍演化（就策略发现而言）是否具备某些特性，使其能够与面临类似评估成本挑战的神经策略方法相竞争。了解这种替代方法在何种情况下更具优势，以及其效果如何随着搜索预算、模型选择和目标噪声而变化，是未来研究的重要方向。

展望未来，我们看到几个很有前景的方向。首先，通过更长的搜索迭代扩大评估规模，将有助于描述收敛行为，并明确当前方法的局限。其次，Magellan 可以与神经方法互为补充，通过自动发现特征来增强这些方法，例如与 IR2Vec 等学习嵌入方法结合使用 [47]。第三，应将 Magellan 应用于不同编译器基础设施和领域特定系统中的全新问题；近期面向 GPU 和 NPU 的编译器，例如用于内核映射、算子调度和任务分配的编译器，都依赖于适合通过自动发现来挖掘的启发式方法 [48–51]。这项研究可以回答的一个重要问题是：在缺乏既有文献和示例的情况下，Magellan 能否发现启发式方法。最后，我们希望 Magellan 发展成为一个通用平台，能够快速将编译器启发式方法适配到新的架构、优化任务和性能约束，从而实现协同设计工作流，让人类与人工智能顺畅协作，推动自动化性能工程的边界。

致谢

感谢 JacquesPienaar、William Moses、Jeremy Kun、Michael D. Moffitt、Sami Alabed 和 Kazu Hirata 为 LLVM 和 XLA 提供优化问题。我们也感谢 David Li、Tipp Moseley 和 Parthasarathy Ranganathan 对我们框架提出反馈，并感谢 Sagi Perel 在使用 Vizier 方面给予的支持。我们还要感谢 AlphaEvolve 团队以及 Google DeepMind 的扩展团队，是他们促成并支持了这一研究方向。

参考文献

[1] Alfred V. Aho、Monica S. Lam、Ravi Sethi 和 Jeffrey D. Ullman。《编译器：原理、技术与工具（第2版）》。美国：Addison-Wesley Longman Publishing Co., Inc.，2006年。[2] Fisher。轨迹调度：一种全局微码压缩技术。《IEEE计算机汇刊》，C-30(7):478–490，1981年。[3] G. J. Chaitin。通过图着色实现寄存器分配与溢出。载于*1982*年*SIGPLAN*编译器构造研讨会论文集，SIGPLAN '82，第98–105页，美国纽约，1982年。计算机协会。[4] M. Lam。软件流水线：一种适用于超长指令字机器的有效调度技术。载于*1988*年*ACM SIGPLAN*编程语言设计与实现会议论文集，PLDI '88，第318–328页，美国纽约，1988年。计算机协会。[5] David G. Bradlee、Susan J. Eggers 和 Robert R. Henry。为精简指令集计算机整合寄存器分配与指令调度。*SIGPLAN* 通讯，26(4):122–131，1991年4月。[6] P. P. Chang 和 W.-W. Hwu。用于编译C程序的内联函数展开。载于*1989*年*ACM SIGPLAN*编程语言设计与实现会议论文集，PLDI '89，第246–257页，美国纽约，1989年。计算机协会。[7] Bernardino Romera-Paredes、Mohammadamin Barekatain、Alexander Novikov、Matej Balog、M Pawan Kumar、E milien Dupont、Francisco JR Ruiz、Jordan S Ellenberg、Pengming Wang、Omar Fawzi 等。利用大语言模型通过程序搜索发现数学结果。《自然》，625(7995):468–475，2024年。[8] Haoran Ye、Jiarui Wang、Zhiguang Cao、Federico Berto、Chuanbo Hua、Haeyeon Kim、Jinkyoo Park 和 Guojie Song。Reevo：以大语言模型作为超启发式方法进行反思式进化。载于《神经信息处理系统进展》（*NeurIPS*），2024年。[9] Fei Liu、Xialiang Tong、Mingxuan Yuan、Xi Lin、Fu Luo、Zhenkun Wang、Zhichao Lu 和 Qingfu Zhang。启发式算法的演化：利用大语言模型实现高效自动算法设计。载于第*41*届国际机器学习会议（*ICML*）论文集，2024年。[10] Alexander Novikov、Ngân V˜u、Marvin Eisenberger、Emilien Dupont、Po-Sen Huang、Ada m Zsolt Wagner、Sergey Shirobokov、Borislav Kozlovskii、Francisco JR Ruiz、Abbas Mehrabian 等。AlphaEvolve：用于科学与算法发现的编程智能体。*arXiv*预印本 *arXiv:2506.13131*，2025年。[11] Robert Tjarko Lange、Yuki Imajuku 和 Edoardo Cetin。ShinkaEvolve：迈向开放式且样本高效的程序演化。*arXiv* 预印本 *arXiv:2509.19349*，2025年。[12] Robert Tjarko Lange、Qi Sun、Aaditya Prasad、Maxence Faldor、Yujin Tang 和 David Ha。迈向稳健的智能体CUDA内核基准测试、验证与优化。*arXiv*预印本 *arXiv:2509 14279*，2025年。。[13] Gang Liao、Hongsen Qin、Ying Wang、Alicia Golden、Michael Kuchnik、Yavuz Yetim、Jia Jiunn Ang、Chunli Fu、Yihan He、Samuel Hsia 等。KernelEvolve：扩展面向异构 Meta 的人工智能加速器。*arXiv* 预印本 *arXiv:2512.23236*，2025年。[14] Charles Hong、Sahil Bhatia、Alvin Cheung 和 Sophia Shao。Autocomp：大语言模型驱动的张量加速器代码优化。载于*2025*年计算机体系结构与系统机器学习会议，2025年。[15] Sujun Tang、Christopher Priebe、Rohan Mahapatra、Lianhui Qin 和 Hadi Esmaeilzadeh。通过大语言模型推理实现编译器优化，以提升模型服务效率。*arXiv* 预印本 *arXiv:2506.01374*，2025年。[16] Chris Cummins、Volker Seeker、Dejan Grubisic、Mostafa Elhoushi、Youwei Liang、Baptiste Roziere、Jonas Gehring、Fabian Gloeckle、Kim Hazelwood、Gabriel Synnaeve 等。用于编译器优化的大语言模型。*arXiv* 预印本 *arXiv:2309.07062*，2023年。[17] Chris Cummins、Volker Seeker、Dejan Grubisic、Baptiste Roziere、Jonas Gehring、Gabriel Synnaeve 和 Hugh Leather。Meta大语言模型编译器：编译器优化基础模型。*arXiv* 预印本 *arXiv:2407.02524*，2024年。[18] Mircea Trofin、Yundi Qian、Eugene Brevdo、Zinan Lin、Krzysztof Choromanski 和 David Li。MLGO：机器学习引导的编译器优化框架。*arXiv* 预印本 *arXiv:2101.04808*，2021年。[19] Chris Cummins、Bram Wasti、Jiadong Guo、Brandon Cui、Jason Ansel、Sahir Gomez、Somya Jain、Jia Liu、Olivier Teytaud、Benoit Steiner 等。CompilerGym：面向人工智能研究的稳健、高性能编译器优化环境。载于*2022*年*IEEE/ACM*国际代码生成与优化研讨会（*CGO*），第92–105页。IEEE，2022年。[20] S. VenkataKeerthy、Siddharth Jain、Umesh Kalvakuntla、Pranav Sai Gorantla、Rajiv Shailesh Chitale、Eugene Brevdo、Albert Cohen、Mircea Trofin 和 Ramakrishna Upadrasta。接下来的 700 种机器学习赋能的编译器优化。载于第*33*届*ACM SIGPLAN*国际编译器构造会议论文集，CC 2024，第238–249页，美国纽约州纽约市，2024年。计算机协会。[21] Qijing Huang、Ameer Haj-Ali、William Moses、John Xiang、Ion Stoica、Krste Asanovic 和 John Wawrzynek。AutoPhase：利用深度强化学习进行HLS编译器阶段排序。载于*2019*年*IEEE*第*27*届年度现场可编程定制计算机国际研讨会（*FCCM*），第308–308页。IEEE，2019年。[22] Hongzheng Chen 和 Minghua Shen。基于深度强化学习的FPGA高层次综合调度器。载于*2019*年*IEEE/ACM*国际计算机辅助设计会议（*ICCAD*），第1–8页。IEEE，2019年。[23] Theodoros Theodoridis、Tobias Grosser 和 Zhendong Su。理解并利用最优函数内联。载于第*27*届*ACM*编程语言与操作系统体系结构支持国际会议论文集，ASPLOS '22，第977–989页，美国纽约州纽约市，2022年。计算机协会。[24] Daniel Golovin、Benjamin Solnik、Subhodeep Moitra、Greg Kochanski、John Karro 和 David Sculley。Google Vizier：黑盒优化服务。载于第*23*届*ACM SIGKDD*知识发现与数据挖掘国际会议论文集，第1487–1495页，2017年。[25] Gheorghe Comanici、Eric Bieber、Mike Schaekermann、Ice Pasupat、Noveen Sachdeva、Inderjit Dhillon、Marcel Blistein、Ori Ram、Dan Zhang、Evan Rosen 等。Gemini 2.5：凭借高级推理、多模态、长上下文和新一代智能体能力，拓展前沿。*arXiv* 预印本 *arXiv:2507.06261*，2025年。[26] Rachel Potvin 和 Josh Levenberg。为什么谷歌将数十亿行代码存放在单一代码库中。《*ACM*通讯》，59 87，2016年 78：–。[27] Krzysztof Choromanski、Mark Rowland、Vikas Sindhwani、Richard E Turner 和 Adrian Weller。利用紧凑架构进行结构化演化，实现可扩展的策略优化。*arXiv* 预印本 *arXiv:1804 02395*，2018年。。[28] Teodor V. Marinov、Alekh Agarwal 和 Mircea Trofin。基于多个基线的离线模仿学习及其在编译器优化中的应用，2024年。

[29] Svilen Kanev、Juan Pablo Darago、Kim Hazelwood、Parthasarathy Ranganathan、Tipp Moseley、Gu-Yeon Wei 和 David Brooks。仓库级计算机的性能分析。载于第*42*届年度国际计算机体系结构研讨会论文集，第158–169页，2015年。[30] Teresa Johnson、Mehdi Amini 和 Xinliang David Li。ThinLTO：可扩展的增量式链接时优化。载于*2017*年国际代码生成与优化研讨会论文集，第111–121页，2017年。[31] 谷歌。Gemini 3，2025年。https://deepmind.google/models/gemini/。[32] Amir H Ashouri、Mostafa Elhoushi、Yuzhe Hua、Xiang Wang、Muhammad Asif Manzoor、Bryan Chan 和 Yaoqing Gao。MLGoPerf：用于优化性能的机器学习引导内联器。*arXiv* 预印本 *arXiv:2207.08389*，2022年。[33] Arya Vohra、Leo Seojun Lee、Jakub Bachurski、Oleksandr Zinenko、Phitchaya Mangpo Phothilimthana、Albert Cohen 和 William S. Moses。谨防抽象鸿沟：将等式饱和引入现实世界的机器学习编译器。《*ACM*编程语言学报》，9(OOPSLA2)，2025年10月。[34] William Moses 和 Valentin Churavy。与其为机器学习重写外部代码，不如自动合成快速梯度。《神经信息处理系统进展》，33 12485，2020年 12472：–。[35] Max Willsey、Chandrakana Nandi、Yisu Remy Wang、Oliver Flatt、Zachary Tatlock 和 Pavel Panchekha。egg：快速且可扩展的等式饱和。《*ACM*编程语言学报》，5(POPL)，2021年1月。[36] Jiaqi Yin、Zhan Song、Chen Chen、Yaohui Cai、Zhiru Zhang 和 Cunxi Yu。e-boost：结合自适应启发式方法与精确求解的增强型等式图提取。载于*2025*年*IEEE/ACM*国际计算机辅助设计会议（*ICCAD*），第1–9页，2025年。[37] Yaohui Cai、Kaixin Yang、Chenhui Deng、Cunxi Yu 和 Zhiru Zhang。Smoothe：可微等式图提取。载于第*30*届*ACM*编程语言与操作系统体系结构支持国际会议论文集，第*1*卷，ASPLOS '25，第1020–1034页，美国纽约州纽约市，2025年。计算机协会。[38] Hongzheng Chen、Cody Hao Yu、Shuai Zheng、Zhen Zhang、Zhiru Zhang 和 Yida Wang。Slapo：用于渐进式优化大型深度学习模型训练的调度语言。载于第*29*届*ACM*编程语言与操作系统体系结构支持国际会议论文集，第*2*卷，第1095–1111页，2024年。[39] Sami Alabed、Daniel Belov、Bart Chrzaszcz、Juliana Franco、Dominik Grewe、Dougal Maclaurin、James Molloy、Tom Natan、Tamara Norman、Xiaoyue Pan、Adam Paszke、Norman A. Rink、Michael Schaarschmidt、Timur Sitdikov、Agnieszka Swietlik、Dimitrios Vytiniotis 和 Joel Wee。Partir：组合机器学习的SPMD分区策略。载于第*30*届*ACM*编程语言与操作系统体系结构支持国际会议论文集，第*1*卷，ASPLOS '25，第794–810页，美国纽约州纽约市，2025年。计算机协会。[40] Lianmin Zheng、Zhuohan Li、Hao Zhang、Yonghao Zhuang、Zhifeng Chen、Yanping Huang、Yida Wang、Yuanzhong Xu、Danyang Zhuo、Eric P Xing 等。Alpa：自动化分布式深度学习的算子间与算子内并行。载于第*16*届*USENIX*操作系统设计与实现研讨会（*OSDI 22*），第559–578页，2022年。[41] Michael D. Moffitt 和 Pratik Fegade。ASPLOS 2025 / EuroSys 2025分布式深度学习算子内并行竞赛。载于第*30*届*ACM*编程语言与操作系统体系结构支持国际会议论文集，第3卷，*ASPLOS 2025*，第5–17页，2025年。[42] Hongzheng Chen、Yingheng Wang、Yaohui Cai、Hins Hu、Jiajie Li、Shirley Huang、Chenhui Deng、Rongjian Liang、Shufeng Kong、Haoxing Ren 等。HeuriGym：用于组合优化中大语言模型生成启发式方法的智能体基准测试。*arXiv* 预印本 *arXiv:2506.07972*，2025年.[43] Audrey Cheng、Shu Liu、Melissa Pan、Zhifei Li、Bowen Wang、Alex Krentsel、Tian Xia、Mert Cemri、Jongseok Park、Shuo Yang 等。门口的野蛮人：人工智能如何颠覆系统研究。*arXiv* 预印本 *arXiv:2510.06189*，2025年。 [44] Cunxi Yu、Rongjian Liang、Chia-Tung Ho 与 Haoxing Ren。自主代码演化遇上 NP 完全性。*arXiv* 预印本 *arXiv:2509.07367*，2025年。[45] Alec M Hammond、Aram Markosyan、Aman Dontula、Simon Mahns、Zacharias Fisches、Dmitrii Pedchenko、Keyur Muzumdar、Natacha Supper、Mark Saroufim、Joe Isaacson 等。面向机器学习专用集成电路的智能体算子生成。*arXiv* 预印本 *arXiv:2512.10977*，2025年。 [46] Yundi Qian 与 Mircea Trofin。MLGO：面向编译器优化的机器学习框架。*Google Research* 博客，2022年7月。[47] S VenkataKeerthy、Rohit Aggarwal、Shalini Jain、Maunendra Sankar Desarkar、Ramakrishna Upadrasta 与 YN Srikant。IR2Vec：基于 LLVM 中间表示的可扩展程序嵌入。*ACM*《体系结构与代码优化汇刊》（*TACO*），17(4):1–27，2020年。[48] Shihan Fang、Hongzheng Chen、Niansong Zhang、Jiajie Li、Han Meng、Adrian Liu 与 Zhiru Zhang。Dato：面向数据流加速器的基于任务的编程模型。*arXiv* 预印本 *arXiv:2509.06794*，2025年。[49] Hongzheng Chen、Bin Fan、Alexander Collins、Bastian Hagedorn、Evghenii Gaburov、Masahiro Masuda、Matthew Brookhart、Chris Sullivan、Jason Knight、Zhiru Zhang 等。Tawa：面向现代 GPU 的基于异步引用的自动线程束专门化。*arXiv* 预印本 *arXiv:2510.14719*，2025年。[50] Rupanshu Soi、Rohan Yadav、Fredrik Kjolstad、Alex Aiken、Maryam Mehri Dehnavi、Michael Garland 与 Michael Bauer。面向张量核心图形处理器的最优软件流水线与线程束专门化。*arXiv* 预印本 *arXiv:2512.18134*，2025年。[51] Hongzheng Chen、Niansong Zhang、Shaojie Xiang、Zhichen Zeng、Mengjia Dai 与 Zhiru Zhang。Allo：面向可组合加速器设计的编程模型。*ACM*《编程语言汇刊》，8(PLDI):593–620，2024年。

用于二进制大小的 Magellan 生成的内联策略

该 f ollowing代码片段展示了 Magellan 生成的用于二进制大小 reductio 的内联策略 n.

```
// AEInlineAdvisor.cpp #include "llvm/Analysis/AEInlineAdvisor.h" #include "
llvm/Analysis/InlineAdvisor.h" #include "llvm/Analysis/OptimizationRemarkEmitter.h" #include "
llvm/IR/PassManager.h" // EVOLVE-BLOCK-START // 你可以在此处包含其他 LLVM 头文件。 using
namespace llvm; // 定义内联阈值常量，以提高可读性和可维护性。 // 这些值属于启发式方法,
可根据性能分析数据进行调整。 // 非常小的函数通常适合内联，以消除调用开销。 constexpr
unsigned TINY_FUNCTION_THRESHOLD = 10; // 常规函数的基础指令数阈值。 constexpr
unsigned SMALL_FUNCTION_THRESHOLD = 25; // 适用于只有一个调用点的函数的额外指令数预算,
// 因为内联这类函数可避免代码重复并促成更多优化。 constexpr unsigned SINGLE_USE_INLINE_
BONUS = 80; // 大幅提高阈值 // 将函数视为具有“许多”基本块并施加惩罚的阈值。 constexpr
unsigned MANY_BASIC_BLOCKS_THRESHOLD = 5; // 对具有许多基本块的函数施加的惩罚。
constexpr unsigned BASIC_BLOCK_PENALTY = 5; std::unique_ptr<InlineAdvice>
AEInlineAdvisor::getAdviceImpl(CallBase &CB) { // 内联策略的实现。不要更改函数接口。 // 默认不
进行内联。 constexpr unsigned CONSERVATIVE_INLINE_PENALTY = 20; constexpr unsigned
HOT_FUNCTION_BONUS = 50; bool IsInliningRecommended =false; Function *Callee =
CB.getCalledFunction(); // 为不同类型的指令定义权重，以计算“复杂度评分”， // 而非直接使用指令
数。 constexpr unsigned WEIGHT_HIGH_COMPLEXITY_INST = 3; constexpr unsigned
WEIGHT_MEDIUM_COMPLEXITY_INST = 2; constexpr unsigned WEIGHT_LOW_COMPLEXITY_
INST = 1; // 如果被调用函数为空（间接调用）或为声明（没有可供内联的函数体）， // 则无法内联。
if (!Callee Callee->isDeclaration()) { // 如果被调用函数为空（间接调用）或为声明（没有可供内联的函
数体）， // 则无法内联。返回不内联的建议。 IsInliningRecommended =false; return
std::make_unique<InlineAdvice>( this, CB, FAM.getResult<
OptimizationRemarkEmitterAnalysis>(*CB.getCaller()), IsInliningRecommended); } // 首先
检查显式内联属性，因为它们优先于启发式方法。 if (Callee->hasFnAttribute(Attribute::NoInline)) {
// 显式禁止内联。IsInliningRecommended 仍为 false。 // 继续执行，直到最终返回。 } else if
(Callee->hasFnAttribute(Attribute::AlwaysInline)) { IsInliningRecommended =true; } else { //
大胆设想：根据指令复杂度计算加权指令数。 // 这样可以更细致地估算内联决策中的“大小”。
unsigned WeightedCalleeInstructionCount = 0; for (const BasicBlock &BB : *Callee) { for (const
Instruction &I : BB) { switch (I.getOpcode()) { case Instruction::Call: case Instruction::Invoke:
case Instruction::CallBr: case Instruction::Ret: case Instruction::Br: case
Instruction::Switch: case Instruction::IndirectBr: WeightedCalleeInstructionCount +=
WEIGHT_HIGH_COMPLEXITY_INST; break; case Instruction::Load: case Instruction::Store:
case Instruction::Alloca: case Instruction::GetElementPtr: case I
nstruction::AtomicCmpXchg: case Instruction::AtomicRMW:
```

```
        case Instruction::Fence: case Instruction::FNeg: case Instruction::FAdd: case Instruction::FSub: case Instruction::FMul: case Instruction::FDiv: case Instruction::FRem: case Instruction::FCmp: case Instruction::SDiv: case Instruction::UDiv: case Instruction::SRem: case Instruction::URem: case Instruction::InsertElement: case Instruction::ExtractElement: case Instruction::ShuffleVector: case Instruction::ExtractValue: case Instruction::InsertValue: WeightedCalleeInstructionCount += WEIGHT_MEDIUM_COMPLEXITY_INST; break; default: WeightedCalleeInstructionCount += WEIGHT_LOW_COMPLEXITY_INST; break; } } } // 极小函数几乎总是适合内联，因为这样可以 // 消除调用开销，不受其他因素影响。这构成了 // 一个绝对的最低阈值，可覆盖其他惩罚， // 确保足够小的函数即使经过大小优化也会被纳入考虑。 // 对此绝对阈值使用原始指令数，因为对于极小函数而言，这种方式更简单， // 且较少受到具体指令组合的影响。 if (Callee->getInstructionCount() < TINY_FUNCTION_THRESHOLD) { IsInliningRecommended = true; } else { // 大胆设想：对纯函数、只读函数或仅有一个使用点且规模仍较小的函数， // 采用更积极的内联策略。这通常能带来显著简化，并在内联后 // 消除死代码，因为它们可以缩小整体代码大小或促成更多优化。 constexpr unsigned AGGRESSIVE_SPECIAL_CASE_THRESHOLD = 150; // 对只读函数采用比纯函数稍保守的阈值， // 因为它们仍可能存在复杂的交互。 constexpr unsigned AGGRESSIVE_READONLY_THRESHOLD = 75; if ((Callee->hasOneUse() && Callee->getInstructionCount() <AGGRESSIVE_SPECIAL_CASE_THRESHOLD) (Callee->getMemoryEffects().doesNotAccessMemory() && WeightedCalleeInstructionCount < AGGRESSIVE_SPECIAL_CASE_THRESHOLD) (Callee->getMemoryEffects().onlyReadsMemory() && WeightedCalleeInstructionCount < AGGRESSIVE_READONLY_THRESHOLD)) { IsInliningRecommended =true; } else { // 对其他情况采用基于启发式的内联策略： unsigned CurrentThreshold =SMALL_FUNCTION_THRESHOLD; // 根据通用函数优化属性调整基础阈值。 // 使用 if-else if 结构优先处理这些标志，避免 // 出现相互冲突的调整（例如同时具有 OptimizeForSize 和 Hot 属性）。 if (Callee->hasFnAttribute(Attribute::OptimizeForSize) Callee->hasFnAttribute(Attribute::MinSize)) { // 对经过大小优化的函数，采用更保守的初始阈值。 CurrentThreshold =std::max(0U, CurrentThreshold - CONSERVATIVE_INLINE_PENALTY); } else if (Callee->hasFnAttribute(Attribute::Hot)) { // 对热点函数，允许更积极地内联。 CurrentThreshold +=HOT_FUNCTION_BONUS; } // 只有一个调用点的函数是很好的内联候选，因为内联 // 可以避免代码重复并促成更多优化。 // 如果单次使用函数尚未被前面更积极的特殊情况内联策略覆盖， // 则应用此奖励。 if (Callee->hasOneUse()) { CurrentThreshold +=SINGLE_USE_INLINE_BONUS; } // 对具有许多基本块的函数施加惩罚，因为这表明其控制流较复杂。 // 这样可以避免被调用函数包含许多小块时，调用方的基本块数量 // 膨胀过多。 if (Callee->size() > MANY_BASIC_BLOCKS_THRESHOLD) { CurrentThreshold =std::max(0U, CurrentThreshold - BASIC_BLOCK_PENALTY); } // 大胆设想的增强：对作为常量或未定义值的参数给予奖励。 // 这表明内联后可能实现显著简化。 constexpr unsigned CONSTANT_ARG_BONUS_PER_ARG = 10; constexpr unsigned UNDEF_ARG_BONUS_PER_ARG = 5;
```

```
        // 疯狂创意增强：根据参数类型匹配情况增加奖励或惩罚。
        constexpr unsigned精确_类型_匹配_奖励_每个_参数 = 7;
        constexpr unsigned指针_可转换_类型_匹配_奖励_每个_参数 = 3;
        constexpr unsigned非_平凡_转换_惩罚_每个_参数 = 5;

        // 获取用于指针类型比较的数据布局。
        const数据布局 &DL  = 被调用函数->getParent()->getDataLayout();

        for(无符号 I = 0,     E = CB.arg_size(); I != E; ++I) {
          值 *参数 =CB.getArgOperand(I);
          Type*参数类型 = Arg->getType();

          // 首先检查常量/未定义值参数。
          如果 (isa<常量>(Arg)){
            当前阈值 += 常量_参数_奖励_每个_参数;
          }否则如果 (isa<未定义值>(参数)) {
            当前阈值 += 未定义值_参数_奖励_每个_参数;
          }

          // 检查参数类型是否与被调用函数的形式参数类型匹配。
          如果 (I < 被调用函数->获取函数类型()->获取参数数量()){
            类型*形参类型 = 被调用函数->获取函数类型()->获取参数类型(I);

            如果 (参数类型 == 形参类型) {
              当前阈值 += 精确_类型_匹配_奖励_每_参数;
            }否则如果 ((参数类型->是指针类型() && 形参类型->是整数类型())
                       (参数类型->是整数类型()&& 形参类型->是指针类型())){
              // 检查无操作指针/整数转换。
              如果 (类型转换指令::是位或无操作指针可转换(参数类型,   形参类型, DL)) {
                当前阈值 += 指针_可转换_类型_匹配_奖励_每个_参数;
              } else {
                当前阈值 = std::max(0U,            当前阈值 - 非_平凡_转换_惩罚_每个_参数);
              }
            }否则如果 (!参数类型->是否为 void 类型()&& 参数类型!=形参类型) {
              // 对于其他类型不匹配的情况（例如,     不同的整数位宽,              浮点数转整数）,
              // 如果所需的类型转换会改变位模式，则施加惩罚。
              // 我们需要推断类型转换操作码，以检查它是空操作位转换还是转换。
              指令::CastOps 操作码  =指令::BitCast;// 默认为 BitCast
              如果 (参数类型->isIntegerTy()&& 形参类型->isIntegerTy()){
                操作码 = 类型转换指令::getCastOpcode(Arg,   /*SrcIsSigned=*/true, 形参类型, /*DstIsSigned=*/true);
              } 否则如果 (参数类型->isFloatingPointTy()&&形参类型->isFloatingPointTy()){
                操作码 = 类型转换指令::getCastOpcode(Arg,   /*源为有符号数=*/false,   形参类型, /*目标是否带符号=*/false);
              }

              如果 (!类型转换指令::isNoopCast(Opcode,   参数类型, 形参类型, DL)) {
                当前阈值 = std::max(0U,                当前阈值 - NON_TRIVIAL_CAST_PENALTY_PER_ARG);
              }
            }
          }
        }

        是否建议内联  =加权被调用函数指令数 <当前阈值;
      }
    }
  }

  // 所有情况的单一返回点。
  返回 std::make_unique<内联建议>(
      this, CB,
      FAM.getResult<优化备注发射器分析>(*CB.getCaller()),
      是否建议内联);
}

// EVOLVE-BLOCK-END
```

B Magellan 生成的性能内联策略

以下代码片段展示了 Magellan 生成的、用于提升性能的内联策略。

```
#include   "llvm/Analysis/AEInlineAdvisor.h"
#include "llvm/Analysis/LoopInfo.h"
#include "llvm/Analysis/BlockFrequencyInfo.h"
#include   "llvm/Analysis/InlineAdvisor.h"
#include "llvm/Analysis/优化备注发射器.h"
#include "llvm/Analysis/目标库信息.h"
#include "llvm/IR/遍历管理器.h"

using namespace llvm;

// 这是由自动调优器获得的最佳参数配置：
// 「名称：  "ae-inline-base-threshold"
```

```
//int_value: 200 // , name: "ae-inline-call-penalty" //int_value: 13 // ,
name: "ae-inline-const-arg-bonus" //int_value: 68 // , name: "
ae-inline-hotness-mul" //int_value: 10 // , name: "ae-inline-hotness-shift" //
int_value: 3 // , name: "ae-inline-large-caller-reduction" //int_value: 21 // ,
name: "ae-inline-large-caller-threshold" //int_value: 9207 // , name: "
ae-inline-loop-bonus" //int_value: 12 // , name: "
ae-inline-recursion-penalty" //int_value: 22 // , name: "
ae-inline-vector-bonus" //int_value: 97 //  ]// EVOLVE-BLOCK-START //此处
可以包含额外的 LLVM 头文件、常量和标志。 #include "
llvm/IR/Instructions.h" #include "llvm/IR/InstIterator.h" // [
hyperparam]: ae-inline-base-threshold, int, 10, 200 staticcl::opt<int>
BaseThreshold( "ae-inline-base-threshold", cl::desc("内联的基础指令数阈值
"), cl::init(60), cl::Hidden); // [hyperparam]: ae-inline-call-penalty, int, 5,
50 staticcl::opt<int> CallPenalty( "ae-inline-call-penalty", cl::desc("被调用
函数中调用指令的惩罚"), cl::init(15), cl::Hidden); // [hyperparam]:
ae-inline-const-arg-bonus, int, 0, 100 staticcl::opt<int>
ConstantArgBonus( "ae-inline-const-arg-bonus", cl::desc("每个常量参数的
奖励"), cl::init(40), cl::Hidden); // [hyperparam]: ae-inline-loop-bonus,
int, 0, 100 staticcl::opt<int> LoopBonus( "ae-inline-loop-bonus", cl::desc("
循环内调用点的奖励"), cl::init(40), cl::Hidden); // [hyperparam]:
ae-inline-vector-bonus, int, 0, 100 staticcl::opt<int> VectorBonus( "
ae-inline-vector-bonus", cl::desc("被调用函数包含向量指令时的奖励"),
cl::init(30), cl::Hidden); // [hyperparam]: ae-inline-hotness-mul, int, 1,
10 staticcl::opt<int> HotnessMultiplier( "ae-inline-hotness-mul", cl::desc("
热点调用点的乘数"), cl::init(3), cl::Hidden); // [hyperparam]:
ae-inline-hotness-shift, int, 0, 15 staticcl::opt<int> HotnessShift( "
ae-inline-hotness-shift", cl::desc("用于确定热点度的移位量"), cl::init(8),
cl::Hidden); // [hyperparam]: ae-inline-recursion-penalty, int, 0, 100
staticcl::opt<int> RecursionPenalty( "ae-inline-recursion-penalty",
cl::desc("递归调用的惩罚"), cl::init(50), cl::Hidden); // [hyperparam]:
ae-inline-large-caller-threshold, int, 1000, 10000 staticcl::opt<int>
LargeCallerThreshold( "ae-inline-large-caller-threshold", cl::desc("用于惩
罚的调用者大小阈值"), cl::init(4000), cl::Hidden); // [hyperparam]:
ae-inline-large-caller-reduction, int, 0, 90 staticcl::opt<int>
LargeCallerReduction( "ae-inline-large-caller-reduction",
```

96 cl::desc("大型调用方阈值的百分比降幅"), 97 cl::init(20), cl::Hidden); 98 99 // 性能优化的内联策略实现。请勿更改 100 // 函数接口。默认实现是不进行内联。 101 // 请确保使用 BlockFrequencyInfo 和 TargetLibraryInfo， 102 // 因为我们针对性能进行优化。 103 std::unique_ptr<InlineAdvice>AEInlineAdvisor::getAEAdviceImpl(CallBase &CB) { 104 Function *Caller = CB.getCaller(); 105 Function *Callee = CB.getCalledFunction(); 106 107 if (!Callee Callee->isDeclaration()) { 108 return std::make_unique<InlineAdvice>(109 this, CB, FAM.getResult<OptimizationRemarkEmitterAnalysis>(*Caller), 110 false); 111 } 112 113 if (Callee->hasFnAttribute(Attribute::AlwaysInline)) { 114 return std::make_ unique<InlineAdvice>(115 this, CB, FAM.getResult<Opti mizationRemarkEmitterAnalysis>(*Caller), 116 true); 117 } 118 119 const TargetLibraryInfo &TLI = FAM.getResult<TargetLibraryAnalysis>(*Callee); 120 LibFunc LibFn; 121 if (TLI.getLibFunc(*Callee, LibFn) && TLI.hasOptimizedCodeGen(LibFn)) { 122 return std::make_unique<InlineAdvice>(123 this, CB, FAM.getResult< OptimizationRemarkEmitterAnalysis>(*Caller), 124 true); 125 } 126 127 int Cost = 0; 128 bool HasVector = false; 129 130 for (const auto &I : instructions(Callee)) { 131 if (isa< AllocaInst>(I) isa<PHINode>(I) isa<BitCastInst>(I) 132 isa<PtrToIntInst>(I) isa<IntToPtrInst> (I)) { 133 continue; 134 } 135 136 if (const auto *CI = dyn_cast<CallBase>(&I)) { 137 Cost += CallPenalty; 138 if (CI->isIndirectCall()) { 139 Cost +=CallPenalty; 140 } 141 } else { 142 Cost += 1; 143 } 144 145 if (!HasVector && I.getType()->isVectorTy()) { 146 HasVector = true; 147 } 148 } 149 150 int Threshold = BaseThreshold; 151 152 for (const auto &Arg : CB.args()) { 153 if (isa<Constant>(Arg)) { 154 Threshold += ConstantArgBonus; 155 } 156 } 157 158 LoopInfo &LI = FAM.getResult<LoopAnalysis>(*Caller); 159 int Depth = LI.getLoopDepth(CB.getParent()); 160 if (Depth > 0) { 161 Threshold +=LoopBonus * std::min(Depth, 3); 162 } 163 164 if (HasVector) { 165 Threshold +=VectorBonus; 166 } 167 168 BlockFrequencyInfo &BFI = FAM.getResult<BlockFrequencyAnalysis>(*Caller); 169 uint64_t EntryFreq =BFI.getEntryFreq().getFrequency(); 170 if (EntryFreq > 0) { 171 uint64_t CallSiteFreq =BFI.getBlockFreq(CB.getParent()).getFrequency(); 172 if (CallSiteFreq > (EntryFreq >> HotnessShift)) { 173 Threshold *= HotnessMultiplier; 174 } 175 } 176 177 if (Caller == Callee) { 178 Threshold -=RecursionPenalty;

```
  }

  整数调用方大小 = 0;
  对于(常量自动 &BB : *调用方) {
    调用方大小 += BB.size();
    如果 (调用方大小 >大型调用方阈值)跳出;
  }

  如果 (调用方大小 > 大型调用方阈值) {
    阈值 =阈值 * (100 - 大型调用方缩减量) / 100;
  }

  布尔值是否建议内联 =开销 < 阈值;

  返回 std::make_unique<内联建议>(
      此对象, CB, FAM.getResult<优化备注发射器分析>(*调用方),
      是否建议内联);
}

// EVOLVE-BLOCK-END

std::unique_ptr<内联建议> AE 内联顾问::获取建议实现(CallBase &CB) {
  // 合法性检查
  auto &调用方 = *CB.getCaller();
  auto &被调用函数 = *CB.getCalledFunction();
  auto &ORE = FAM.getResult<优化备注发射器分析>(Caller);

  自动强制种类 =内联顾问::获取强制种类(CB, FAM, ORE);
  // 如果这是 "永不内联" 这种情况, 内部不会有任何更改
  // 我们需要跟踪的状态, 因此可以直接返回基础的内联建议, 它
  // 不会执行任何有意义的操作。
  // 如果这是递归情形, 也同样如此。
  如果(强制种类 ==内联顾问::强制内联类型::Never
      &调用方 ==&被调用函数)
    返回 getMandatoryAdvice(CB, 假);

  自动 是否可行 = 内联可行性(被调用函数);
  如果 (!是否可行.成功())
    返回 标准::创建_唯一<内联建议>(此, CB, ORE, 假);

  布尔 强制 =
      强制种类 == 内联顾问::强制内联类型::始终;

  如果 (强制)
    返回 std::make_unique<内联建议>(此, CB, ORE, 真);

  返回 getAEAdviceImpl(CB);
}
```