合肥工業大學
HEFEI UNIVERSITY OF TECHNOLOGY

全国大学生系统能力大赛编译系统设计赛

自动混精编译系统 AMP-2025

队名 O4

队长 牟长青

队员 邓杰涌、胡苏桓、焦超然

学校 合肥工业大学

教师 刘彬彬、李宏芒

完成时间 2025 年 8 月 20 日

2025 年 8 月 23 日

目录

1 绪论

高性能计算程序中往往有大量的浮点数，浮点数对程序运行的结果与性能至关重要。大部分程序员缺乏专业的数值分析背景，所以往往为了结果正确而盲目使用高精度浮点数，这对程序的性能有较大副作用，尤其是在对性能敏感的极端环境下盲目使用高精度浮点数是不可取的。

针对这一问题，我们提出了 AMP-2025 混精编译优化系统。该系统属于动态变量级混精优化方法，通过多层次搜索架构，保证了解的有效性，通过机器学习、循环层次语义分组等方法有效提高了系统的收敛性。该系统在华为鲲鹏 920ARM 服务器上，针对 HPL-AI 基准测试程序在短时间内搜索出了有效解，对基准测试程序进行了混精优化，性能获得了大幅度提升。

2 背景知识

2.1 xgboost

XGBoost（eXtreme Gradient Boosting）是一种**梯度提升决策树算法**，其核心思想是：

- **集成学习**：通过组合多个弱学习器（决策树）构建强学习器
- **梯度提升**：每次训练新树时，都针对前一轮预测的残差进行优化
- **正则化**：通过 L1/L2 正则化防止过拟合

2.1.1 目标函数优化

XGBoost 基于梯度提升决策树，其核心思想是**加法模型**：

$$F(x) = \sum_{k=1\to K}^{n} f_k(x)$$

- $F(x)$ 是最终预测
- $f_k(x)$ 是第 k 棵决策树
- K 是树的总数

XGBoost 的目标函数包含损失函数和正则化项

$$Obj = \sum_{i=1\to n} l(y_i, y_i^{\cdot}) + \sum_{i=1\to K} (f_k)$$

- $l(y_i, y_i^{\cdot})$ 是损失函数（如 MSE）
- (f_k) 是正则化项，控制模型复杂度

正则化项的设立可以防止过拟合，通过在目标函数中添加惩罚项来约束模型复杂度，主要有以下几种正则化项：

$$\Omega(f) = \alpha * \sum |w_j| \quad L1$$
$$\Omega(f) = \lambda * \sum (w_j)^2 \quad L2$$

XGBoost 使用泰勒展开来近似目标函数：

$$Obj \approx \sum_{i=1\to n} [g_i * f_t(x_i) + 0.5 * h_i * f_t(x_i)^2] + \Omega(f_t)$$

2.1.2 特征工程引擎

特征工程方法是提高代理模型的预测精度、更加智能化的评估配置的关键组件
首先需要获取原始特征空间

$$\phi(x) = [\phi_1(x), \phi_2(x), ..., \phi_d(x)]^T$$

然后需要使用 StandardScaler 将现有特征标准化，获取叶子节点特征

$$\phi_{std}(x) = \frac{\phi(x) - \mu}{\sigma}$$

接着进行特征重要性计算，基于基尼不纯度（越小越好）:

$$I_j = \frac{1}{M} \sum_{m=1}^{M} \sum_{t \in T_m} \Delta Gini(t, j)$$

精度交互特征，定义精度交互函数:

$$\text{precision}(x) = \phi_1(x) \times \phi_2(x)$$
$$\text{precision}(x) = |\phi_1(x) - \phi_2(x)|$$

2.1.3 置信度评估

训练的时候会记录两类量：训练集 RMSE 和增强特征的标准差向量。预测时，在增强特征空间对样本做高斯微扰的 Monte Carlo 采样，观测预测的方差作为不确定度。
然后将“预测标准差”与训练 RMSE 做尺度归一并映射为置信度。

2.2 遗传算法 GA

2.2.1 混合精度问题的进化操作建模

进化引擎将混合精度优化问题建模为多基因位遗传算法，每个变量对应一个基因位，每个基因位有 3 个等位基因（double/float/half）:

EvolutionEngine

```
class EvolutionEngine:
    def __init__(self, mutation_rate=0.3, crossover_rate=0.7):
        # 标量类型: double, float, half
        self.scalar_types = ["double", "float", "half"]
        # 指针类型: double*, float*, half*
        self.pointer_types = ["double*", "float*", "half*"]
```

基因编码原理:

- **基因位**：每个变量对应一个基因位

- **等位基因**：每个基因位有 3 个精度选择（double/float/half）
- **适应度**：基于 HPL-AI 基准测试的性能表现

2.2.2 锦标赛选择机制

锦标赛选择通过随机采样比较的方式选择优秀个体，避免过早收敛，如图2。

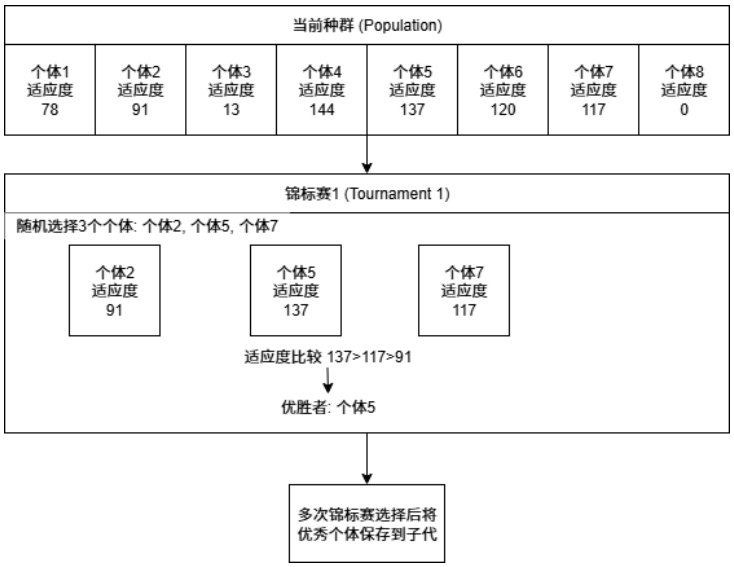

图 2 锦标赛搜索

2.2.3 单点交叉操作

交叉操作通过基因位级别的交换组合优秀个体的特征：

crossover

```python
def crossover(self, parent1, parent2):
    child1 = copy.deepcopy(parent1)
    child2 = copy.deepcopy(parent2)

    # 对每个变量进行交叉
    for i, var in enumerate(child1.get("localVar", [])):
        if random.random() < 0.5:
            # 交换类型
            child1["localVar"][i]["type"] = parent2["localVar"][i]["type"]
            child2["localVar"][i]["type"] = parent1["localVar"][i]["type"]

    return child1, child2
```

交叉操作原理：

- **基因位交换**：每个变量有一定概率与其他个体交换精度类型

- **独立交换**：每个基因位独立进行交换决策
- **子代生成**：产生两个新的子代个体

2.2.4 位变异操作

变异操作通过随机改变基因位维持种群多样性，避免局部最优：

- **位变异**：每个基因位有 10% 概率发生变异
- **类型约束**：指针类型和标量类型分别变异
- **随机选择**：变异时随机选择新的精度类型

crossover

```
    def mutate(self, individual):
        mutated = copy.deepcopy(individual)
        for var in mutated.get("localVar", []):
            if random.random() < 0.1:  # 10%变异概率
                if var["type"].endswith("*"):
                    var["type"] = random.choice(self.pointer_types)
                else:
                    var["type"] = random.choice(self.scalar_types)

        return mutated
```

2.2.5 精英保留策略

精英保留策略确保**最优个体不会在进化过程中丢失**：

- **最优保护**：保留种群中适应度最好的 20% 个体
- **直接复制**：精英个体直接复制到下一代
- **质量保证**：确保最优解不会在进化过程中丢失

2.2.6 种群进化流程

进化引擎通过选择-交叉-变异-精英保留的完整流程实现种群进化：
进化流程：

1. **精英选择**：保留最优的 20% 个体
2. **父代选择**：通过锦标赛选择选择父代
3. **交叉操作**：父代交叉产生子代

4. **变异操作**：子代变异增加多样性

5. **种群更新**：精英个体与新个体组成新种群

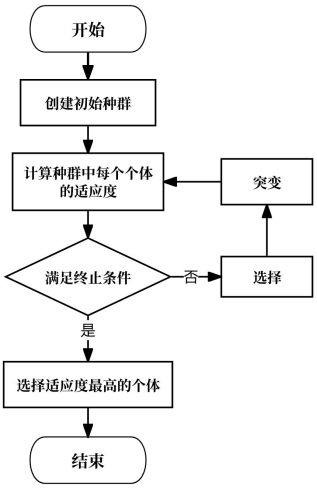

图 3 GA

2.3 模拟退火 SA

2.3.1 原理

模拟退火（Simulated Annealing, 简称 SA）是一种常用的全局优化算法，主要用于在大规模、复杂的搜索空间中寻找最优解。能以一个基于当前温度 T 概率接受较差解跳出局部最优，提升个体质量，但全局搜索能力有限，如图4。

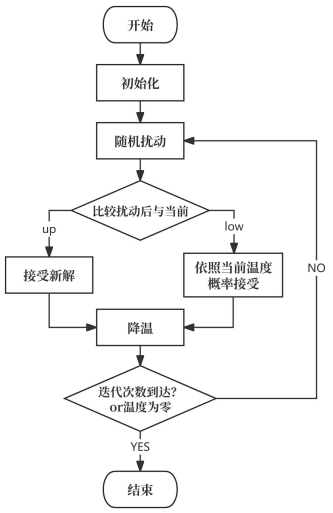

图 4 SA

2.3.2 算法主流程

1. **初始化**：随机生成一个初始解，并设定初始温度 T。
2. **扰动**：在当前解的邻域内随机产生一个新解。
3. **接受准则**：如果新解比当前解更优（能量更低），则接受新解；如果新解更差，以一定概率接受新解。
4. **降温**：按照某种规则逐步降低温度 T（ $T = \alpha$,$T_0 < \alpha < 1$）。
5. **终止条件**：温度降到某个阈值或达到最大迭代次数时停止。

3 设计与实现

3.1 总体架构设计

3.1.1 配置解析模块

该模块用于解析精度配置文件，为降精模块准备降精信息。

3.1.2 降精模块

该模块根据降精信息，降低指定变量的精度。

3.1.3 搜索系统

首先根据循环层次分组算法给出分组配置，由 GA 驱动进行粗糙搜索，最后将搜索结果传给精细搜索，作为其热启动配置。精细搜索中，经过 GA 的全局搜索之后，针对种群中的最优个体再进行 SA 局部调优, 借此加快收敛速度和跳出局部最优。

其中，性能预测器与缓存优化的目的是为了加快收敛速度。

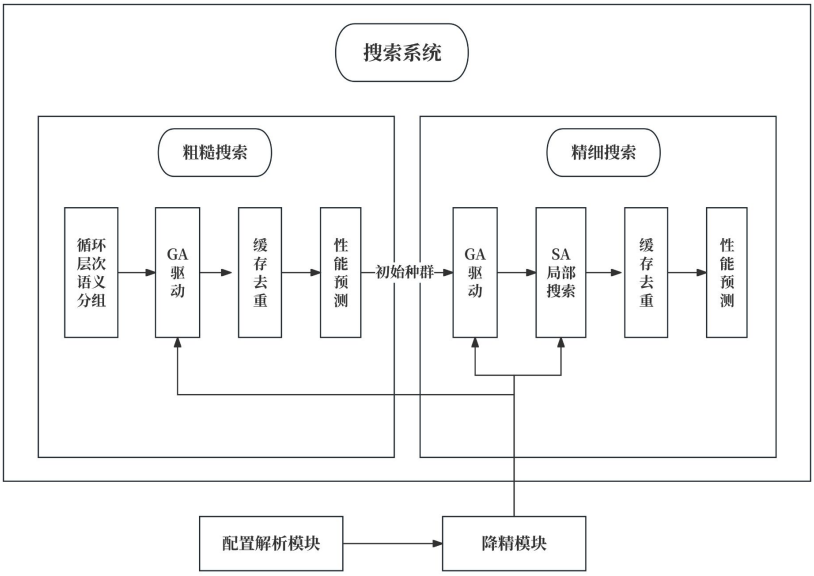

图 5 总体架构

3.2 clang 前端 Pass 的设计与实现

本插件的目的是识别比赛中 #pragma AutoMxPrec 标记的代码块，并输出其中的函数调用与局部变量信息。插件会自动定位这些标记区域，在其中提取函数调用和变量声明，并将分析结果组织成 JSON 文件，便于后续使用。代码采用 Clang 插件化实现，无需修改编译器源码，同时具备良好的扩展性，可支持更多类型的分析需求。

3.2.1 系统架构

插件的系统架构主要围绕指定的 #pragma AutoMxPrec 标记展开, 插件的任务是识别这些区域并提取其中的语义信息。插件主要包括四个组成部分:

1. Pragma 捕获模块: 通过自定义 PragmaHandler, 记录 #pragma AutoMxPrec 的源码位置。
2. AST 遍历模块: 基于 RecursiveASTVisitor 遍历 Clang AST, 在标记区域中收集函数调用和变量声明。
3. 数据存储与输出模块: 将收集的信息保存为 JSON 数据, 输出到指定文件。
4. 插件入口模块: 完成插件注册、命令行参数解析和 AST 消费器的创建。

3.2.2 关键实现

AutoMxPrecPragmaHandler 继承自 PragmaHandler,在编译阶段捕获源码中的 #pragma AutoMxPrec 指令, 并将其在源文件中的位置保存到全局数组 GlobalPragmaLocs 中, 为后续 AST 遍历提供标记信息。

AutoMxPrecVisitor 负责递归遍历 AST, 并在遍历复合语句 (CompoundStmt) 时, 通过判断代码块是否紧邻某个 #pragma AutoMxPrec 来识别目标区域。一旦进入目标区域, 插件会记录所有函数调用的名称, 并提取局部变量的名称及其所在函数信息, 用于 JSON 输出。

关键实现示例:

```
// 捕获 pragma
class AutoMxPrecPragmaHandler : public PragmaHandler {
    void HandlePragma(Preprocessor &PP, PragmaIntroducer, Token &Tok) override {
        GlobalPragmaLocs.push_back(Tok.getLocation());
    }
};
// 遍历复合语句并提取信息
    bool VisitCompoundStmt(CompoundStmt *CS) {
        SourceLocation StartLoc = CS->getBeginLoc();
        if (InsideTargetBlock) return true;

        for (auto PragmaLoc : GlobalPragmaLocs) {
            if (!SM.isWrittenInSameFile(PragmaLoc, StartLoc)) continue;
            if (SM.isBeforeInTranslationUnit(PragmaLoc, StartLoc) &&
                areLocationsAdjacent(PragmaLoc, StartLoc)) {

                InsideTargetBlock = true;
                TraverseStmt(CS);
                InsideTargetBlock = false;
                break;
```

```
            }
        }
        return true;
    }

    bool VisitCallExpr(CallExpr *CE) {
    if (!InsideTargetBlock) return true;
        if (auto *FD = CE->getDirectCallee()) {
            json callObj;
            callObj["type"] = "call";
            callObj["name"] = FD->getNameAsString();
            Data.push_back(callObj);
        }
        return true;
    }
```

3.3 llvm ir 循环嵌套深度分析 pass 的设计与实现

MaxLoopDepthPass 用于分析模块中每个函数的局部变量和函数调用的最大循环嵌套深度。该分析可用于程序性能优化和混合精度优化场景，通过记录每个变量和调用所在的最大循环层数，为优化工具提供循环上下文信息，从而指导优化策略。

3.3.1 模块分析流程

模块分析从 LLVM 模块层面遍历函数，主要步骤如下：

1. 获取函数的循环信息（LoopInfo），用于循环深度计算。
2. 遍历函数符号表，分析参数和局部变量的循环嵌套深度：对函数参数，若存在对应.addr 指针变量，则使用其深度，否则使用参数本身。对局部变量，仅分析浮点类型，忽略.addr 变量。
3. 遍历函数调用指令，识别函数调用并计算循环深度，对同名调用取最大值。
4. 将函数名称、变量信息和调用信息组织为 JSON 输出。

3.3.2 实现特点

插件利用 LLVM 的 LoopInfo 精确计算变量和函数调用的循环嵌套深度。在分析过程中，函数参数和局部变量分开处理，避免对.addr 指针重复统计。函数调用记录最大循环深度，保证数据的准确性和简洁性。输出结果为统一的 JSON 结构，可直接供后续分析工具使用。

3.4 精度配置解析 pass

ParseConfigPass 用于解析 JSON 配置文件中的精度信息，将其映射到 LLVM IR 中的全局变量、局部变量、函数调用和操作指令上。插件从模块层面遍历全局变量和函数，根据配置文件生成的 StrChange 或 FuncStrChange 对象，完成类型和精度的更新记录。插件通过命令行参数指定 JSON 文件路径，并在模块初始化时加载配置。

3.4.1 系统架构与分析流程

插件启动后首先加载 JSON 配置文件，将每个条目初始化为内部变换对象，记录变量变换信息。随后插件会遍历模块中的全局变量和函数。函数分析时，插件对参数和局部变量进行处理，参数会尝试查找对应的 Alloca 指令进行更新，避免重复记录。类型解析功能支持基础类型、数组类型、指针类型，能够根据配置对指针数据类型进行精度修改。

3.4.2 类型解析与指针依赖处理

插件调用 json 库，将 JSON 配置中的类型描述解析为 LLVM Type 对象，支持基础类型、数组、指针。基础类型包括半精度、单精度、双精度。对于数组类型，插件根据 [size] 描述从内向外构造多维数组。指针类型通过统计尾部 * 来确定指针层数，并生成对应的 PtrDep 对象用于处理不透明指针。通过这些机制，插件能够将 JSON 配置精确映射到 LLVM IR 的类型体系中。

3.4.3 实现特点

插件通过统一的数据结构管理全局变量、局部变量的类型变换。全局变量和函数参数在处理时进行了特殊处理，避免重复统计和错误更新。类型解析功能支持指针、数组，保证精度修改的准确性。所有变换信息可通过调试输出验证，并可直接用于后续自动精度转换流程，为 LLVM IR 层面的混合精度优化提供可靠基础。

3.5 llvm ir 降精 PASS 的设计与实现

3.5.1 普通变量降精

对于普通变量，其存储位置位于函数栈帧中，因此 llvm ir 文本文件中需要 alloca 指令来开辟内存存储它。故在降低其精度前，需要开辟一块新内存来存储该变量的低精度版本。

由于 llvm ir 是 SSA 的，所以 llvm ir 的 use-def 链非常清晰单一，后面使用该变量的所有指令的最初 def 只能是 alloca 指令的结果。故我们完全可以从 alloca 指令的结果出发，通过 use 链，递归访问该变量的所有使用点。

当访问该变量的某个使用点时，需要根据当前使用点的语义创建低精度版本指令。

降精过程一直递归，直到遇见无使用点、不需要浮点数的使用点（比如返回值是布尔值的指令）就终止。大致流程如图6。

由于 LLVM 17 强制开启不透明指针，即指针不再记录它所指向的类型，而是用 ptr 统一表示，这便使得我们无法直接获取 store 指令的目标地址的实际类型。如下所示:

```
int main(){
    double a = 2025.819;
    return 0;
}
```

上图中对应的 store 指令是

```
store double 0x409FA746A7EF9DB2, ptr %a, align 8
```

其中，目标地址的类型是指针类型，我们无法直接获取它指向的实际类型。

这里，我们提出通过 store 的存储值的类型推出目标地址指向的实际类型。

实际的代码获取方式如下:

```
Value* valueOp = storeInst->getValueOperand();
valueOp->getType();
```

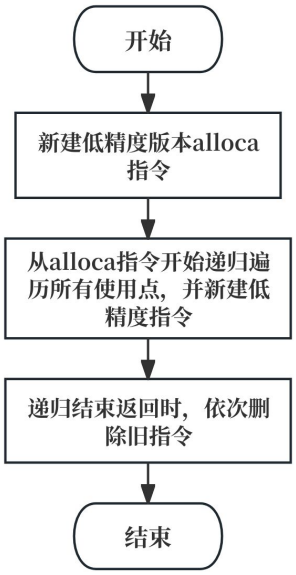

图 6 降精

3.5.2 指针变量降精

LLVM 17 的不透明指针机制，在上面的普通变量情况下，处理起来很简单，但是对指针变量就非常不友好。以下面代码为例。

```
int main(){
    double* p = (double*)malloc(sizeof(double));
    *p = 3.14159;
    free(p);
    return 0;
}
```

上面会生成下面三条指令：

```
%call = call noalias ptr @malloc(i64 noundef 8) #3
store ptr %call, ptr %p, align 8
%0 = load ptr, ptr %p, align 8
```

此时可以发现，操作数类型基本全为 ptr，无法直接得到任何实际类型。

对于该问题，我们提出了指示指针的 PtrDep 数据结构来解决该问题。

PtrDep 数据结构大致如下所示：

```
struct PtrDep {
    Type* ty;
    int dep;
    llvm::PointerType* getPoint();
    …… 
};
```

PtrDep 数据结构有两个属性，分别指示该指针指向的最终基类型 (ty) 与指针深度 (dep)。它还有若干成员函数，比如 getPoint(从 PtrDep 获得可用指针)。除此之外，还有若干运算符重载来支持 PtrDep 的基本运算，比如是否相同的判断。

3.5.3 函数调用的处理

函数调用可分为两种，一种是调用源程序中的自定义函数 (如 dtrsm)，另一种是调用数学库的函数 (如 sqrt)。

对于自定义函数，由于是变量级搜索，所以自然会将整个函数拆开来分析。此时，我们关注的是调用函数的参数与返回值降精，按照 C 程序语义，它们会被视作调用函数内的局部变量，所以便归为上面两种情况去处理。

对于数学库的函数，会创建低精度版本去替代它。

3.6 缓存搜索优化的设计与实现

3.6.1 去重机制与配置识别

缓存模块采用 **MD5 哈希算法**实现配置的快速去重和识别。每个配置参数组合通过哈希值唯一标识，确保在大量配置测试中能够以 **O(1) 时间复杂度**完成重复检测。系统还实现了**配置去重器 (ConfigDeduplication)**，在生成新配置时主动避免重复，从源头减少无效评估，**显著提升优化效率和收敛速度**

3.6.2 持久化存储与增量更新策略

缓存系统支持 **JSON 格式的持久化存储**，确保优化过程中断后能够快速恢复。采用**增量更新机制**，每次评估完成后立即更新缓存文件，避免数据丢失。同时提供**缓存统计信息**，包括已测试配置数量、缓存文件大小等，便于监控优化进度和系统状态。

3.6.3 多级缓存策略与查询优化

缓存模块实现了**多级缓存架构**，包括内存缓存和磁盘缓存。通过**查询优化**，优先检查内存缓存，减少 I/O 操作。系统还集成了**配置特征提取**功能，能够快速识别配置的相似性，为代理模型提供训练数据支持。缓存管理器 (CacheManager) 提供了丰富的 API 接口，支持缓存清理、详情查看、统计导出等功能。

3.7 性能预测器的设计与实现

3.7.1 运行决策策略设计

首先，我们构建起 xgboost 模型

为了判断是否需要进行实际评估，我们制定了运行决策策略，其中我们就指定了三个**重要的参数**：

- 跳过评估阈值 $skip\,evaluation\,threshold$
- 强制评估阈值 $force\,evaluation\,threshold$
- 置信度阈值 $surrogate\,confidence\,threshold$

那么当我们运行模型进行预测之后，会出现以下**几种情况**：

1. **预测分数方面**
   - 较历史最优差距大于 skip evaluation threshold（A 情况）
   - 较历史最优差距小于 skip evaluation threshold 大于 force evaluation threshold （B 情况）
   - 较历史最优差距小于 force evaluation threshold（C 情况）
2. **置信度方面**
   - 置信度大于 surrogate confidence threshold（可信，为 a 情况）
   - 置信度小于 surrogate confidence threshold（不可信，为 b 情况）

对于每种情况，我们会导向以下三种**方向**：

- **skip**：跳过实际评估，fitness 直接设为 0。
- **actual**：需要实际评估，最后返回实际评估的 fitness。
- **surrogate**：无需实际评估，直接返回预测分数。

判定方式依照以下**策略**，如表1。

	A	B	C
a	skip	surrogate	actual
b	skip	actual	actual

表 1 选择策略

3.7.2 预测器实现方案

1. **数据集构建与预处理**

首先提取原始的特征，即各个函数的变量指针比例，各精度的比例等等，

接着进行标准化：

$$\epsilon = (1/n)_{i=1}^{n} x_i$$

之后进行叶子节点特征提取：

$$L(x) = [l_1(x), l_2(x), ..., l_M(x)]^T$$

再通过基于梯度提升特征引擎工程将特征进行融合，得到交互特征:

$$I(x) = [I_1(x), I_2(x), ..., I_k(x)]^T$$

$$\Phi(x) = [x_{scaled}, L(x), I(x)]^T$$

最后进行重要性计算。

融合之后的交互特征可以表征每一个函数内的每一个精度的比例，从而代表该配置的特征。(我们曾经试过用每个变量对应的精度来作为离散特征，后因过拟合而弃用，加上了置信度来平衡精度，实测误差更小)

2. **预测模型训练方法**

多策略决策, 按优先级依次检查不同策略。同时采用 Monte Carlo 采样算法, 对输入特征添加高斯噪声，对每个样本进行多次预测，计算预测结果的标准差作为不确定性度量，借此获取不确定度

3. **性能预测执行流程**

得到每一代的种群及其个体之后，会对每一个个体进行分数预测，然后根据选择策略，判断是否预测。

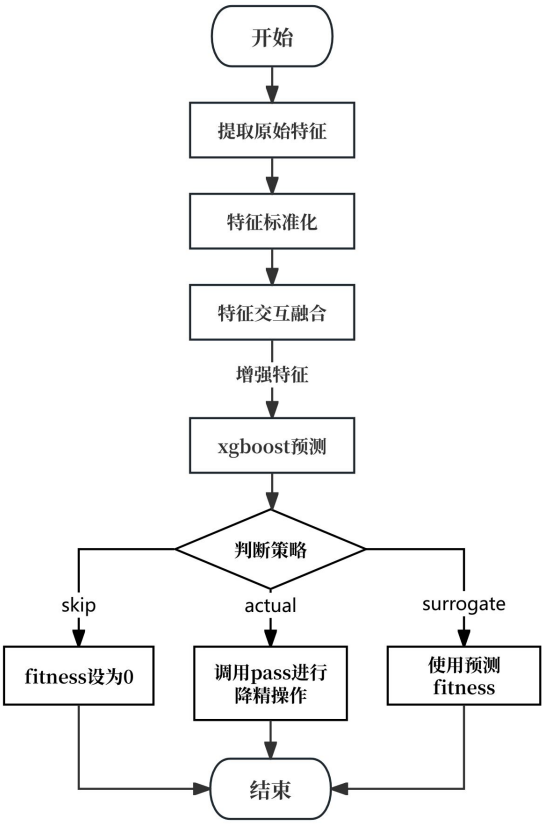

图 7 性能预测流程

3.8 粗糙搜索的设计与实现

3.8.1 分组获取算法 (Loop-Hierarchy Partitioning)

1. **分组的原理与算法**

在基准测试程序中，存在大量的多层嵌套循环，这导致众多变量位于不同的循环层次中，因此变量间的交互关系与访问模式各有差异。同一函数中位于同一循环层次的变量更倾向于拥有相同的精度，因为这可以降低变量间的精度转换开

销和保证缓存中数据的对齐与局部性。同时，我们通过先前进行的大量预实验也发现得分较高的配置中，同一循环层次中的不同变量往往是同一精度。

基于上面的发现与结论，我们提出**基于循环层次语义指导的分组算法**，以此来减少搜索的解空间。

基于循环层次语义指导的分组算法的输入是一个 json 文件，该 json 文件中指明了某变量的循环层次与所在函数。该算法核心思路为：对变量按循环层次分组。如下所示：

- **第一次分组**：以函数为粒度，同一函数中的若干变量组成一组。
- **第二次分组**：同一函数中嵌套循环深度大于等于 1 的为一组。
- **第三次分组**：同一函数中嵌套循环深度大于等于 2 的为一组。

  ……
- **第 n 次分组**：同一函数中嵌套循环深度大于等于 $n-1$ 的为一组。

通过分组，可以指数级减少搜索的解空间。经实测，在本测试用例中，可以显著缩小解空间，如下图8所示：

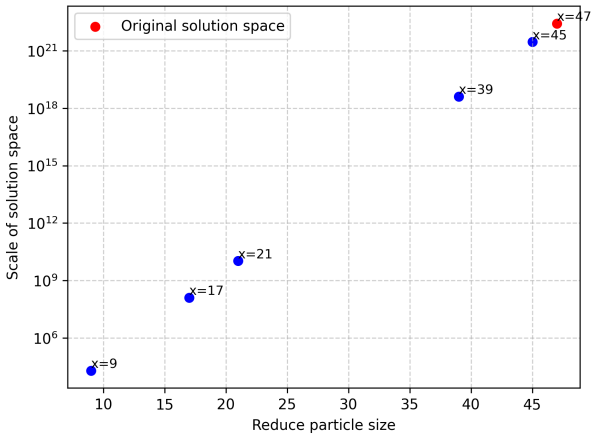

图 8 分组搜索空间

可以看到，最右侧是原始的 3^{47} 的搜索空间，经过分组算法之后，我们的多层缩减会将解空间变为逐步缩小的 3^x 个解空间，分别为：3^{45}、3^{39}、3^{21}、3^{17}、3^9。效果显著。

2. **基于分组的粗糙搜索**

基于嵌套深度的智能分组策略分组算法采用循环层次深度作为核心分组依据，通过分析变量在嵌套循环中的访问模式，将具有相似循环层次特征的变量进行分组。系统支持多种分组策略：

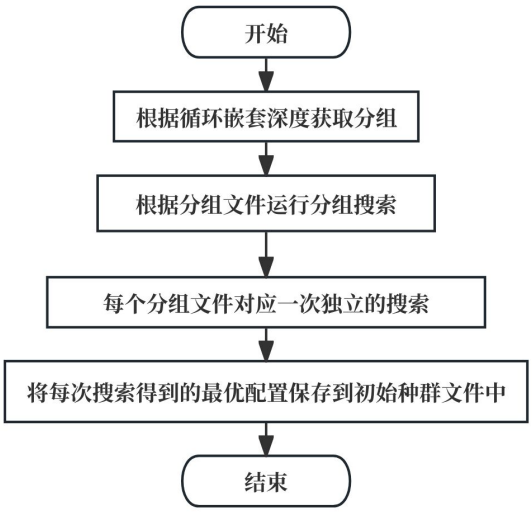

图 9 粗糙搜索

- **深度分层分组**：group_depth_ge_1.json 到 group_depth_ge_4.json，按循环嵌套深度 1 到 4 进行分组
- **函数级分组**：group_by_function.json，按函数边界进行变量分组
- **精度层次映射**：建立 double(3)→float(2) → half(1) 的精度层次体系

3. **工程实现细节**

模块化预处理与**数据流管理**实现采用预处理 + 主处理的两阶段架构：

架构

```
# 预处理阶段：数据清洗和格式转换
preprocess.py → local_var.json

# 主处理阶段：循环层次分析和分组生成
loop_hierarchy-partitioning.py → out.json → grouped/*.json
```

核心实现主要分为 4 点

- **数据预处理**：从原始变量数据中提取循环访问信息
- **层次分析算法**：计算每个变量在循环嵌套中的最大深度
- **分组文件生成**：自动生成多种分组策略的 JSON 文件
- **变量关系映射**：建立变量与循环层次的对应关系

3.8.2 分组搜索实现

1. **分组搜索引擎 (GroupSearchEngine)**

基于函数边界的精确分组调整分组搜索引擎实现了按函数分别处理的智能调整机制，确保同一分组内的变量保持一致的精度策略：

策略

```
    def adjust_config_by_group(self, config, group_file):
    # 按函数分别处理每个分组
    for function_name, var_list in self.group_data[group_file].items():
        # 找到该函数中分组变量的最低精度
        lowest_precision = self._find_lowest_precision_in_function_group(
            config, function_name, function_variables
        )
```

支持 5 种不同的分组策略文件，每种策略针对不同的优化目标：

架构

```
    self.group_files = [
        "group_depth_ge_1.json",     # 循环深度 1的变量分组
        "group_depth_ge_2.json",     # 循环深度 2的变量分组
        "group_depth_ge_3.json",     # 循环深度 3的变量分组
        "group_depth_ge_4.json",     # 循环深度 4的变量分组
        "group_by_function.json",    # 按函数边界分组
    ]
```

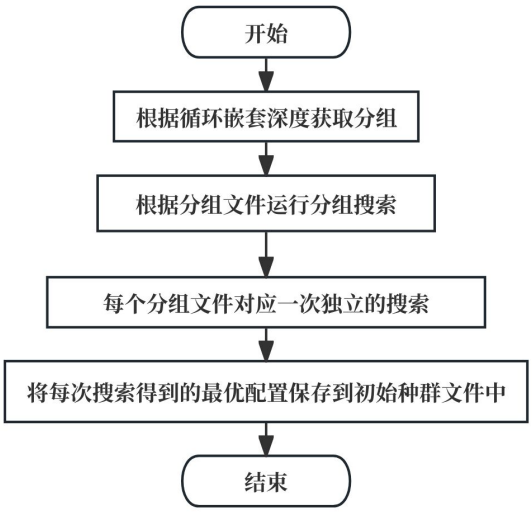

图 10 粗糙搜索

2. **简化分组优化器 (SimpleGroupOptimizer)**

- **代理模型保留**

  简化分组优化器在保持轻量级架构的同时，完整保留了代理模型的智能评估策略，通过减少实际评估次数显著提升搜索效率。同时，系统将每次实际测试的数据自动添加到训练数据集，为后续模型训练提供持续的数据积累

- **分组降维与快速收敛策略**

  由于分组约束显著降低了搜索空间的维度，系统采用小代数小种群的遗传算法即可快速搜索到较优配置。这种轻量级策略特别适合分组搜索的快速探索需求

- **多策略结果保存与精细化搜索准备**

  系统支持 5 种不同的分组策略，每种策略搜索得到的较优配置都会自动保存到初始种群文件夹，为后续的精细化搜索提供高质量的初始解：

Code Listing:

```
def _generate_group_search_summary(self, best_configs, configs_dir):
    """生成分组搜索汇总报告"""
summary = {
#报告内容
    }
    # 记录每个分组的结果到初始种群文件夹
    for result in best_configs:
        summary["group_results"].append({
            "group_file": result["group_file"],
            "fitness": result["fitness"],
            "config_file": f"group_search_{result['group_file'].replace('.json', '')}_best.json"
        })
```

- **粗糙搜索到精细化搜索的桥梁作用**

  通过快速的粗糙搜索，系统能够大大减少后续精细化搜索的收敛代数，实现从粗粒度到细粒度的渐进式优化

  桥梁机制优势

  可提供高质量初始解，粗糙搜索提供的高质量初始解加速精细化收敛，同时实现渐进式优化，实现从约束搜索到全局搜索的渐进式优化策略，以及实现了计算资源优化，通过粗糙搜索预筛选，减少精细化搜索的计算开销

3.9 精细搜索的设计与实现

3.9.1 核心算法流程设计

在精细搜索中，我们用到了前面的 GA 搜索进行全局搜索，之后针对最优个体进行具体 SA 搜索，对于每一个的配置，用测试用例评估适应度分数，同时采用了配置去重、代理模型预测等方法减少迭代时间，增快迭代速度。具体流程如图11

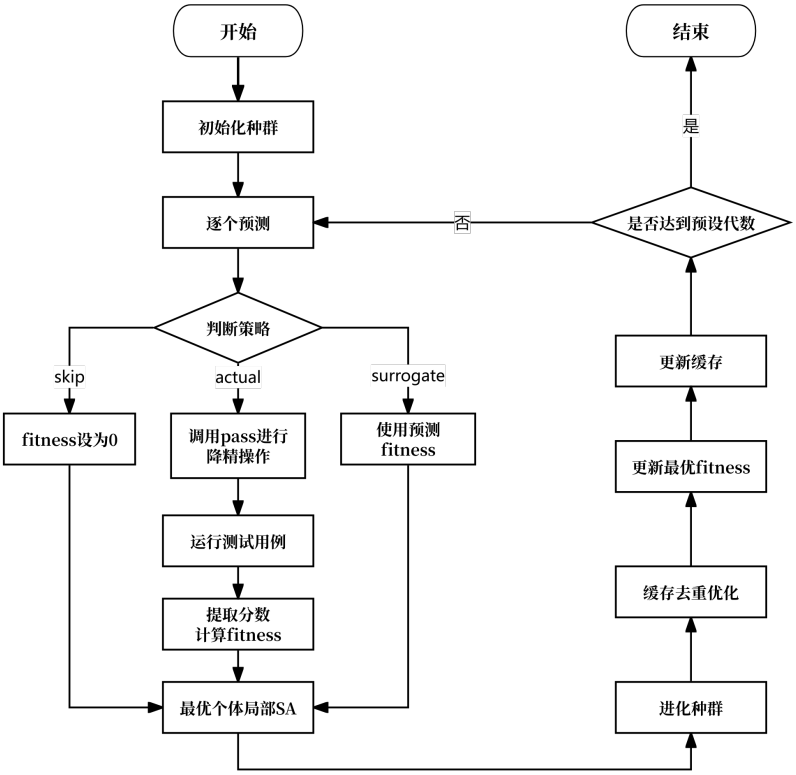

图 11 精细搜索主流程

3.9.2 功能模块实现方案

1. **配置评估策略**

我们利用 llvm 的工具 opt、llc 等工具，根据搜到的精度配置通过我们的降精 pass 将原测试用例调整精度，之后再将其链接成一个.ll 文件，再转换为可执行文件。通过运行可执行文件，可以读取到相应的分数，分为下列四个维度:

- Passing rate
- $Flops_{min}$
- $Flops_{mean}$
- $Flops_{max}$

接着，根据基准值算出当前分数的性能分数:

$$T_{0min} = 4.5471$$

$$T_{0mean} = 7.5516$$

$$T_{0max} = 9.6609$$

$$GFlops = (\frac{Flops_{min}}{T_{0min}} \times 10\% + \frac{Flops_{mean}}{T_{0mean}} \times 30\% + \frac{Flops_{max}}{T_{0max}} \times 60\%) \times 100/2 \times 60\%$$

那么最终适应度分数：

$$Fitness = PassingRate \times 40\% + GFlops \times 60\%$$

2. **精度转换方法**

我们的 pass 可以将目前的.ll 文件转换到对应精度的.ll 文件，而对于直接从 double 变到 half 的情况，我们这里采用了分步中间配置渐进式降精，具体流程图如图12。

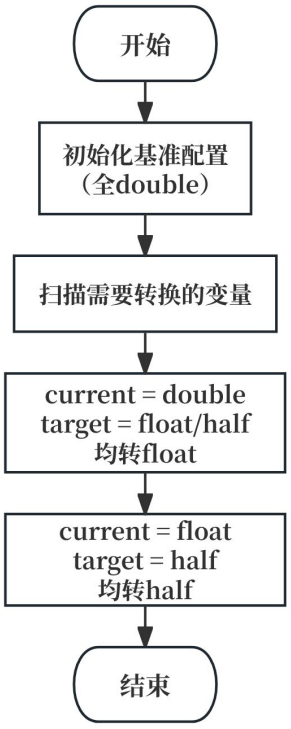

图 12 渐进式转换

3. **配置搜索算法**

我们首先使用遗传算法 GA 对目标精度配置进行全局搜索，在离散变量的搜索过程中，GA 的搜索功效显著。我们针对初始配置特点，定义结构体，将其精度配置进行染色体编码作为个体的基因，每一个配置都是一个个体，之后进行遗传算法的迭代搜索

为了加快收敛速度以及跳出局部最优解，我们同时对于每一代的最优个体进行了 SA 的局部搜索，这就使得我们在原始优秀配置的基础上能够更进一步。

4. **迭代优化机制**

基于测试用例运行的速度缓慢这一现状，为了缩短迭代时间，提升收敛速度，我们采取了配置去重机制和代理模型预测机制：一方面可以让已测的配置不再测，另一方面可以让不需要测试的配置无需测。

配置去重记录了已经运行过的配置，将其存放在缓存之中，

代理模型采用的是 xgboost 模型，相对于随机森林和贝叶斯模型，在实操下，其在预测精度、训练速度以及泛化性等等方面均有明显优势。

4 系统测试

我们对比了三种算法：dd2、GA 和 AMP-2025，实验环境是华为鲲鹏 920ARM 服务器，测试数据规模是 5 300 1600，测试时长均为 19 分钟。

4.1 系统有效性测试

我们将三种算法：dd2、GA 和 AMP-2025 的搜索最高分与初始的分数进行对比，得到其加速比如表2:

原始	dd2	GA	AMP-2025
1	1.58	1.15	2.06

表 2 加速比对比

从表中可以发现，AMP-2025 搜出的解质量远大于另外两种算法的质量。

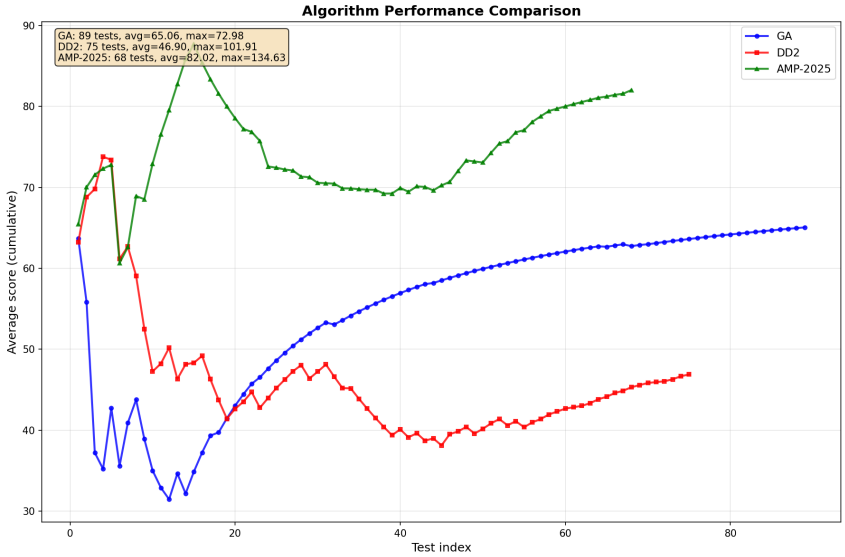

图 13 algorithm_comparison

上图展示了三种算法的累积平均值的变化情况，可以发现在绝大部分情况下，AMP-2025 无论是平均值还是最大值，都高于其他两种算法，这说明在相同时间内，AMP-2025 搜索出的解质量最高。

4.2 系统收敛性测试

下面的图14分别展示了三种算法在搜索过程中的分数变化情况，图中的 AMP-2025 只展示了精细搜索过程中的个体分数，其粗糙搜索时长为 4 分钟，所以图中所展示的搜索过程对应的时长为 15 分钟。

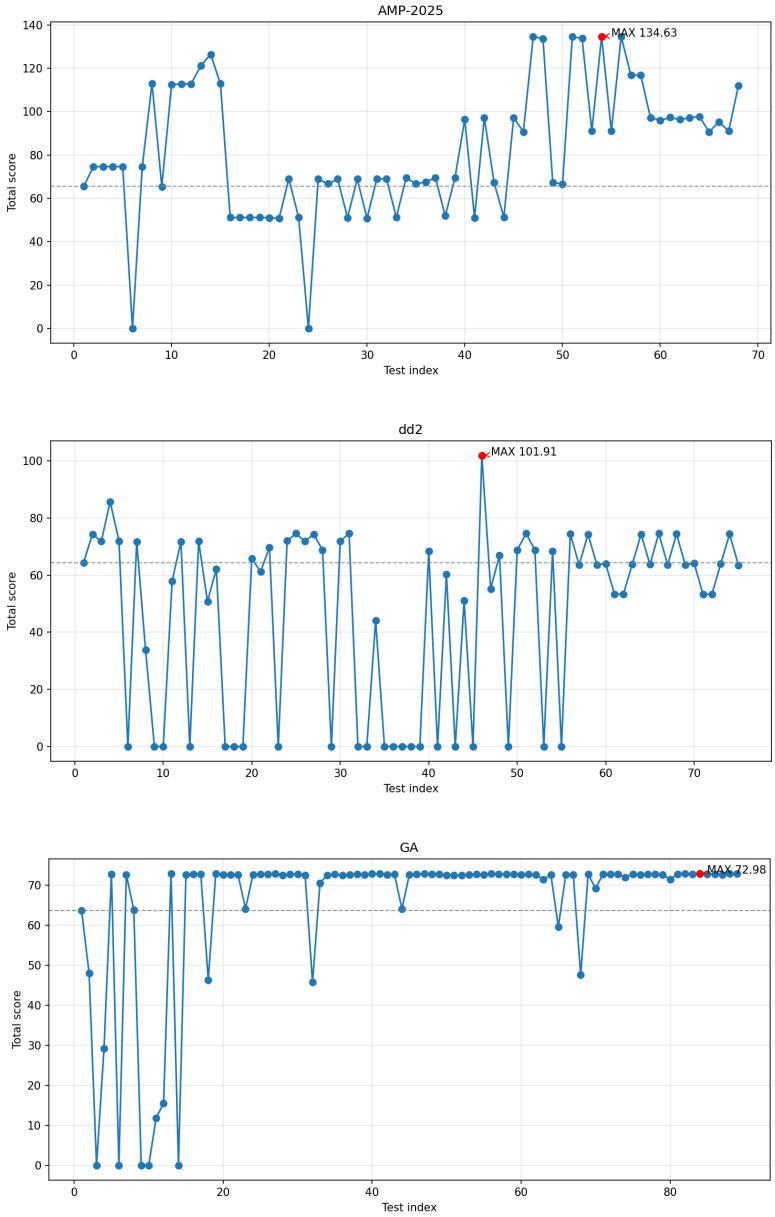

图 14 从上至下：AMP-2025、dd2、GA 的收敛性能

上述图可以看出，dd2 的搜索过程分数波动很大，这是因为 dd2 的核心是二分法，缺乏信息指导。GA 的搜索刚开始波动很大，但后续趋于平稳，陷入局部最优解。两者搜到的最高分均远小于 AMP-2025 的最高分。

AMP-2025 的整体波动幅度较小，且有多个平台期，其成因主要是性能预测器规避了不必要的实际评估，加速了搜索过程，最终在相同时间内获得了远优于其他两种算法的最高分。

5 创新点说明

AMP-2025 混精编译优化系统的主要创新点如下：

1. 我们实现了**自动识别预处理** #pragma AutoMxPrec 指令的 clang 插件
2. 我们提出了 **PtrDep 数据结构**来解决了 llvm17 中的**不透明指针**问题。
3. 我们在 llvm17 上实现了**渐进式降精 pass**
4. 我们提出了**基于循环层次语义的分组算法**来加速收敛
5. 我们提出了**基于 xgboost 的性能预测器**来减少迭代时间
6. 我们设计了**缓存优化机制**，参考禁忌搜索存储已测配置，借此加快迭代速度
7. 我们将 **GA 的全局搜索**和 **SA 的局部调优**结合，显著提高收敛速度和精度上限
8. 我们提出了**粗糙 + 精细搜索**多层复合搜索策略。

6 参考资料

本研究主要参考了 precimonious 和 CSAPP。

7 致谢

感谢合肥工业大学刘彬彬老师、李宏芒老师对比赛提供的支持。
感谢大连理工大学江贺老师对问题研究总体方向提出的宝贵建议。
感谢大连理工大学任志磊老师、周志德老师、贾昂老师对部分技术细节给予的支持。
感谢成龙老师等专家对我们研究提出的珍贵建议与展望。
感谢张琪凡学长、张[illegible]学长、杨梓学长对我们的支持。