兆松科技
Tera Pines

利用人工智能大模型优化RISCV编译器性能

演讲者：赵正朋

职位：软件工程师

兆松科技（武汉）有限公司

传统编译器优化现状

编译器优化算法已经较为成熟，现代编译器常常使用以下手段进行调优：

- 添加优化参数
- 调整各个优化执行的顺序
- 通过分析汇编代码找出优化不佳的代码片段进行修复针对目标平台特性进行调优
- 针对平台特性进行优化

使用优化参数进行调优

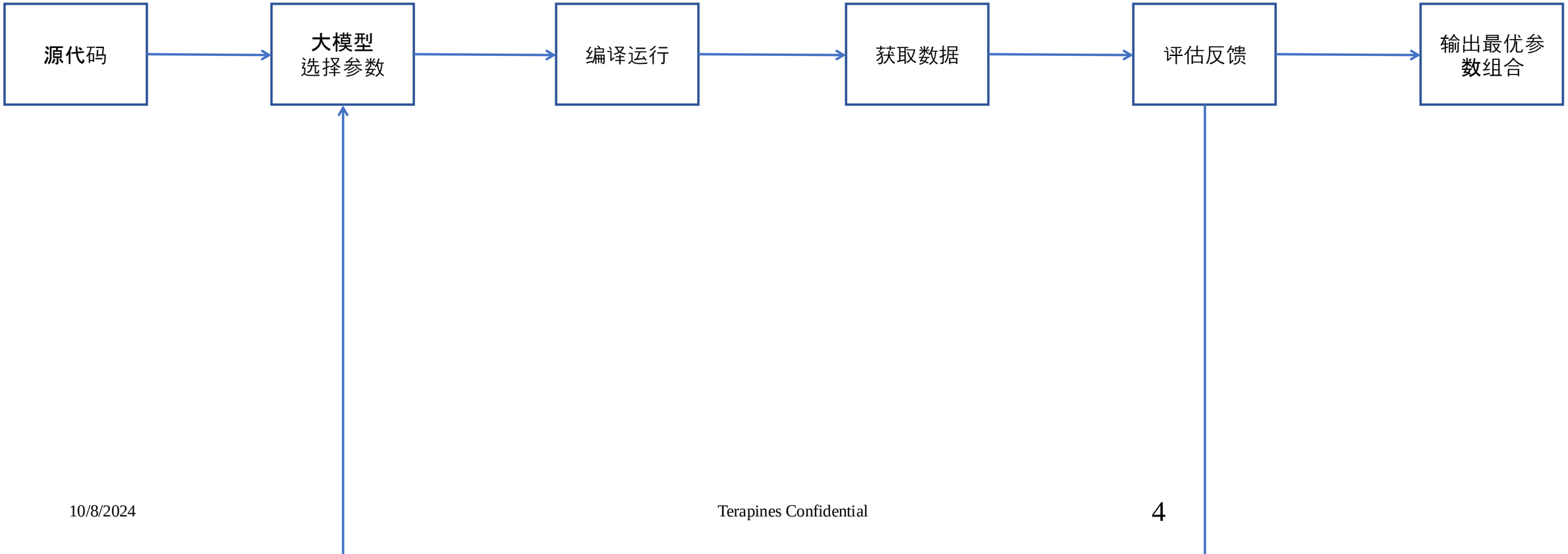
使用编译器O3优化coremark程序：
gcc -O3 core_main.c core_matrix.c core_util.c core_list_join.c core_portme.c core_state.c -o coremark

使用额外调优参数将coremark程序跑分再提升20%：
gcc -O3 -Ofast -Wno-int-to-pointer-cast -mbranch-cost=1 -mstrict-align -funroll-all-loops -ftree-dominator-opts -fselective-scheduling -finline-functions -falign-functions=12 -falign-jumps=7 -finline-limit=800 -fno-common -fno-tree-vectorize -fno-tree-loop-ivcanon -fgcse-las --param=max-loop-header-insns=4 --param max-jump-thread-duplication-stmts=14 core_main.c core_matrix.c core_util.c core_list_join.c core_portme.c core_state.c -o coremark

但是该参数组合对其他程序却没有优化效果甚至有反效果！

使用大模型自动调优

使用大模型，根据输入代码特性选择参数。
可以降低优化参数的使用门槛，最大化编译器的优化性能。
针对不同的基准测试程序，与O3优化相比，最终加速比在 1.15 倍到 2.82 倍之间。



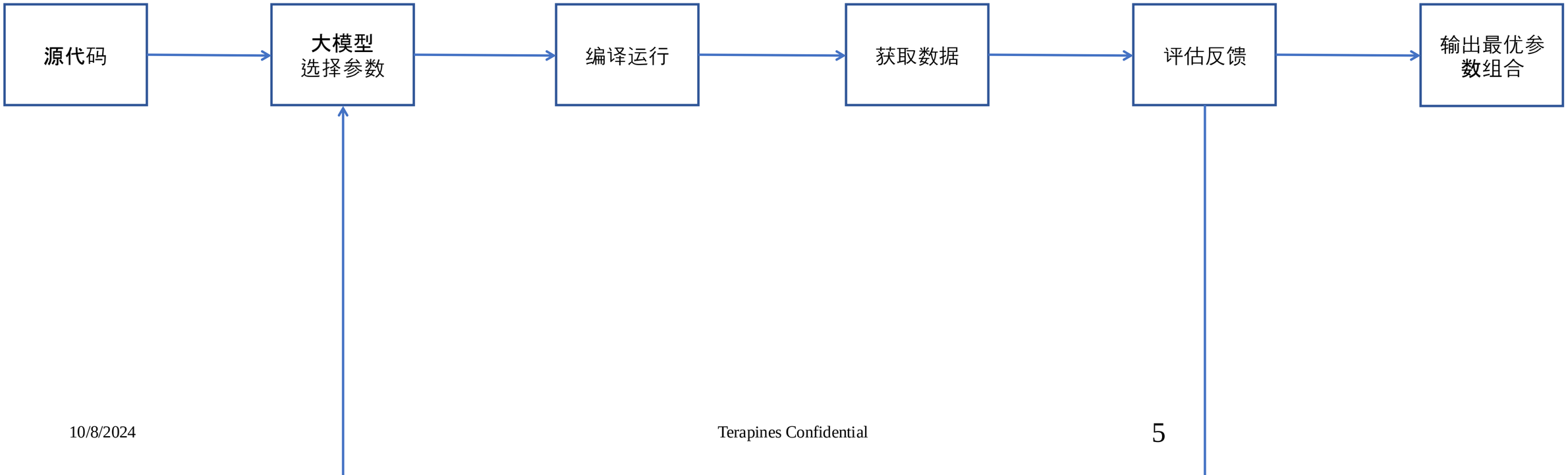
使用大模型自动调优

优势：

- **参数**组合极其复杂，使用大模型可以在**有限**时间选择出较好的参数组合
- **可以根据源**码特性选择不同的参数组合

难点：

- **源**码输入较大
- **参数**优化效果和编译器版本相关



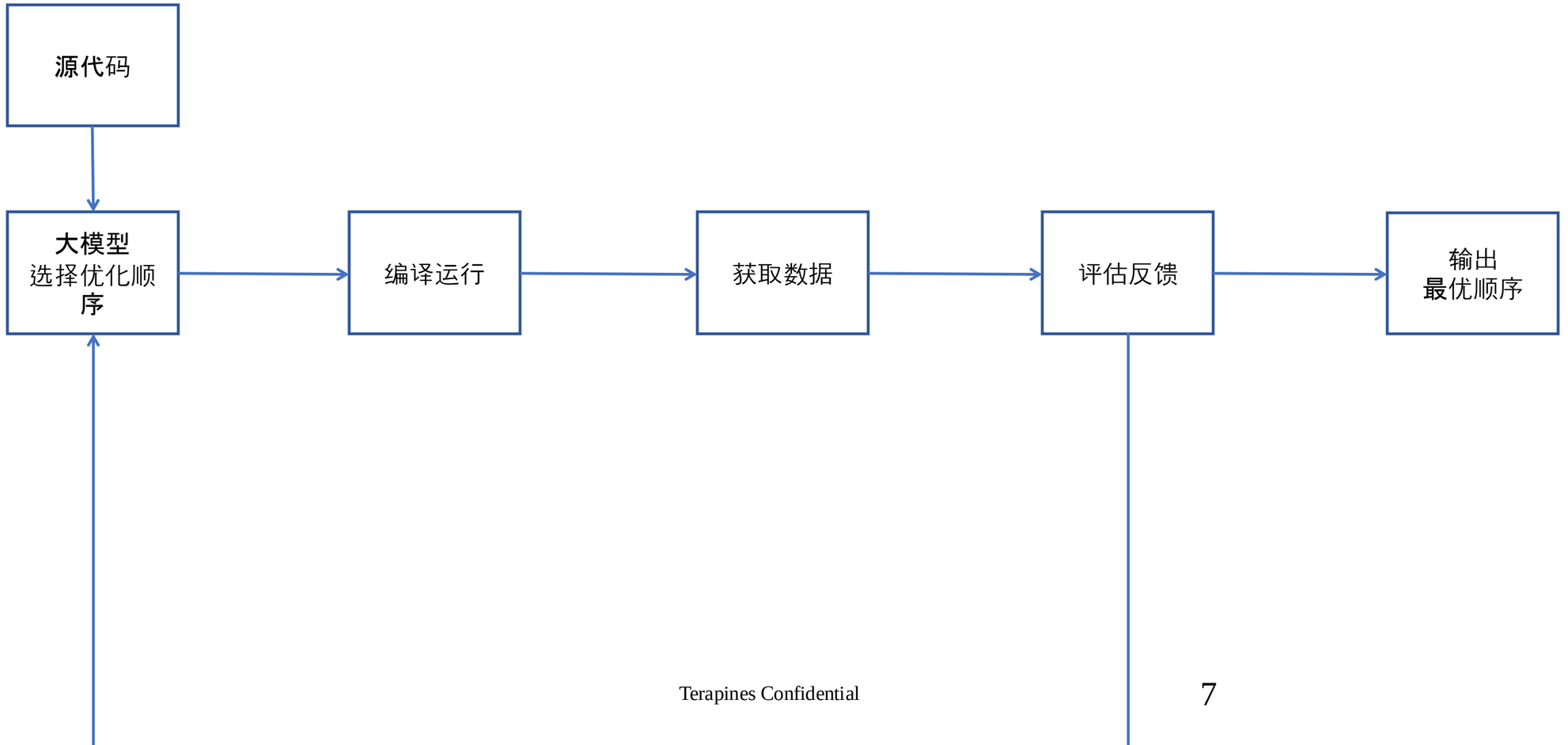
优化算法执行顺序

编译器使用预先定义好的顺序来执行各个优化算法，这些顺序根据经验而来，对不同的代码有不同效果

```
ModulePassManager
PassBuilder::buildPerModuleDefaultPipeline(OptimizationLevel Level,
                                           bool LTOPreLink) {
  if (Level == OptimizationLevel::O0)
    return buildO0DefaultPipeline(Level, LTOPreLink);

  ModulePassManager MPM;

  // Convert @llvm.global.annotations to !annotation metadata.
  MPM.addPass(Annotation2MetadataPass());

  // Force any function attributes we want the rest of the pipeline to observe.
  MPM.addPass(ForceFunctionAttrsPass());

  if (PGOOpt && PGOOpt->DebugInfoForProfiling)
    MPM.addPass(createModuleToFunctionPassAdaptor(AddDiscriminatorsPass()));

  // Apply module pipeline start EP callback.
  invokePipelineStartEPCallbacks(MPM, Level);

  const ThinOrFullLTOPhase LTOPhase = LTOPreLink
                                          ? ThinOrFullLTOPhase::FullLTOPreLink
                                          : ThinOrFullLTOPhase::None;
  // Add the core simplification pipeline.
  MPM.addPass(buildModuleSimplificationPipeline(Level, LTOPhase));

  // Now add the optimization pipeline.
  MPM.addPass(buildModuleOptimizationPipeline(Level, LTOPhase));

  if (PGOOpt && PGOOpt->PseudoProbeForProfiling &&
      PGOOpt->Action == PGOOptions::SampleUse)
    MPM.addPass(PseudoProbeUpdatePass());

  // Emit annotation remarks.
  addAnnotationRemarksPass(MPM);

  if (LTOPreLink)
    addRequiredLTOPreLinkPasses(MPM);
  return MPM;
}
```

优化算法执行顺序

目前的优化顺序高度依赖于编译器工程师的经验，很难调整出一个最优的顺序。
使用AI模型寻找最优的优化执行顺序。
针对不同的基准测试程序，与O3优化相比，最终加速比在 1.1倍左右。



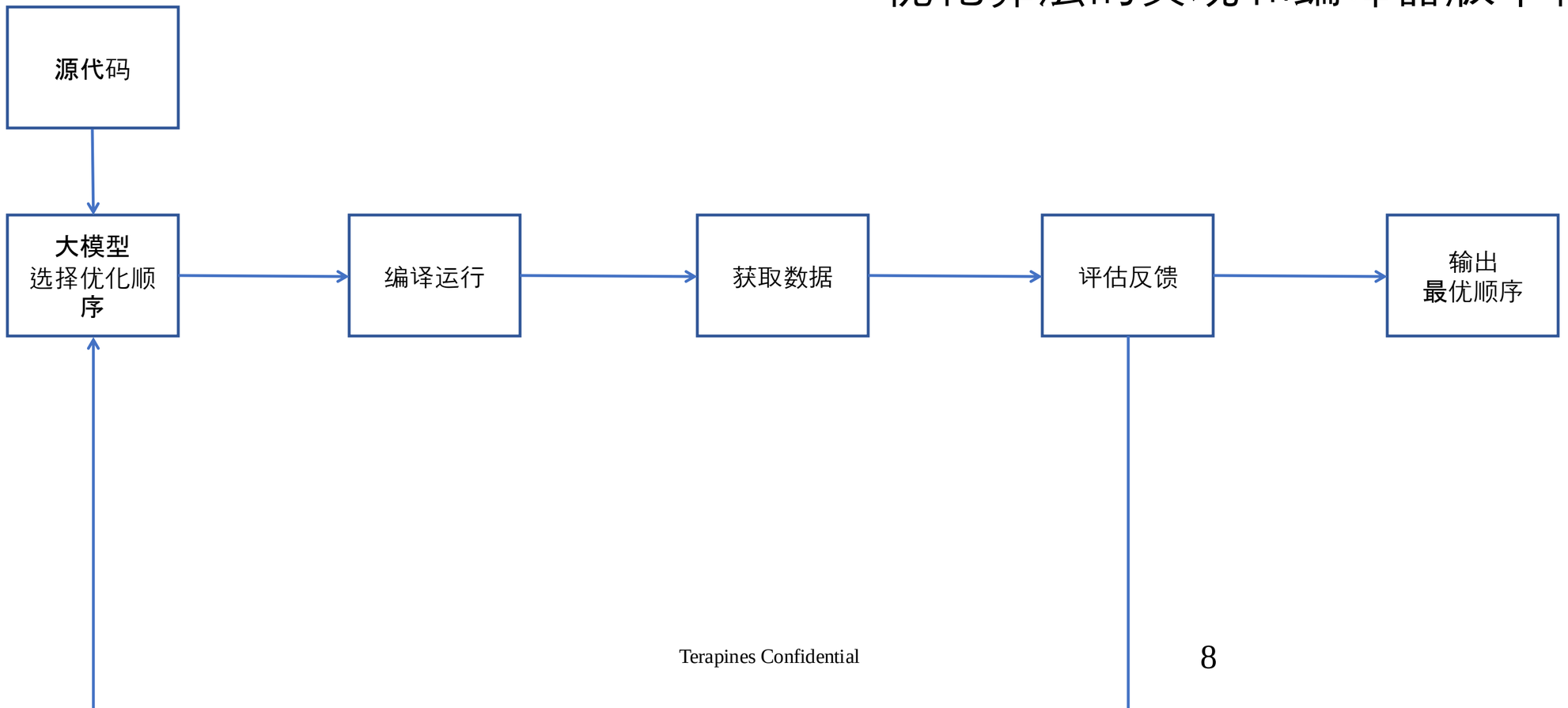
优化算法执行顺序

优势：

- **可以**选择最优的优化顺序
- **可以根据源**码特性选择不同的优化顺序

难点：

- **部分**优化算法之间有依赖关系
- 选择合适大模型
- 优化算法的实现和编译器版本相关



代码片段优化

编译器工程师在分析代码性能时如果发现了左边的代码片段，则会想办法将其优化为右边的代码片段

```
foo:
        addi    a0, a0, 1
        lbu     a2, 0(a0)
        sb      a2, 0(a1)
        ret
```

```
foo_opt:
        lbu     a2, 1(a0)
        addi    a0, a0, 1
        sb      a2, 0(a1)
        ret
```

使用大模型生成代码

使用大模型学习代码逻辑，直接产生优化后的代码。

源代码常用的格式有：

- 高级语言如c、c++
- 中间代码如LLVM IR
- 汇编语言

使用大模型生成代码

难点：

- 源码输入较大，且上下文有联系
- 确保产生代码的正确性

平台相关优化

```
int min(int num) {
    return num > 0x7FFF ? 0x7FFF : num;
}
```

编译为不带zbb扩展的汇编文件

```
min(int):
        lui     a1, 8
        addi    a1, a1, -1
        blt     a0, a1, .LBB0_2
        mv      a0, a1
.LBB0_2:
        ret
```

编译为带zbb扩展的汇编文件

```
min(int):
        lui     a1, 8
        addi    a1, a1, -1
        min     a0, a0, a1
        ret
```

min指令的产生依赖于编译器工程师编写相应的匹配逻辑

使用大模型生成自定义指

使用大模型学习指令定义，自动生成扩展指令。

实现：

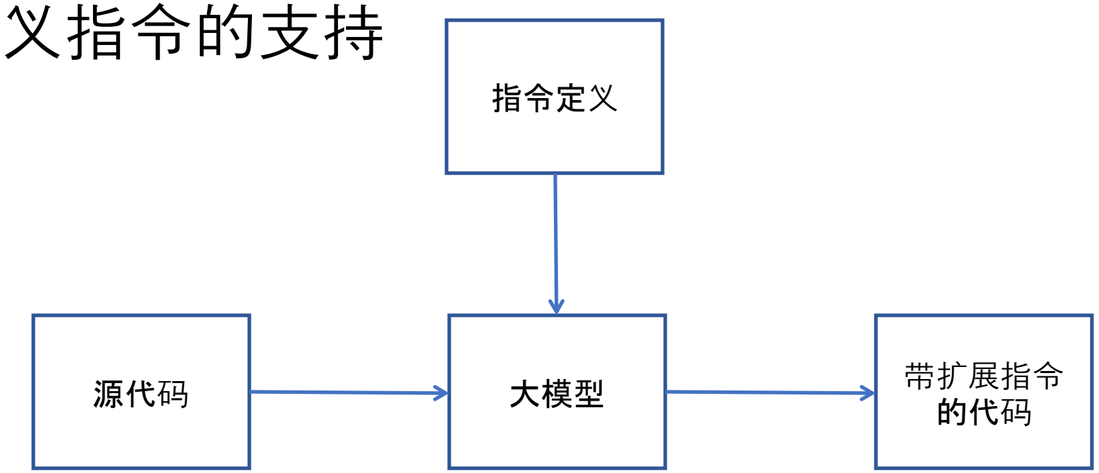
- 对代码片段进行分析是否符合产生自定义扩展的语义

优势：

- 快速添加自定义指令的支持

难点：

- 产生代码的正确性难以保证



总结

- **使用大模型**选择优化参数

  可以极大降低编译器优化参数的使用门槛。
- **使用大模型**调整各个优化执行的顺序

  可以供编译器工程师探索更优的优化流水线。根据源代码特性自动产生优化流水线可以进一步提高优化性能。
- **使用大模型**优化代码

  正确性的问题若能得到解决，未来可以完全替代传统编译器。
- **使用大模型**产生自定义指令

  使用大模型可以快速进行指令的定制化，但是还需先解决大模型产生代码的正确性问题。

兆松科技
Tera Pines

谢谢观看

Thanks