编译器后端设计与优化

杨子凡

Jun 26, 2026

后端在编译流程中负责把前端生成的中立中间表示最终转化为能在具体硬件上高效运行的机器代码，其核心工作直接决定了程序的执行速度与资源占用。本文聚焦 LLVM 后端，同时兼顾 GCC 与 TVM 等系统，围绕中间表示、指令选择、寄存器分配、指令调度、目标文件生成以及现代优化技术，系统梳理后端各阶段的设计思路与实现细节，力求让读者理解 Pass 流水线的内在机制，并具备编写自定义优化的能力。

1　编译器后端整体架构

编译器通常采用前端、中端、后端的三段式结构，前端负责词法与语法分析并产出抽象语法树，中端在语言无关的中间表示上实施通用优化，后端则把优化后的中间表示映射到目标机器指令。LLVM 把后端进一步划分为指令选择、寄存器分配、指令调度、窥孔优化与汇编发射等阶段，每一阶段均以可插拔的 Pass 形式组织，构成一条完整的代码生成流水线。

在 LLVM 中，代码生成从 LLVM IR 开始，逐步降低到 Machine IR，再到 MCInst，最后输出汇编或目标文件。Machine IR 保留了控制流图结构，但已经携带虚拟寄存器与目标相关的信息。整个过程由 `TargetPassConfig` 负责编排，开发者可通过在目标描述文件中注册自定义 Pass 来插入或替换默认阶段，从而实现针对特定应用场景的优化。

2　中间表示在后端的作用

LLVM IR 采用静态单赋值形式，清晰表达了变量的定义-使用关系，便于进行跨基本块的分析。当后端开始工作时，LLVM IR 首先被转换为 MachineFunction，每个函数对应一个 MachineFunction 对象，其内部由若干 MachineBasicBlock 组成，而每个 MachineBasicBlock 又包含一系列 MachineInstr。MachineInstr 的操作数可以是虚拟寄存器、物理寄存器、立即数或内存地址，这一层次的表示既保留了数据流信息，又引入了目标相关约束。

为了进一步支持汇编与目标文件生成，LLVM 提供了 MC 层。MCInst 与 MCOperand 构成了一套与具体指令集无关但可直接映射到二进制编码的中间表示。开发者可通过 `llc -print-after-all` 在任意 Pass 之后打印 MIR，从而直观地观察指令选择、寄存器分配等阶段对代码形态的影响。此外，`llvm-mca` 工具能够基于 MC 层信息进行乱序执行模拟，快速评估指令延迟与吞吐量。

3 指令选择

指令选择负责把目标无关的 IR 操作映射到具体指令集中的一条或多条机器指令。经典的树覆盖算法把表达式构造成树，然后用代价最小的模式覆盖整棵树。BURG 与 BURS 自动生成器可根据指令集描述自动生成匹配器，显著降低手工编写选择逻辑的工作量。

LLVM 同时维护 SelectionDAG 与 GlobalISel 两套引擎。SelectionDAG 把基本块内的操作构造成有向无环图，通过 TableGen 描述的模式进行匹配；GlobalISel 则采用更通用的 Legalize-Select-RegBankSelect 流程，适合支持更多目标与更细粒度的优化。无论采用哪种引擎，开发者都可在 .td 文件中编写形如以下的模式描述：

```
def : Pat<(add i32:$src1, i32:$src2),
          (ADDWrr i32:$src1, i32:$src2)>;
```

该模式把 LLVM IR 中的 32 位加法直接映射到 AArch64 的 ADDWrr 指令。指令选择阶段还可进行多指令融合，例如把连续的加载-加法合并为带偏移的加载指令，从而减少动态指令数并改善流水线利用率。

4 寄存器分配

寄存器分配把无限的虚拟寄存器映射到有限的物理寄存器，是后端优化的核心环节。图染色算法把变量活跃区间抽象为冲突图，图中的节点代表变量，边代表同时活跃的冲突关系；若图不可染色，则需要插入溢出代码把变量写回内存。线性扫描算法则按变量活跃区间的起始位置排序，采用启发式策略快速完成分配，在编译时间与代码质量之间取得了良好平衡。

LLVM 内置 Greedy、Basic 与 Fast 三种分配器。Greedy 分配器在 SSA 形式下利用 Live Interval Analysis 精确计算活跃区间，并通过 Coalescing 合并拷贝指令以减少动态指令。Basic 分配器则更注重编译速度，适合调试场景。针对向量与谓词寄存器，LLVM 额外维护了寄存器 bank 与重叠约束信息，确保分配结果满足硬件对寄存器对齐与重叠使用的限制。

5 指令调度与布局

指令调度在保证语义的前提下重新排列指令顺序，以隐藏延迟并提高指令级并行度。数据相关性分析首先构建 Data Dependence Graph，图中的边表示读后写、写后读或写后写关系；Alias Analysis 则进一步判断两条内存访问是否可能指向同一地址，从而放宽或收紧调度约束。List Scheduling 是最常用的启发式算法，它按优先级从就绪队列中挑选指令发射；Modulo Scheduling 则针对循环，通过固定启动间隔实现软件流水，把相邻迭代的指令交错执行以充分利用功能单元。

基本块布局优化关注控制流与指令缓存行为。Block Placement 算法把频繁执行的路径放在连续的内存区域，减少分支预测失败与 I-Cache 未命中。LLVM 还支持对函数分段、异常处理帧等特殊区域进行布局，确保运行时能够正确解析栈回溯与异常信息。

6 窥孔优化与 CodeGen 后期清理

窥孔优化在指令序列中寻找可局部改写的模式，例如把连续的 `ldr r0, [r1]` 与 `add r0, r0, #1` 合并为带偏移的加载指令。LLVM 的 `PeepholeOptimizer` Pass 提供了一套基于 `MachineCombiner` 的框架，开发者只需注册匹配模式与替换规则即可自动生效。该阶段还能消除零开销抽象，例如把 C++ 中空的析构函数调用直接删除，避免不必要的函数调用开销。

7 目标文件生成与后端工具链

MC 层负责把 MCInst 编码为目标文件或可执行文件。MCStreamer 定义了统一的流式接口，具体实现包括 `MCAsmStreamer`（输出汇编文本）与 `MCObjectStreamer`（输出 ELF/Mach-O）。`MCCodeEmitter` 把指令编码为二进制字节流，`MCObjectWriter` 则负责写入段、符号表与重定位信息。目标描述文件 `.td` 中定义的 `InstrInfo`、`RegisterInfo` 与 `CallingConv` 表格会被 TableGen 转换为 C++ 代码，供代码生成各阶段查询。

运行时支持包括异常处理信息与线程局部存储模型。LLVM 使用 `libunwind` 兼容的 DWARF 调用帧信息来实现栈回溯，同时支持 GD、LD 与 IE 等 TLS 模型，确保多线程程序能够正确访问线程私有数据。

8 现代优化专题

链接时优化 LTO 把多个模块的 IR 合并后再次进行全局优化，ThinLTO 则通过摘要文件实现跨模块内联与常量传播，同时保持较低的编译内存占用。过程间优化 IPA 进一步分析全局值编号与逃逸信息，帮助消除冗余计算与内存访问。

面向特定硬件的优化在 GPU 与 AI 芯片领域尤为关键。NVIDIA 的 PTX 到 SASS 后端需要考虑线程束级调度与共享内存 bank 冲突；TVM 的 TensorIR 则通过张量布局重写与算子融合，把高层次张量表达式映射到高效的硬件指令。安全性加固方面，LLVM 支持 ARM Pointer Authentication、Intel CET 以及堆栈保护等机制，在性能损失可控的前提下提升程序抗攻击能力。

9 性能剖析与调试

后端性能指标主要包括代码大小、执行周期与寄存器溢出率。`llvm-mca` 可对汇编片段进行乱序执行模拟，快速给出每条指令的延迟与吞吐量；`perf` 与 VTune 则在真实硬件上采集 Cycle、Cache Miss 与分支预测失败等事件。`llvm-exegesis` 工具能够自动生成微基准，测量单条指令在目标 CPU 上的实际延迟，为调度算法提供精确的代价模型。

10 实践项目建议

开发者可通过新增一条饱和加法指令并实现自动选择来熟悉指令选择流程。首先在 `.td` 文件中定义新指令的编码与语义，再编写对应的 SelectionDAG 模式，最后通过测试用例验证选择器能否正确匹配。自定义寄存器分配器则需要继承 `RegAllocBase`，实现 `selectOrSplit` 接口，并通过对比不同分配策略在 SPEC CPU 上的性能差异来评估效果。在 RISC-V 平台上实现软件流水，需要分析循环的数据依赖图，计算最小启动间隔，并生成

相应的模调度代码，最后用 Cycle 级模拟器验证吞吐量提升。

11 未来趋势与挑战

机器学习引导的优化 MLGO 利用强化学习动态调整内联、展开与寄存器分配策略，已在 Google 内部生产环境取得显著收益。异构与分布式编译则要求后端能够同时面向 CPU、GPU 与专用加速器生成代码，并支持跨设备的数据移动与同步。安全、性能与能效三者之间的权衡日益成为后端设计的核心约束，如何在保证机密性与完整性的同时维持高能效，仍是开放的研究课题。

12 结论与参考资料

后端在可移植性与极致性能之间扮演着关键角色，其设计质量直接决定了软件在真实硬件上的表现。深入理解 LLVM 的 Pass 流水线与目标描述机制，能够帮助开发者快速定位性能瓶颈并定制优化策略。延伸阅读可参考《Engineering a Compiler》以及 LLVM Discourse 上的 RFC 列表，结合动手实践不断打磨后端调优能力。