# 优化设计

## 总览

编译优化从实现效果角度可以分为分析类和执行类，分析类优化着重于对中间代码分析，提取有效信息，而真正实现对中间代码质量上的修缮提升还是执行类优化的任务，这一特点主要体现在中端优化，而恰恰中端优化才是我们大展拳脚的地方。在最终的编译器实现上，我们并没有完全按照先分析后执行的顺序进行优化，这是因为每阶段优化完成之后会改变分析目标的结构，我们需要将各个分析过程拆开来独立分析，但是需要注意的是显然中间代码优化一定在后端优化之前。下面笔者将按照自己的编译优化顺序进行各阶段剖析。

## 中间代码优化

### DeadCodeRemove

> 死代码删除

编译理论中，“DeadCode” 指的是永远执行不到的指令或者执行了也没有实际作用的指令，我们减少这种不可达指令的好处是可以缩小中间代码的体积，实际生产中，减少代码体积可以提高 Cache 块的命中率，减少与磁盘的交互，进一步提高性能。

笔者主要实现了三方面的死代码删除

- 无法通过 Br 跳转到的基本块
    - 实现：DFS 遍历函数内基本块，将每个基本块末尾 Br 指令的目标块加入可达基本块集合，剩余基本块则不可达，删除即可
- 块内 Br 之后的不可达指令
    - 实现：基本块内 Br 指令之后的指令都不可达，根据基本块的划分方式，块内的最后一条终端指令一定是 Br 或者 ret，故其后指令必不可达
- 块内的无用指令
    - 如果块内某些指令的结果值可以被使用，但是并没有任何的使用者，则可以将其删除。但是注意：CallInsr 的执行可能产生对全局变量的操作，以及其内可能存在的 IOInstr 是必然执行的IO操作，所以 CallInstr 和 IOInstr 都不可以删除

### CFG

> Control Flow Graph 控制流图

这是一个大型分析模块，主要包含了五个方面的分析：

- 建立 CFG 流图
    - 目的：为后续建立 DomTree ，活跃变量分析等提供基础
    - 实现：对于任意的基本块X，若 X 块内末尾 Br 指令的 Target BasicBlock 为 Y，则 X 的 successor 中有 Y， Y 的 predecessor 中有 X

- 建立 DomTree 支配树
  - Dominate Tree
  - 定义：X 为从 entry 到 Y 的必经块，则 X dom Y，X 支配 Y，Y 被 X 支配。
  - 目的："支配" 内含 "必经" 这一强约束，遍历时先 X 后 Y，可以保证对于 Y 内变量的引用一定已经存在声明，同时 DomTree 服务于 IDom 的确定
  - 实现：从定义出发，BFS 遍历 CFG 流图，对于路径 entry -> X 上的所有基本块，都是可以不经过 X 而达的基本块，欲到达剩余基本基本块，必须经过 X，故剩余基本块为 X 的支配块
- 建立 IDom 直接支配关系
  - Immediate Dominate
  - 定义：所有严格支配 Y 的节点中离 Y 最近的那一个节点。严格支配：若 X支配 Y，且 X 不等于 Y，则 X 严格支配 Y
  - 目的：服务于 DF 支配边界建立，寄存器分配等
  - 实现：对每个节点 X（基本块），判断其和 successor 中的每一个 Y是否有直接支配关系：遍历 X 的 successor，如果存在某个 Z，Z 不等于 Y，并且 Z 支配 Y，显然支配关系为 X->Z->Y，则 X 不是 Y 的直接支配者。对每个块执行如上操作即可。
- 建立 DF 支配边界
  - Dominance Frontier
  - 定义：节点 X 的支配边界是 CFG 中刚好不被 X 支配到的节点集合
  - 实现：实现教程上的伪代码即可

**Algorithm 3.2:** Algorithm for computing the dominance frontier of each CFG node.

```
1 for (a, b) ∈ CFG edges do
2     x ← a
3     while x does not strictly dominate b do
4         DF(x) ← DF(x) ∪ b
5         x ← immediate dominator(x)
```

- 建立 CallTree 函数调用树
  - 目的：后端生成目标代码时，服务于 CallInstr 函数调用，具体目的在活跃变量分析部分剖析，此处先谈实现
  - 实现：以函数为单位，某函数内所有 CallInstr 的 Target Function（目标函数），都为该函数的 CalledFunction，求解函数调用闭包 ClosureCalledFunc，BFS 基于 CallTree，即：被调用的函数所调用的函数也包含在该函数调用闭包中

## Mem2Reg

> 非 SSA 的形式转化到含有 φ 函数的 SSA 形式

目的：在程序设计中，内存变量是一种常见的数据类型，在进行编译优化时，内存变量可能会成为一个障碍。因为内存变量的值可能会在不同的程序路径中发生变化，这就使得程序的数据流分析和优化变得更加困难。Mem2reg 技术的主要目的是将内存变量转换为静态单赋值（SSA）形式的寄存器变量。通过这种转换，Mem2reg 可以减少内存操作，简化数据流分析和优化过程，从而提高编译器的优化效率和程序的性能。主要地，内存访问是程序中的一个瓶颈，它需要较长的时间来完成。通过将内存变量转换为寄存器变量，Mem2reg 可以减少内存访问的次数，提高程序的性能。

- preprocess
    - 目的：收集所有 storeInstr 和 loadInstr，对于 storeInstr 我们视之为 defInstr， loadInstr 视之为 useInstr，同时收录所有定义块，此番操作为 insertPhi 服务
- insertPhi
    - 目的：欲将对内存变量的操作转变为对寄存器的操作，就需要知道该内存变量是从程序的哪个位置传来的，但是对于汇聚块，其内变量可能有多个来源，我们无法确定具体是哪一个，因此引入 φ 节点：根据来源基本块来决定给这个变量赋什么值
    - 实现：以基本块中的 Alloca Int 的 AllocaInstr 为基本单位，我们需要对该指令的汇聚块（经过分析我们知道该指令所在块的 DF 支配边界即为该指令可达的汇聚块）执行插入 φ 的操作

**Algorithm 3.1:** Standard algorithm for inserting $\phi$-functions

```
1  for v: variable names in original program do
2      F ← {}                                   ▷ set of basic blocks where φ is added
3      W ← {}                                   ▷ set of basic blocks that contain definitions of v
4      for d ∈ Defs(v) do
5          let B be the basic block containing d
6          W ← W ∪ {B}
7      while W ≠ {} do
8          remove a basic block X from W
9          for Y: basic block ∈ DF(X) do
10             if Y ∉ F then
11                 add v ← φ(...) at entry of Y
12                 F ← F ∪ {Y}
13                 if Y ∉ Defs(v) then
14                     W ← W ∪ {Y}
```

对于某内存变量的每个定义点所在的基本块，找到该基本块的支配边界，由于它的支配边界不由该基本块支配，即有可能来自于其他基本块，因此要对该块插入 φ 指令，目前只是插入空 φ，此后rename（重命名）会填充 φ

- rename
    - 目的：因为要删去 AllocaInstr， StoreInstr， LoadInstr，所以需要更新原本使用了这些值的指令
    - 实现：注意：当前操作的最外层仍然是某变量的声明指令，当下操作都是对该声明点而言的。

        - 遍历支配树，遍历节点的每条指令：
            - 若为 LoadInstr，将所有使用该 LoadInstr 对应 Value 的指令，改为使用最新的 substituteValue，并删除 LoadInstr
            - 若为 StoreInstr，更新 substituteValue 为该指令即将写入内存的 Value ，删除 StoreInstr
            - 若为 PhiInstr，压栈
        - 扫描其在后继基本块，将最新的数据流信息（substitute），写入子节点中的 PhiInstr
        - BFS on DomTree，继续更新数据流信息
- conclusion
    - 简单来说，整个 Mem2reg 就是将 storeInstr 的来源值反向给到使用该loadInstr 的位置，期间需要维护一个最新来源值，最后将该值汇入数据流传播下去，即填入后继节点中可能存在的 phi 中，局部变量相当于一个管道，我们只是用它在 storeInstr 和 loadInstr 之间传值，所以这个管道是可以完全消除的，数据最终的源头只能是 Constant / Getint

## LoopAnalyze

> 循环分析

- 目的
    - GCM 需要给每个指令选择循环深度最小的基本块，故需进行循环分析，所谓循环深度就是该基本块参与了多少个循环圈，也就是 for 循环的层数
- 实现
    - 对基本块 X，遍历其支配集合，若其中的 Y 的直接支配块包含 X，则称 Y -> X 为一条回边，X->Y->X 构成循环，一旦判定构成循环，则循环内所有块的循环深度增加
    - 求解循环体：DFS on IDomTree，求解 start -> end 循环体，等价于求解 start + start.child -> end ，递归求解即可

## GVN

> Global Variable Numbering 全局值编号

GVN 寻找具有相同操作和相同输入的代数恒等式或指令，并为他们赋值为相同的编号。

- 目的：实现局部公共子表达式删除，以及为 GCM 服务
- 实现
    - 常量折叠
        - 双常量：直接计算
        - 单常量：a + 0, a - 0, a * 0, a * 1, a / 1, a % 1

- 乘法优化
  - 对于乘数或者被乘数，若为 2 的幂次，转化为移位
  - 否则转为移位 + 加法
- 除法优化
  - 对于被除数为常量且为 2 的幂次，转化为移位，但是需要警惕 -3 / -4 = 0，若移位不对负数考虑会出错
- GVNMap & GVNHash
  - GVN 为每个全局量标号，对于相同的编号，我们可以用已收录 GVNMap 的 Value 来更换该值，若没有收录则收录，显然这样做会出现某值未定义而被使用的情况，此情况可以被 GCM 纠正，所以 GVN & GCM 是绑定的
  - 标号的对象是 AluInstr，IcmpInstr，GEPInstr，CallInstr
  - 标号的实现方式多种，笔者采用运算数 + 运算符 的字符串作为全局标号，注意加法、乘法满足交换律。函数的标号则使用函数标签，但是需要首先确保该函数是可以被 GVN 的，需要保证参数不是指针，函数未修改或读取全局变量，没有调用其他函数。
  - GVN 进行标号并且全局替换数值的操作应 BFS on IDomTree

## GCM

> Global Code Motion 全局代码移动

- 目的
  - 修正 GVN 全局标号后出现的未定义先使用的情况
  - 循环不变式外提
- 实现
  - findPinnedInstr(确定固定点)
    - callInstr，retInstr，loadInstr，storeInstr，branchInstr，jumpInstr，moveInstr，IOInstr，phiInstr
    - 以上指令都为固定指令，不可以被移动
  - scheduleEarly(代码前提)
    - 尽可能将指令前移，但是不能超过其所用操作数的前移边界
    - 对于 PinnedInstr 可以直接确定前提块，即其所在块
    - 对于非 PinnedInstr 和 PinnedInstr 的 operand(操作数) 都可以再移动
    - 移动方式是递归，为尽可能将指令前移，先前移其操作数，再移动该指令：
      - 除了 Pinned 指令外，所有指令都被前移到第一个块

        - 第一次调整是对于使用到了 Pinned 指令值的指令，会调整他的块到 Pinned 指令的块
        - 之后的每次调整，既可能因为操作数本身为 Pinned 指令，也可能因为该操作数受到 Pinned 指令影响，从而间接影响该指令应该放在的块
        - 本质上，每个指令的前移块，是由其所涉及到的深度最大的 Pinned 指令决定的
        - 当然需要提前求解每个块在直接支配树中的深度，BFS on the IDomTree
    - scheduleLate(代码后移)
        - 尽可能将指令后移，但不能超过其 User 的后移边界
        - 处理方式同代码前移，先将其 User 后移
        - 目标后移块是该指令所有 User 的 Least Common Ancestor(LCA 最近公共祖先)
            - 特别的，对 PhiInstr，则需要求解 PhiInstr 各来源块的 LCA
    - selectBlock(选择块)
        - 选择区间是后移块到前移块路径上的所有块。选择方向是从后移目标块向前搜索，寻找循环深度最小的块作为目标块，直至到达前移块边界
        - 倒序遍历，当循环深度相同时，我们选择支配树深度更大的块作为插入块，即我们将其尽量后移。原因是避免 GCM 将本可能不会执行的分支中的代码提前
    - insert(插入)
        - 第一插入原则：如果目标块内有该指令的 User，则指令插入到该 User 之前，由于指令是顺序排列的，所以插入第一个 User 之前既可保证插入到所有 User 之前
        - 第二插入原则：如果目标块内有该指令的 operand，则将该指令插在最后一个 operand 之后
        - 第一插入原则优先

## ActivenessAnalyze

> 活跃变量分析

- 目的
    - 服务于寄存器分配
- 实现
    - 块级活跃变量分析：按照理论课讲解的定义求解：OUT = Σ successor's IN， IN = Use + OUT - Def，而 Def 和 Use 对每个块是确定且固定的，我们可以提前求解，无需每次用到才求解。之后开启循环，只要有一个块的 IN/OUT 发生了变化，就继续进行 IN/OUT 分析
    - 指令级活跃变量分析：同块级异曲同工，可以将每条指令视作一个基本块，Use 就是指令的操作数，Def 就是指令本身，IN = Use + OUT - Def，得到的即为该指令之前活跃的变量，求解指令级

活跃变量更多的原因是：对于 CallInstr 调用函数之前，需要由调用者自身保存已经使用的寄存器，此处可以优化为：只保存该 CallInstr 指令调用之前活跃的活跃变量所在寄存器，但是，这些被保存的寄存器当中可能有某些寄存器，被调用函数也没有用到，相当于白压栈，所以可以进一步可以优化为：调用方 CallInstr 之前活跃变量所占有寄存器和被调用函数的闭包函数集所用到的所有寄存器的交集

### RegAlloc

> 寄存器分配

- 目的
  - 真正的实现与内存的脱离，尽可能找到最优的分配方案，将变量存储在寄存器
- 实现
  - 简易图着色算法
  - 建立全局寄存器池：收录 $t0 - $t9, $s0 - $s7, $gp, $fp, $v1，而临时寄存器只有两个：$k0, $k1
  - 记录每条指令的每个操作数最后一次被使用到的地方，对于最后一次被使用的操作数，如果该基本块的 OUT 中并不含有该操作数，并且为该变量分配了寄存器，则可以释放掉该寄存器，留给下一个基本块使用，并记录该贡献出寄存器的变量 distribute
  - 对于可以被使用的定义点，尝试分配寄存器。如果临时寄存器池有空闲寄存器，则分配空闲寄存器。否则从中随机取出一个寄存器分配，并抛弃原 Value 对寄存器的使用，所以该值是不会被分配寄存器的，只会在栈上开辟空间。即笔者将寄存器尽量分配给位置靠后的定义点
  - 为直接支配节点分配寄存器：对其中某一个直接支配节点进行如下分析：如果当前基本块所占用的寄存器对应的变量，并没有出现在直接支配节点的 IN 中，也就是该变量没有到达此节点，可将该变量所占有的寄存器暂时释放，并予以该节点分配使用，记录被释放的关系到 releaseMap
  - DFS on the IDomTree，为子节点分配寄存器，之后恢复 releaseMap 中寄存器与变量分配对应关系，避免对子节点的兄弟节点产生影响
  - 整个函数应该以递归为分界来看，递归调用之前，就是这个块的寄存器分配和释放，递归调用之后，需要恢复本块分配之前的寄存器映射信息
  - 准备回到父节点，占有寄存器的定义点释放寄存器，因为分配函数为递归函数，所以该函数在返回的时候，需要恢复父节点的分配情况，在父节点这些定义点是没有分配的寄存器的
  - 将后继不再使用但是是从前驱基本块传过来的变量对应的寄存器映射恢复回来，也就是在 distribute 中，但是不是在当前基本块定义的变量

## 后端优化

### RemovePhi

> 消除 phi 指令

φ 指令无法被常规的汇编指令表示，代码生成前的最后一步需要将 SSA 形式的代码转换为汇编指令集能够接受的代码

- Phi2ParallelCopy
    - 删除 φ 指令，添加 ParallelCopy 指令
    - 对于每个基本块 X，遍历其前驱，若前驱 Pre 只有 X 一个后继，则将 ParallelCopy 插入 Pre 的跳转指令之前即可；如果 Pre 有多个后继，需要建立空的中间块 MidBasicBlock，并向其中插入 ParallelCopy 指令，然后修改 Pre 和 Mid 的跳转关系，并添加 Pre 和 Mid， Mid 和 X 的前驱后继关系
    - 遍历基本块的所有 φ，将每个 φ 中的 option 放入各个对应的 ParallelCopy 指令中，option 本就是按照前驱顺序排列的，故按照对应关系直接填充 ParallelCopy 即可，最后删除 φ
- ParallelCopy2Move
    - 删除 ParallelCopy 指令，添加 Move 指令
    - 实现
        - 将 ParallelCopy 转化为一组 MoveInstr 并插入基本块的指令序列中，插入点为指令集末尾，跳转之前，此处我们需要解决一个关键问题：并行赋值
            - 并行赋值（形式）
                - 对于 MoveInstr X，检查 X 之后的每一条指令 Y，若 X.Dst = Y.Src = Value，则存在并行赋值，如果出现了并行赋值，我们需要增加中间变量 midValue，将所有使用 Value 作为 Src 的 MoveInstr，改为使用 midValue 作为 Src，并添加 Value -> midValue 的 MoveInstr，并将该指令添加到 moveInstrList 的开头，相当于赋初值，保证了旧值不会被覆盖，利用串行实现并行赋值
            - 寄存器冲突（物理）
                - 逆序遍历 MoveInstrLst，对于每条 MoveInstr 指令 X，检查其前的所有指令 Y，若 X.Src.Reg = Y.Dst.Reg，发生寄存器冲突，添加中间变量 MidValue，将所有使用 value 作为 Src 的 Move 指令，将改为使用 MidValue 作为 Src，并添加 Value -> midValue 的 MoveInstr，此前的“并行赋值”只是解决了形式上的并行赋值问题，寄存器冲突要求我们实现物理层面的并行赋值。处理方式与上述如出一辙，仍然是通过添加临时变量，为临时变量赋初值实现的
        - 删除 ParallelCopy
- conclusion
    - 消phi之后其实是不符合SSA的，有 φ 的基本块会消去 φ，本质是将对 φ 的赋值前移到了各个 pre 块中，所以LLVM并没有继续做下去，而是将消phi留给了后端

## RequeueBlock

> 汇编块排序

受限于 CFG 流图对于基本块的切割，每个基本块的末尾指令只能是终端指令：RetInstr / BrancnInstr / JumpInstr，而实际上在汇编程序是没有基本块概念的。严格按照中间代码生成的目标代码，可能会出现大量的

无效跳转：即前一个基本块的跳转目标块是紧邻的后一个基本块，此时我们可以删去前一个基本块末尾的跳转指令。笔者此处通过对汇编块的排序来增加这样去掉跳转的机会，后续窥孔优化删除无效跳转指令

- 目的：汇编块排序，增加去除跳转的机会
- 实现：遍历汇编块，对汇编块 X，将其跳转的目标块 Y 紧邻 X 的后方放置

## PeepHole

> 窥孔

实践证明效果相对明显的窥孔优化是：删除无效跳转和同地址存取

- 删除无效跳转
    - 实现：承接 RequeueBlock，跳转目标块是紧邻后继块，删除该跳转
- 同地址存取
    - 目的：前为 sw，后为 lw 且二者访问同一地址的情况，可以用 move 指令代替 lw，减少访存开销，但是仍然要保留 sw