编译优化

Serval | 2024 编译技术

2024 年 11 月 15 日

SSA 形式

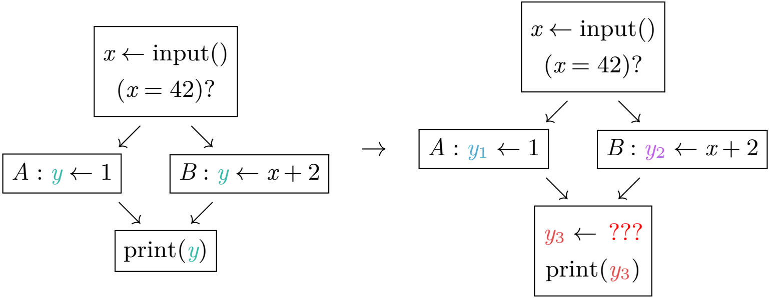
- 静态单赋值（Static Single Assignment）形式：每个变量在程序中仅被赋值一次。
- 变量的值与引用变量的位置无关。
- 对于简单的控制流：

$$\boxed{\begin{array}{l} x \leftarrow 1 \\ y \leftarrow x + 1 \\ x \leftarrow 2 \\ z \leftarrow x + 1 \end{array}} \xrightarrow{\text{重命名 } x} \boxed{\begin{array}{l} x_1 \leftarrow 1 \\ y \leftarrow x_1 + 1 \\ x_2 \leftarrow 2 \\ z \leftarrow x_2 + 1 \end{array}}$$

变量 x 分别重命名为 x_1，x_2 之后即是 SSA 形式。

控制流

- 控制流图（Control Flow Graph）描述了程序执行的流程。
- 分支汇聚时无法为变量定值：



ϕ 函数

- 为了解决这个问题，引入 ϕ 函数。ϕ 函数根据跳转前的基本块选择返回值。例如：

$$x \leftarrow \mathrm{input}()$$
$$(x = 42)?$$

$$A: y_1 \leftarrow 1 \qquad B: y_2 \leftarrow x + 2$$

$$y_3 \leftarrow \phi(A: y_1, B: y_2)$$
$$\mathrm{print}(y_3)$$

从基本块 A 跳转而来时，y_3 取 y_1 的值；
从基本块 B 跳转而来时，y_3 取 y_2 的值。

phi 指令

- LLVM IR 中的 phi 指令对应 ϕ 函数。

```
define i32 @func(i32 %0) {
1:
  %2 = icmp eq i32 %0, 42
  br i1 %1, label %3, label %4
3:
  br label %6
4:
  %5 = add i32 %0, 2
  br label %6
6:
  %7 = phi i32 [1, %3], [%5, %4]
  ret i32 %7
}
```

- 需要注意：
  1. phi 指令只能放置在基本块开头。
  2. phi 指令是**并行执行**的。也就是说，phi 指令的返回值取决于进入当前基本块时的状态。

mem2reg：内存引用提升为寄存器引用

参考资料：The SSA Book，Chapter 2 & 3

- 将仅被 `load` 指令与 `store` 指令引用的 `alloca` 指令提升为虚拟寄存器。
- mem2reg 分为以下两个步骤：

1. **插入 ϕ 函数**：对于一个变量 v，原有的 v 的定义和新插入的 ϕ 函数将程序分割为多个活跃区间。每个活跃区间以 v 的定义开始，并且仅有这个定义能到达活跃区间内 v 的引用。
2. **变量重命名**：对于变量的每个活跃区间，重命名变量的所有定义与引用。

插入 ϕ 函数 (1)： Join node

- 基于之前的讨论，可以直接在有超过一个前驱的基本块中插入 ϕ 函数。但这样插入了过多冗余的 ϕ 函数。
- 为了确定哪些基本块必须插入 ϕ 函数，引入 join node 的概念：
  - 对于两个不同的基本块 n_1, n_2，以及基本块 n_3（可能与 n_1 或 n_2 相同，也可能不同），
  - CFG 中存在从 $n_1 \to n_3$ 的路径与 $n_2 \to n_3$ 的路径，并且路径至少经过一条 CFG 边，
  - 这两条路径仅有 n_3 一个交点，
  - 则称 n_3 是 n_1 与 n_2 的 join node。
- 两个基本块 n_1, n_2 的 join node 不一定只有一个。

插入 ϕ 函数 (2)：Join set

- 对于同一变量 v 两处不同定义所在的两个基本块，二者的 join node 必须插入 ϕ 函数。
- 变量 v 可能在多个基本块都有定义，记这些基本块构成的集合为 D_v，D_v 中任意两个不同基本块的 join node 都必须插入 ϕ 函数。
- 因此引入 join set 的概念：
  - 对于基本块构成的集合 S，S 的 join set 为 S 中任意两个不同基本块的 join node 构成的集合，记作 $\mathscr{J}(S)$。
- 对于变量 v，$\mathscr{J}(D_v)$ 需要插入 ϕ 函数。
- 注意到插入的 ϕ 函数也是 v 的定义，因此 $\mathscr{J}(D_v \cup \mathscr{J}(D_v))$ 也需要插入 ϕ 函数。
- 实际上 $\mathscr{J}(D_v \cup \mathscr{J}(D_v)) = \mathscr{J}(D_v)$，因此求出 $\mathscr{J}(D_v)$ 即可。

插入 ϕ 函数 (3)：支配关系

- 迭代方法计算 join set 并不实用。为了更高效地计算 join set，考察 SSA 形式的性质：定义（结点）是引用（结点）的必经结点。
- 具体来说，对于 SSA 形式中的一个变量 v:
  - 如果 v 被基本块中的一个 ϕ 函数 $\phi(n : v, \cdots)$ 引用，那么 v 的定义结点是 n 的必经结点。
  - 如果 v 被基本块中不是 ϕ 函数的语句引用，那么 v 的定义结点是这个基本块的必经结点。
- 其中基本块 x 是基本块 y 的必经结点是指：
  - 若 CFG 中从入口到基本块 y 的所有路径都经过 x，则称 x 是 y 的必经结点，或者称 x 支配（dominate）y。
  - 若 x 支配 y，且 $x \neq y$，则称 x 严格支配（strict dominate）y。

插入 ϕ 函数 (4)：支配树

- 如果 x 严格支配 y，则让 x 成为 y 的祖先。通过这种方式能让支配关系唯一地确定一个树状结构，这即是支配树。支配树的根结点是 CFG 图的入口结点。
- 接下来给出直接支配者的定义：
  - 若 x 严格支配 y，并且不存在被 x 严格支配同时严格支配 y 的结点，则称 x 是 y 的*直接支配者*（immediate dominator）。
- 从支配树的角度考虑，结点 y 的直接支配者 x 即是 y 在支配树上的父结点。
  - 这是因为不存在 x 子树中的其他结点（等价于被 x 严格支配）是 y 的祖先（等价于直接支配 y），从而 x 与 y 在支配树上是直接相连的。

插入 ϕ 函数 (5)：支配关系计算

- 根据 CFG 图可以迭代计算支配关系：
  1. 设支配 x 的点集为 $\text{Dom}(x)$。
  2. 对于每个结点 x 初始化支配关系 $\text{Dom}(x) = \{x\}$。
  3. 对于每个结点 x，按照 $\text{Dom}(x) = \left(\bigcap_{\langle u,x\rangle \in \text{CFG}} \text{Dom}(u)\right) \cup \{x\}$ 迭代计算支配关系。
  4. 重复上一步直到所有结点的支配关系不再发生变化。
- 代码实现中可以使用 bitset 加速上述流程。
- 可以通过以下方式从支配关系得到支配树：
  - 注意到以下事实：若 x 严格支配 y，那么 $|\text{Dom}(x)| < |\text{Dom}(y)|$。
  - 因此对于结点 y，所有严格支配 y 的结点 x 中 $|\text{Dom}(x)|$ 最大的即是 y 的直接支配者，也即 y 在支配树上的父结点。
- Lengauer–Tarjan 算法可以更高效地计算支配树，因过于复杂此处不作展开。

插入 ϕ 函数 (6)：支配边界

- 回顾：对于 SSA 形式中的一个变量 v，其定义结点支配引用结点。
- 需要在定义结点恰好不能支配的结点插入 ϕ 函数。
- 因此给出*支配边界*（Dominance Frontier）的定义：
  - 若结点 n 不严格支配结点 x，但严格支配 x 的一个前驱结点，则 x 在 n 的支配边界 $DF(n)$ 中。
  - 结点 n 的支配边界 $DF(n)$ 是所有满足以上条件的点 x 构成的集合。
  - 结点集合 S 的支配边界 $DF(S) = \cup_{n\in S} DF(n)$。
- 回顾：插入的 ϕ 函数也是变量 v 的定义。因此需要迭代支配边界：
  - $DF_1(S) = DF(S), \qquad DF_{k+1}(S) = DF(S \cup DF_k(S))$
- D_v 支配边界迭代至不再变化的结果 $DF^+(D_v) = DF_{k\to\infty}(D_v)$ 满足 $DF^+(D_v) = \mathscr{J}(D_v \cup \{\text{entry}\})$。这即是我们需要的。

插入 ϕ 函数 (7)：支配边界计算

- 完成支配关系计算后可以按照以下方式计算支配边界：

Algorithm 3.2, The SSA Book

for $\langle a, b\rangle \in$ CFG 图的边集 **do**
  $x \leftarrow a$
  while x 不严格支配 b **do**
    $DF(x) \leftarrow DF(x) \cup b$
    $x \leftarrow x$ 的直接支配者

- 在上述伪代码中：
  - 先枚举 DF 中的点 b 以及其被严格支配的前驱 a 之间的边 $\langle a, b\rangle$。
  - 对于严格支配 a 的点 x，$DF(x)$ 包含点 b，故枚举 x 并将 b 加入 $DF(x)$ 中。

插入 ϕ 函数 (8)

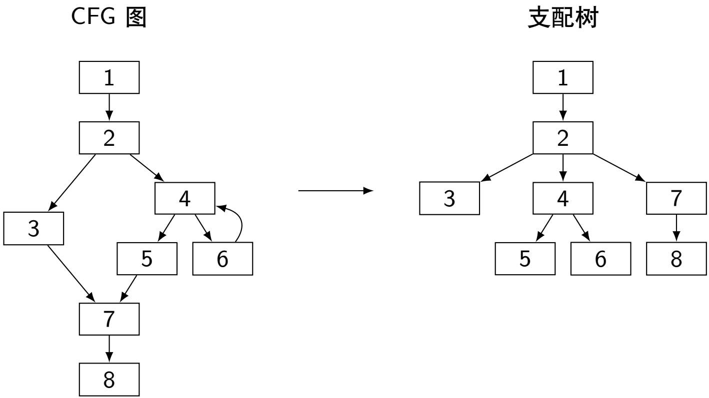
- 以下是插入 ϕ 函数的算法：

Algorithm 3.1, The SSA Book

```
for v : 原程序中的变量 do
    F ← {}                                        ▷ 已插入 φ 函数的基本块集合
    W ← {}               ▷ v 的定义（包括新插入的 φ 函数）所在的基本块集合
    for d ∈ v 的定义 Defs(v) do
        W ← W ∪ {d 所在的基本块}
    while W ≠ {} do
        从 W 中选择并删除一个基本块 X
        for Y : DF(X) 中的基本块 do
            if Y ∉ F then
                在基本块 Y 的入口处插入 φ 函数 v ← φ(...)
                F ← F ∪ {Y}
                if Y ∉ Defs(v) then
                    W ← W ∪ {Y}
```

- 对于变量 v，W 初始化为 D_v，并不断与 $DF(W)$ 合并。最终 $DF^+(D_v)$ 中的结点都会被 Y 遍历。

插入 ϕ 函数 (9)：示例



- $DF(4) = \{4, 7\}$

变量重命名 (1)

- 插入 ϕ 函数后，变量 v 的一个活跃区间包含：
  - 位于活跃区间开头的 v 的唯一一个定义，
  - 以及活跃区间中 v 的引用。
- 在这一步骤，需要对一个活跃区间重命名变量的定义与所有引用。
- 具体来说，变量重命名时对于定义和引用分别需要：
  - 在遍历到变量的定义时创建新变量。
  - 对于变量的每处引用，找到支配这处引用的定义，并将引用替换为所创建的新变量。
- 变量重命名需要利用之前求出的支配树。

变量重命名 (2)

- 接下来给出变量重命名的伪代码：

Algorithm 3.3, The SSA Book

for v : 原程序中的变量 **do**
  v.reachingDef ← null ▷ 变量的到达定义（支配当前位置的定义）
for BB : 前序遍历支配树 **do**
  for i : 基本块 BB 中的指令 **do**
    for v : 非 ϕ 函数 i 引用的变量 **do**
      updateReachingDef(v, i)
      用 v.reachingDef 替换 v 的这处引用
    for v : i 定义的变量 **do**
      updateReachingDef(v, i)
      创建新变量 v'
      用 v' 替换 v 的这处引用
      v'.reachingDef ← v.reachingDef
      v.reachingDef ← v'
  for ϕ : BB 后继结点中的 ϕ 函数 **do**
    for v : ϕ 中引用的变量 **do**
      updateReachingDef(v, ϕ)
      用 v.reachingDef 替换 v 的这处引用

变量重命名 (3)

- 伪代码中调用的 updateReachingDef 函数如下：

updateReachingDef(v, i), The SSA Book

输入：v : 程序中的变量
输入：i : 程序中的一条指令
$r \leftarrow v$.reachingDef
while not (r 是空指针，或者 r 的定义支配 i) **do**
  $r \leftarrow r$.reachingDef
v.reachingDef $\leftarrow r$

- updateReachingDef 函数维护变量 v 的 reachingDef，没有返回值。
- 变量重命名是在前序遍历支配树，updateReachingDef 是在支配树上回溯。

消除 ϕ 函数

参考资料：The SSA Book，Chapter 3 & 21

- LLVM IR 中有 phi 指令，但 MIPS 中没有直接实现 ϕ 函数的指令，因此从 LLVM IR 生成 MIPS 时需要消除 ϕ 函数。
- 消除 ϕ 函数主要分为以下两个步骤：
  1. 将 ϕ 函数替换为*并行复制*（parallel copy, pc）指令。
  2. 并行复制指令串行化，也即替换为一系列 move 指令。
- pc 指令是抽象出来的指令，并不在 MIPS 指令集中。
- 之所以需要 pc，是因为 ϕ 函数的语义是并行执行的。
- 与 phi 指令不同，pc 指令插入到前驱基本块的末尾，而在 mem2reg 中 phi 指令插入到后继基本块的开头。

关键边

- 如果前驱基本块有多个后继，那么所有后继中 ϕ 函数对应的 pc 指令都会插入到前驱基本块的末尾。但程序执行时只会沿 CFG 中的一条边执行，其余后继产生的 pc 指令不应当被执行。
  - 其余 pc 指令不仅冗余，而且执行其余 pc 指令可能导致程序语义发生变化。
- 对于 CFG 中的边 $\langle u, v\rangle$，如果前驱基本块 u 有多个后继，并且后继基本块 v 有多个前驱，那么这条边是*关键边*（critical edge）。
  - 没有关键边的 SSA 形式称为*边分割的 SSA 形式*。
- 对于有多个后继的基本块，在其每条出边上新建一个基本块可以移除 CFG 中的所有关键边。

关键边分割

- 以下是分割关键边，并将 ϕ 函数替换为 pc 指令的算法：

Algorithm 3.5, The SSA Book

for B : CFG 中的基本块 **do**
  for $E_i = \langle B_i, B\rangle$: 基本块 B 的入边 **do**
    PC$_i \leftarrow$ 空的 pc 指令
    if B_i 有多条出边 **then**
      创建新的基本块 B_i'
      将边 $E_i = \langle B_i, B\rangle$ 替换为边 $\langle B_i, B_i'\rangle$ 与边 $\langle B_i', B\rangle$
      将 PC$_i$ 加入 B_i' 中
    else
      将 PC$_i$ 加入 B_i 末尾
  for $a_0 = \phi(B_1 : a_1, \ldots, B_n : a_n)$: 基本块 B 中的 ϕ 函数 **do**
    for a_i : ϕ 函数中对应各个基本块 B_i 的参数 **do**
      将 PC$_i$ 指令置为 $a_i' \leftarrow a_i$
    删除该 ϕ 函数 $a_0 = \phi(B_1 : a_1, \ldots, B_n : a_n)$

- 在枚举入边的循环中分割关键边。
- 枚举 ϕ 函数的循环将 ϕ 函数替换为各个前驱基本块末尾的 pc 指令。

pc 指令串行化 (1)

- 方便起见，称 pc 指令 $b \leftarrow a$ 中的 b 为目标寄存器，a 为源寄存器。
- 将 pc 指令涉及的虚拟寄存器看作有向图的结点，pc 指令 $b \leftarrow a$ 视为结点 b 依赖结点 a，因为 b 需要获得 a 的值。
- 每次只能选择目标寄存器 b 未被依赖的 pc 指令 $b \leftarrow a$，将其改写为 move 指令。
- 否则所选择的目标寄存器 b 被其他结点（例如 c）依赖，存在其他 pc 指令 $c \leftarrow b$ 需要使用目标寄存器的值。$b \leftarrow a$ 改写为 move 指令后，寄存器 b 的值会丢失。
- 若依赖关系成环，可以选择一个环上的寄存器 a，将其复制为 a' 实现破环。

pc 指令串行化 (2)

- 以下是将 pc 指令转化为串行 move 指令的算法：

Algorithm 3.6, The SSA Book

$pcopy \leftarrow$ 需要串行化的 pc 指令
$seq \leftarrow ()$ ▷ *pc 指令串行化后的 move 指令序列*
while $pcopy$ 中存在 $a \neq b$ 的指令 $b \leftarrow a$ **do**
  if 存在指令 $(b \leftarrow a) \in pcopy$ 使得不存在 $(c \leftarrow b) \in pcopy$ **then**
    将 $b \leftarrow a$ 加入到 seq 末尾，并从 $pcopy$ 中删除 $b \leftarrow a$
  else
    从 $pcopy$ 中选择 $a \neq b$ 的指令 $b \leftarrow a$
    新建寄存器 a'
    将 $a' \leftarrow a$ 加入到 seq 末尾
    将 $pcopy$ 中的 $b \leftarrow a$ 替换为 $b \leftarrow a'$

  - 每次选择目标寄存器未被依赖的 pc 指令改写为 move 指令，对应 **if** 分支。
  - 否则依赖图中存在环，选择某个结点拆点破环，对应 **else** 分支。

死代码删除 (1)

- 死代码删除（Dead Code Elimination）能有效减少程序体积。
- 认为变量是无用的，除非它们被证明是活跃的。具体来说：
  - 对于涉及控制流或有副作用的指令，指令引用的变量是活跃的。例如：`br`、`call`、`store`、`ret` 等指令。
  - 对于为活跃变量定值的指令，指令引用的变量是活跃的。
- 以函数为单位遍历所有指令，维护指令和变量的活跃标记，并迭代更新活跃标记。对于无用的变量，可以删除定义它的指令。
- 修改程序的同时需要维护定义–使用关系。

死代码删除 (2)

- 从函数入口出发无法到达的基本块是不可达的。从程序入口出发无法到达的函数是不可达的。
- 根据调用关系从 `main` 函数出发遍历函数，可以求出并删除不可达的函数。
- 在 CFG 上从入口基本块出发遍历基本块，可以求出并删除不可达的基本块。
  - 这一步在 mem2reg 之前分析支配关系时可能已经完成了。
  - 分析支配关系时可以发现，入口基本块不支配不可达的基本块。
- CFG 发生变化之后，可以再次分析并删除不可达代码。
  - 在常量折叠或常量传播之后，如果跳转条件被证明为常量，那么可以将条件跳转改写为无条件跳转。类似情形会改变 CFG 中的点或边。

函数内联

- 函数调用的调用方称为 caller，被调用方称为 callee。
- 如果 callee 不包含递归并且较为简单，那么可以将 callee 的代码展开至 caller 中。这即是函数内联。
- 函数内联可以节省函数调用时的开销，并且内联后 callee 可以获得 caller 传入的参数，有助于后续的常量传播等优化。
- 为什么不展开所有不包含递归的函数调用？
  - 考虑以下情形：有函数 $f_1, \ldots, f_n$，f_i 调用两次 f_{i+1} $(1 \leq i \leq n)$。
  - 展开这些函数调用会产生巨大开销。
- 实现函数内联时，由于函数可能有多处 `return`，可以借助 ϕ 函数传递参数与返回值。

乘法优化

- 乘法指令时间开销大于加法、移位等指令。
- 变量与常数的乘法，视情况可以优化为移位或加法。例如：

$$x \times 2^n \rightarrow x \ll n$$

$$x \times (2^n \pm 2^m) \rightarrow (x \ll n) \pm (x \ll m)$$

  - 与常数 2^n 相乘，优化后仅需一条移位指令。
  - 与常数 $2^n \pm 2^m$ 相乘，优化后需要两条移位指令与一条加法指令。
- 在计算地址偏移时常常需要乘法优化。

除法优化 (1)

参考资料：Division by Invariant Integers using Multiplication

- 除法指令时间开销远大于其他指令。
- 除数为常数的除法可以改写为乘法。具体来说有以下定理：

Theorem

设 m, d, ℓ 是非负整数，$d \neq 0$ 且 $2^{N+\ell} \leq m \times d \leq 2^{N+\ell} + 2^{\ell}$。则对于 $0 \leq n < 2^N$ 的整数 n 有 $\lfloor n/d \rfloor = \lfloor m \times n/2^{N+\ell} \rfloor$。

- 可以依次枚举 ℓ，验证 $2^{N+\ell} \leq m \times d \leq 2^{N+\ell} + 2^{\ell}$ 是否成立，得到一组合法的 m, ℓ。
- 由于是向零取整的有符号除法，在计算 m, ℓ 时 N 需要代入 $N-1 = 31$，并且 d 需要取绝对值。

除法优化 (2)

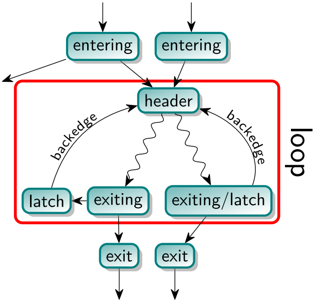
- 在求出 d 对应的 m, ℓ 之后，$d > 0$ 的情形按照以下公式可以改写为乘法：

$$n/d = \begin{cases} n & , d = 1 \\ \mathsf{SRA}(n + \mathsf{SRL}(\mathsf{SRA}(n, k-1), 32-k), k) & , d = 2^k \\ \mathsf{SRA}(\mathsf{MULSH}(m, n), \ell - 1) + \mathsf{SRL}(n, 31) & , m < 2^{31} \\ \mathsf{SRA}(n + \mathsf{MULSH}(m - 2^{32}, n), \ell - 1) + \mathsf{SRL}(n, 31) & , \text{otherwise} \end{cases}$$

  - $\mathsf{SRA}(x, n)$ 代表 x 算数右移 n 位，对应 `sra` 指令。
  - $\mathsf{SRL}(x, n)$ 代表 x 逻辑右移 n 位，对应 `srl` 指令。
  - $\mathsf{MULSH}(x, y)$ 代表 x 与 y 有符号乘法的高 32 位，对应 `mult` + `mfhi` 指令。
- $d < 0$ 的情形按照 $|d|$ 改写后对计算的结果取反即可。

循环

- 在 LLVM 中，*循环*是 CFG 中满足以下条件的点集：
  1. 点集的导出子图（点集本身，以及端点都在点集中的边）是强连通的（任意两点互相可达）。
  2. 存在某个结点支配点集中的所有点。这个点称为 header。
  3. 点集是极大的。再向点集中添加点会破坏强连通性，或是使得 header 变为其他点。
- 进入 header 前可能会有出边唯一的 preheader。
- 循环中有出边指向 header 的结点称为 latch，这条出边称为 back edge。



循环分析

- 完成支配关系分析之后，可以通过 back edge 确定循环：
  - 对于 CFG 中的一条边 $n \to h$，如果 h 支配 n 则这条边是 back edge，对应以 h 为 header 的循环。
- 根据定义不难发现，一个结点最多只能作为一个循环的 header。若有多条 back edge 指向同一个结点，这些 back edge 均属于以这个结点为 header 的一个大循环。
- 对于分别以 a, b （$a \neq b$）为 header 的两个循环 A, B，如果 B 的结点是 A 的结点的真子集，则称 B 嵌套在 A 内部，或者称 B 是内层循环。
- 循环之间的嵌套关系形成*循环嵌套树*（loop-nest tree）。
  - 可以将整个函数看作位于循环嵌套树根结点的伪循环。

循环嵌套树

- 可以按以下方式建立循环嵌套树:
  1. 基于 CFG 计算支配关系并建立支配树。
  2. 遍历 CFG 找出所有 back edge $n \to h$，并标记对应的 header h。
  3. 对于每个 header h 找出对应循环内的结点。
     对于 back edge $n \to h$，从 h 到 n 的所有可能路径均在以 h 为 header 的循环内。
  4. 确定每个循环在循环嵌套树上的父结点。
     若循环 B 嵌套在循环 $A_1, \ldots, A_k$ 中，包含结点数最少的循环 A_i 即是 B 在循环嵌套树上的父节点。
- 循环嵌套树可以确定循环的深度。GVN & GCM 会使用循环分析的结果。

循环优化

- 循环不变量外提
  - 对于循环中的定值，如果操作数是常数、或是循环不变量、或定值在循环之外，那么定义的变量是循环不变量。
- 循环变量归纳
  - 循环中仅有定值 $i \leftarrow i + c$ 的变量 i 是归纳变量，可以求出第 k 轮循环中 i 的值为 $c \times k + c'$，其中 c, c' 均为循环不变量。
  - 参考现代编译原理（C 语言描述）一书的 18.3 节。
- 循环展开
  - 可以展开循环次数较小的循环以便后续优化。
  - 实验中计算周期数时不会考虑指令并行。
- 循环合并

LVN, GVN

参考资料：Global Code Motion Global Value Numbering

- 局部值编号（Local Value Numbering）通过哈希寻找基本块内的等价指令，合并这些等价指令定义的变量。
- 具体来说，在处理一条指令 $d \leftarrow op(a_1, a_2, \ldots)$ 时：
  - 若指令计算结果是常数，则将 d 的所有引用替换为这个常数。这实际上就是常量折叠。
  - 若指令计算结果与操作数 a_i 相同，则将 d 的所有引用替换为 a_i。
  - 若指令计算值与其他指令 $d' \leftarrow op(\ldots)$ 相同，则将 d 的所有引用替换为 d'。
- 全局值编号（Global Value Numbering）与 LVN 类似，但不局限于基本块内部，而是寻找函数内部的等价指令。
- GVN 可能会导致定义在引用之后，需要全局代码移动（Global Code Motion）予以纠正。

GCM

- 全局代码移动（Global Code Motion）分为以下步骤：
  - 计算支配关系并建立支配树。
  - 基于循环分析求出各个基本块的循环深度。
  - Schedule Early：求出每条指令能够被调度的最早的基本块。
  - Schedule Late：求出每条指令能够被调度的最晚的基本块。
  - Select Block：在 Early 与 Late 确定的区间内选择合适的基本块。
- GCM 选择基本块的依据是循环深度尽可能浅，并且尽可能靠前。
- 算法具体细节与伪代码参考原论文。

图着色寄存器分配

参考资料：现代编译原理（C 语言描述），Chapter 11

- 寄存器分配可以被规约为冲突图的着色问题。
- 图着色寄存器分配主要分为以下阶段：
  1. 构造：分析数据流并构造冲突图。
  2. 简化：删除低度数结点，简化冲突图。
  3. 合并：保守合并，减少 `move` 指令。
  4. 冻结：选择结点，放弃与之相关的潜在合并。
  5. 溢出：选择高度数结点在栈上保存，降低其余结点度数。
  6. 选择：对虚拟寄存器指派颜色（物理寄存器）。
  7. 重新开始：对无法着色的虚拟寄存器改写程序，重新计算。
- 伪代码可以参考*现代编译原理（C 语言描述）*一书的 11.4 节。

窥孔优化

- 伪指令对应一条或多条基本指令。
- 生成汇编代码时选择基本指令，或是将伪指令改写为基本指令，可降低程序运行的周期数。
- 可以在汇编代码中使用异或等运算。例如：

```
slt $t0, $s1, $s0
ori $at, $zero, 1
subu $t0, $at, $t0
```

←

```
sle $t0, $s0, $s1
```

→

```
slt $t0, $s1, $s0
xori $t0, $t0, 1
```

  - 伪指令 sle 直接展开需要三条指令。
  - 可以使用 xori 改写为两条指令。

基本块重排

- 如果基本块 B_1 的后继仅有 B_2，也即 B_1 无条件跳转到 B_2，那么在汇编代码中将 B_1 与 B_2 相邻放置可以节省 B_1 的跳转指令。

```
B2:
  ...
B1:
  ...
  j B2
```

$\xrightarrow{\text{重排基本块}}$

```
B1:
  ...
B2:
  ...
```

- 如果 B_2 有多个前驱，只能节省其中一个前驱的跳转指令。
- 结合循环分析的信息，可以估计每条 CFG 边和基本块的执行的次数，优先选择执行次数更多的基本块减少跳转指令。

其他优化

- 尾递归优化
  - 将函数末尾直接调用自身的递归函数优化为循环。
- 激进的死代码删除（Aggressive DCE）
  - 循环中的变量会依赖其本身，因此 DCE 无法删除无用的循环。
  - ADCE 根据控制依赖关系标记活跃指令与变量，并删除无用的指令。
  - 参考现代编译原理（C 语言描述）一书的 19.5 节。
- 稀疏条件常量传播（SCCP）
  - 循环中变量可能依赖自身，简单的常量传播对此无效。
  - SCCP 乐观地认为变量是常数，直到有证据表明变量不是常数。
  - 参考 Constant Propagation with Conditional Branches。

Thanks!

Q & A