第 5 章

中端分析与优化

编译器的中端读入前端生成的程序的中间表示，对中间表示进行程序分析和程序优化，并进一步输出程序的后端表示供编译器后端进一步处理。编译器中端首先依赖控制流分析来理解程序的执行流，并建立合适的控制流数据结构。编译器使用数据流分析技术，来理解程序中值的流动信息，并依赖这信息进行程序优化。为了取得更好的优化效果，编译器不仅需要对函数内的代码做优化，还需要具有跨函数进行过程间程序分析和程序优化的能力。本章对编译器中端的主要理论和技术进行讨论，首先讨论理解程序执行流程的控制流分析；其次讨论刻画程序值流动的数据流分析，以及基于数据流分析结构的程序优化算法；再次讨论对于指针程序非常重要的别名分析和优化；接着讨论过程间程序分析和程序优化；最后讨论循环优化。

5.1 控制流分析

为实现编译优化，编译器必须对程序如何使用系统中的可用资源具有全局性的“理解”能力。编译器必须能刻画程序的控制流和程序对数据执行的操作的相关特征，以便删除未优化的无用代码，从而使低效的、较通用的操作被更高效的专用操作所替代。

当编译器编译程序时，它一开始读入的是程序的字符流。词法分析器将这些字符流转换为记号流，语法分析器从中进一步发现并检查语法结构。编译器前端输出的结果可以是抽象语法树或某种其他形式的中间代码。但是，不论这种中间表示是什么形式，它对程序做什么和怎样做仍然没有提供更多的信息。编译器使用控制流分析(control-flow analysis)来发现每一个函数内控制流的层次结构，并使用数据流分析(data-flow analysis)来确定与数据定值和使用有关的全局信息。

在讨论控制流分析和数据流分析中使用的技术之前，先讨论一个简单的例子。从图 5-1(a)的 C 程序开始，这个程序对给定的 m>0，计算第 m 个斐波那契数。对于给定的输入值 m，它检查该数是否小于或等于 1，当小于或等于 1 时返回该参数值；否则，它迭代计算过程直到得出数列的第 m 个成员，并返回此数。图 5-1(b)给出了这个 C 程序被转换为中间表示后的代码。

在分析这个程序时，第一个任务是发现它的控制流。就这个程序的源代码而言，其控制结构是显然的，函数体由一个 if-then-else 结构组成，并且在 else 部分含一个循环。但是在中间代码中，控制结构就不那么明显了。此外，循环也可能是用 if 和 goto 形成。因此，即便在源代码中，控制结构也可能不那么明显。为了使控制流分析方法更加直观，需要首先将程序转换为一种更形象的表示，即如图 5-2 所示的流程图。

```
unsigned int fib(m){
  unsigned int m;
  unsigned int f0 = 0,f1 = 1,f2,i;
  if (m <= 1){
    return m;
  }
  else {
    for(i = 2;i <= m;i++){
      f2 = f0 + f1;
      f0 = f1;
      f1 = f2;
    }
    return f2;
  }
}
```

(a) 计算斐波那契数的 C 程序

```
    receive m (val)
    f0 = 0
    f1 = 1
    if m <= 1 goto L3
    i = 2
L1 : if i <= m goto L2
    return f2
L2 : f2 = f0 + f1
    f0 = f1
    f1 = f2
    i = i + 1
    goto L1
L3 : return m
```

(b) C 程序的中间代码

图 5-1 C 程序和对应的中间代码

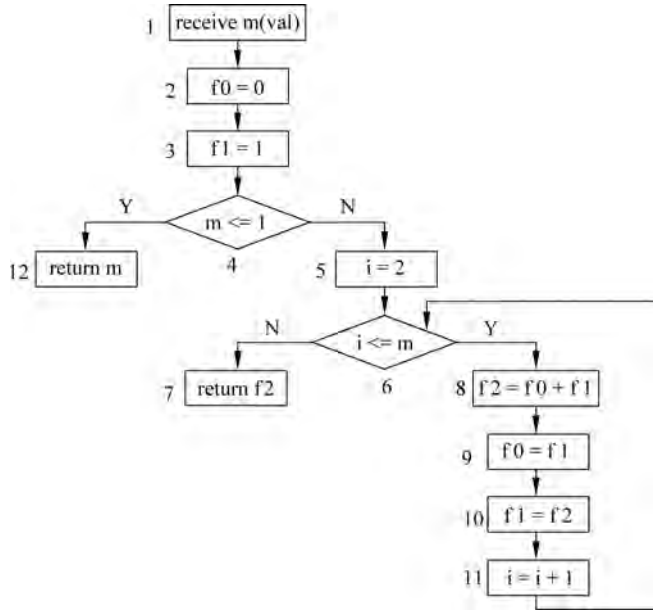

图 5-2 C 程序的流程图

控制流图是一个由基本块构成的有向图,有向边代表基本块之间的跳转关系;而基本块是一段只能从它的入口进入,并从它的出口离开的线性代码序列。接下来开始标识基本块。显然,节点 1 到节点 4 构成了一个基本块,称为 B1;节点 8 到节点 11 构成了另一个基本块,称为 B6。其他每一个节点独自构成一个基本块,其中节点 12 对应于 B2,节点 5 对应于 B3,节点 6 对应于 B4,节点 7 对应于 B5。得到的控制流图如图 5-3 所示。

增加一个以第一个基本块作为其后继的 entry 基本块，和一个放在流图结尾的 exit 基本块，并使流图中每一个实际从该程序出口出去的分支（基本块 B2 和 B5）转向 exit 基本块。

下面利用必经节点来标识该程序中的循环，将流图表示成以 entry 节点为根的一棵树。对于图 5-3 中的控制流图，其必经节点树如图 5-4 所示。

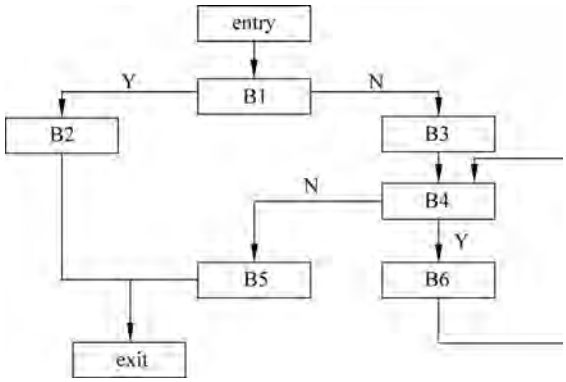

图 5-3　C 程序的控制流图

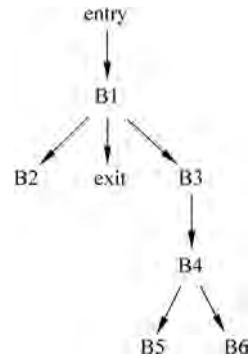

图 5-4　图的必经节点树

在控制流图中，回边的头节点是其尾节点的必经节点，例如从 B6 到 B4 的边。一个循环由满足下述条件的所有节点组成：循环的入口节点（即回边的头节点）是所有节点的必经节点，所有这些节点都可以到达入口节点，而且其内只有一条回边。因此 B4 和 B6 形成了一个循环，其中 B4 是循环的入口节点，而且此循环不包含图中其他的节点。

5.1.1　控制流分析方法

编译器对单一过程进行控制流分析，从确定构成过程的基本块开始，然后构造出它的流图，并使用必经节点来找出循环并优化它所找到的循环。当前大多数优化编译器使用的是必经节点和基于迭代的数据流分析，这可以满足大部分程序分析和优化的需要。

第 4 章讨论了从线性代码构建程序控制流图的算法（算法 12）。接下来，引入在本书后面的章节中将会用到的符号，记控制流图 $G=(V,E)$，其中 V 是节点集合，E 是有向边集合 $E\subseteq N\times N$。由于控制流图 G 是有向图，所以对于有向边 $(a,b)\in E$，经常写成 $a\rightarrow b$。

来进一步定义一个基本块 b 的后继基本块集合和前驱基本块集合。用 succ(b)表示基本块 $b\in V$ 的后继集合，用 pred(b)表示基本块 $b\in V$ 的前驱集合。则

$$succ(b)=\{n\in V \mid \exists e\in E,\text{使得 } e=b\rightarrow n\}$$

$$pred(b)=\{n\in V \mid \exists e\in E,\text{使得 } e=n\rightarrow b\}$$

分支节点是有多个后继的节点，汇合节点是有多个前驱的节点。

扩展基本块（Extended Basic Block，EBB）是从一个首领块开始的最长基本块序列，在这个基本块序列中，除了第一个节点之外不含其他汇合节点，第一个节点本身不必一定是汇合节点，例如，它可以是过程入口节点。因为扩展基本块只有一个入口但可能有多

个出口，所以可以将它看成是以它的入口基本块为根的一棵树。以这种方式构成的扩展基本块称为诸基本块。某些局部优化，如指令调度，在扩展基本块上比在一般基本块上更加有效。在图 5-3 所示的例子中，基本块 B1、B2 和 B3 构成了一个由多个基本块组成的扩展基本块。

算法 16 对以 r 为根的扩展基本块，构造其内诸基本块的索引集合；而算法 17 对以 r 为入口节点的流图，构造它的所有扩展基本块集合。算法首先设置 AllEbbs 为一个偶对集合，偶对的第一个元素为根基本块的索引，偶对的第二个元素为扩展基本块内的诸基本块的索引集合。这两个算法都使用全局变量 EbbRoots 记录扩展基本块的根基本块。

算法 16 构造具有指定根节点的扩展基本块内的诸基本块集合

输入：以 r 为根的扩展基本块
输出：扩展基本块内诸基本块的集合

```
1. procedure Build_Ebb (r, succ, pred)
2.    Ebb = Φ
3.    Add_Bbs(r, Ebb, succ, pred)
4.    return Ebb
5. procedure Add_Bbs(r, Ebb, succ, pred)
6.    x = Node
7.     Ebb∪ = {r}
8.    for succ(r) 中的每个节点 x do
9.          if |Pred(x)| = 1 并且 x ∉ Ebb then
10.             Add_Bbs(r, Ebb, Succ, Pred)
11.         else if  x∉ EbbRoots  then
12.             EbbRoots∪ = { x }
```

算法 17 构造给定流图中的所有扩展基本块

输入：以 r 为入口节点的流图
输出：流图中的所有扩展基本块集合

```
1. procedure Build_All_Ebbs(r, succ, pred)
2.      x = Node
3.      s = set of Node
4.      EbbRoots = {r}
5.      AllEbbs = Φ
6.      while EbbRoots ≠ Φ do
7.          从 EbbRoots 中选择一个节点 x
8.          EbbRoots -= {x}
9.          if ∀ s ∈ AllEbbs(s[1] ≠ x) then
10.             AllEbbs  ∪ = {< x, Build_Ebb(r, succ, pred) >}
11.     Build_All_Ebbs(entry, succ, pred)
```

作为 Build_Ebb()和 Build_All_Ebbs() 的例子，考虑如图 5-5 所示的流图。由该算法识别的扩展基本块是{entry}、{B1，B2，B3}、{B4，B6}、{B5，B7}和{exit}，如图 5-5 中虚线框所示。

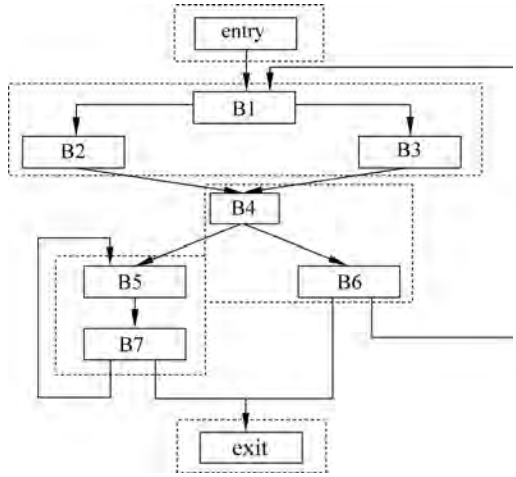

图 5-5　用虚线框指出扩展基本块的流图

类似地，反扩展基本块是以分支节点结束的，且除最后一个节点之外不包含其他分支节点的最长基本块序列。

5.1.2　流图的遍历

在程序编译过程中，编译器经常需要对控制流图进行遍历。第一个编译器常用的遍历策略是深度优先遍历(depth-first traversal)，这种遍历在访问图中的节点时，首先优先访问的是该节点的后继，而不是其兄弟。例如，图 5-6(a)的深度优先查找表示是图 5-6(b)。用深度优先遍历依次给每一个节点指定的编号，就是节点的深度优先编号。

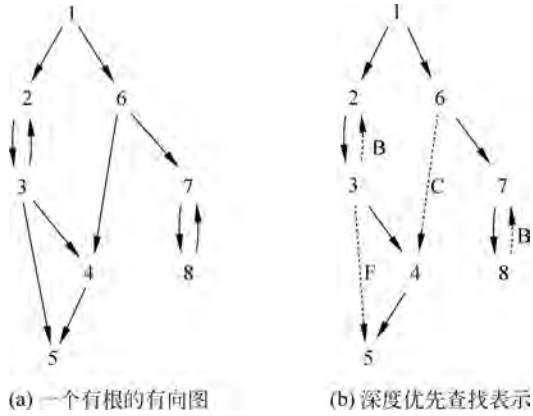

图 5-6　有向图及其深度优先查找表示

算法 18 用于构造图的深度优先表示(depth-first representation)，深度优先表示将图中所有的节点和那些构成深度优先次序的边表示为树的形式，称为深度优先生成树(depth-first spanning tree)，并将其他不是深度优先次序的一部分边用一种有别于树边的方式来表

示(用虚线表示)。

属于深度优先生成树的边称为树边(tree edge),不属于深度优先生成树的那些边分为以下三类。

(1) 前向边(forward edge):从一个节点到一个直接后继并且不是树边的边,用"F"标识。

(2) 后向边(back edge):从一个节点到树中它的一个祖先的边,用"B"标识。

(3) 交边(cross edge):连接两个在树中相互不是祖先的节点的边,用"C"标识。

算法 18 深度优先查找

输入: 有根的有向图
输出: 深度优先遍历

```
 1. N : in set of Node
 2. r, i : Node
 3. visit : Node→ boolean
 4. succ : set of Node
 5. x : in Node
 6. procedure Depth_First_Search(N, succ, x)
 7.     y : Node
 8.     Process_Before(x)
 9.     visit(x) = true
10.     for succ(x) 中的每个节点 y do
11.         if !visit(y)  then
12.             Process_Succ_Before(y)
13.             Depth_First_Search(N, succ, y)
14.             Process_Succ_After(y)
15.         Process_After(x)
16.         for V 中的每个节点 i do
17.             visit(i) = false
18.         Depth_First_Search(N, succ, r)
```

注意,图的深度优先表示不是唯一的。例如,图 5-7(a)所示的有向图有图 5-7(b)和图 5-7(c)所示的两个不同的深度优先表示。

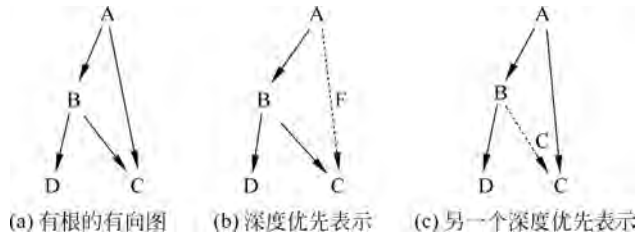

图 5-7 一个有根的有向图和它的两个深度优先表示

算法 18 实现通用的流图深度优先遍历,它提供 4 个动作执行点:

(1) Process_Before()允许在访问一个节点之前执行某种动作。

(2) Process_After()允许在访问一个节点之后执行某种动作。

(3) Process_Succ_Before()允许在访问一个节点的每一个后继之前执行某种动作。

(4) Process_Succ_After()允许在访问一个节点的每一个后继之后执行某种动作。

有向图的另外两种遍历策略是前序遍历(Preorder Traversal)和后序遍历(Postorder Traversal)。令有向图 G=(V,E),令 $E'\subseteq E$ 是 G 的深度优先表示中不包含反向边的边集合,则图 G 的前序遍历是这样一种遍历:其中每一个节点的遍历早于其后继,后继关系由 E' 定义。例如,entry、B1、B2、B3、B4、B5、B6、exit 是图 5-3 的一种前序遍历,序列 entry、B1、B3、B2、B4、B6、B5、exit 是图 5-3 的另一种前序遍历。

假设 G 和 E' 的定义如前所述,则图 G 的后序遍历是这样一种遍历:其中每一个节点的遍历晚于其后继,后继关系由 E' 定义。例如,exit、B5、B6、B4、B3、B2、B1、entry 是图 5-3 的一种后序遍历,而 exit、B6、B5、B2、B4、B3、B1、entry 是图 5-3 的另一种后序遍历。

算法 19 是深度优先查找的一种特定的版本,它计算根 $r\in V$ 的图 G=(V,E) 的深度优先生成树,以及它的前序遍历和后序遍历。在 Depth_first_Search_PP()执行之后,由根节点开始,并沿着类型为树边的边 e 而行,便得到深度优先生成树。编译器给每一个节点的前序编号和后序编号,分别存放在 Pre()和 Post()中。

算法 19　计算深度优先生成树及其前序遍历和后序遍历

输入: 有根的有向图
输出: 深度优先生成树、前序遍历顺序、后序遍历顺序

```
 1. r, x : Node
 2. i, j = 1: integer
 3. Succ, N : set of Node
 4. Pre, Post: Node → integer
 5. Visit: Node → boolean
 6. EType: enumtree, forword, back, cross
 7. procedure Depth_First_Search_PP(N, succ, x)
 8.     y : Node
 9.     visit(x) = true
10.     Pre(x) = j
11.     j += 1
12.     for succ(x) 中的每个节点 y do
13.         if !visit(y) then
14.             Depth_First_Search_PP(N, succ, y)
15.             EType(x → y) = tree
16.         else if Pre(x) < Pre(y) then
17.             Etype(x → y) = forward
18.         else if Post(y) = 0 then
19.             EType(x → y) = back
20.         else
21.             EType(x → y) = cross
22.     Post(x) = i
23.     i += 1
24.     for  V 中的每个节点 i do
25.         visit(i) = false
26.     Depth_First_Search_PP(N, succ, r)
```

最后，编译器还可以对流图进行宽度优先遍历(Breadth-first Traversal)。在这种遍历顺序中，对一个节点的所有直接后继的处理早于这些后继的任何还未处理的后继。例如，如果一个节点有两个直接后继 a 和 b，其中节点 b 同时又是节点 a 的后继，则 a 的处理先于 b 的处理。在宽度优先查找中节点被访问的次序是宽度优先次序(Breadth-first Order)。算法 20 构造流图中节点的宽度优先次序。对于图 5-6 的例子，次序 1、2、6、3、4、5、7、8 是宽度优先次序。

算法 20 构造流图中节点的宽度优先次序。

算法 20 计算宽度优先遍历

输入：有根的有向图
输出：宽度优先次序

```
 1. i = 2: integer
 2. N : in set of Node
 3. s: in Node
 4. procedure Breadth_First(N, succ, s)
 5.     x = Node
 6.     T = Φ
 7.     Order (r) = 1
 8.     i = 2
 9.     for succ(s) 中的每个节点 t do
10.         if Order(t) == nil then
11.             Order (t) = i
12.             i += 1
13.             T ⋃ = {t}
14.     for  T 中的每个节点 t do
15.         Breadth_First(N, succ, t)
16.    return Order
```

5.2 数据流分析及优化

基于数据流的优化主要分两步：首先，编译器对程序进行某种特定的数据流分析，得到分析的结果；然后，编译器按照分析结果对程序进行特定的优化。本节将讨论数据流分析和优化的主要技术。

5.2.1 优化的基本结论

优化编译器能够在不改变程序输出的情况下优化程序，以提高程序的性能。能够提高性能的转换有以下几种。

(1) 寄存器分配：使两个不重叠的临时变量存放在同一个寄存器中。

(2) 公共子表达式删除：如果一个表达式被计算了不止一次，删除多余的计算，只保留一个。

(3) 死代码删除：如果一个计算的结果从未被使用过，则删除该计算。

(4) 常数折叠：在编译期计算操作数为常数的表达式。

以上只是编译器优化的几种方式，事实上编译器并不能实现所有的优化方式，因为可计算性理论证明总是能发明出新的优化算法。

定义完全优化的编译器是这样一个编译器：它将每一个程序 P 转换为程序 Opt(P)，使得程序 Opt(P)是和 P 具有相同输入/输出行为的最小的程序。也可以假设编译器优化的是程序运行的速度，而不是程序大小，这里选择程序大小只是为了简化讨论。

对于任何一个既不产生输出又不会停止的程序 P，一个最短的 Opt(Q)是：

```
L: goto L
```

因此，假如有一个完全优化的编译器，就应当能用它来解决停机问题。即为了说明是否存在一个能使 P 停止的输入，只需看一下 Opt(P)是不是这个只有一行代码的无限循环。但是理论表明停机问题是不可判定问题，因此也不可能存在完全优化的编译器，即无论如何添加编译器优化，总是会有另一个更好(通常也更大)的优化编译器存在。

虽然不能构造出一个完全优化的编译器，但可以构造一些部分优化编译器。优化编译器将程序 P 转换为和 P 具有相同输入/输出行为，但比 P 更小或更快的程序 P′。尽管 P′ 未必是最优的，但只要在实践中 P′ 能够达到实际需求，优化就可被认为是成功的。

5.2.2　三地址码中间表示

本节集中讨论过程内的全局优化。过程内意味着分析局限在单一过程或函数内；全局意味着对过程内的所有语句或基本块统一进行分析。相比于过程内的优化，过程间优化更具有全局性，它能一次针对若干个过程和函数进行优化。

全局程序优化遵循以下通用步骤。

(1) 分析：遍历流图，分析并收集运行时所有相关信息，这必定是一种保守的近似信息。

(2) 转换：用某种方法修改程序，对程序进行改进，使它运行得更快，或体积更小。分析阶段收集的信息用来指导程序转换的过程，保证程序的语义不会发生改变。

编译器基于求解数据流方程来得到描述数据流的信息，该方程是从流图节点衍生出来的一组联立方程。接下来，基于三地址代码中间表示讨论描述各种数据流问题的方程及其求解。

为了更方便地讨论数据流方程问题，使用前面已经讨论过的三地址代码中间表示。下面给出三地址代码的抽象语法：

$$
\begin{aligned}
S \rightarrow\ & t = a \oplus b \\
\mid\ & \text{goto } L \\
\mid\ & L : \\
\mid\ & a = b \\
\mid\ & a = M[b]
\end{aligned}
$$

$$
\begin{aligned}
&| \ M[a] = b \\
&| \ \text{if}(a \ominus b, L_1, L_2) \\
&| \ f(a_1, a_2, \cdots, a_n) \\
&| \ t = f(a_1, a_2, \cdots, a_n)
\end{aligned}
$$

语句 S 包括几种典型的结构：t=a⊕b 代表某种二元运算，其中⊕是一个二元运算符；goto L 是跳转到标号 L 的控制转移语句；L：是一个语句标号；a=b 代表数据移动；a=M[b]和 M[a]=b 分别表示内存读和写；if (a⊖b，L_1，L_2)是有条件跳转语句，它对操作数 a 和 b 进行⊖比较运算，并根据比较结果跳转到标号 L_1 或标号 L_2 处；f(a_1，a_2，…，a_n)是具有 n 个参数的函数调用；t=f(a_1，a_2，…，a_n)是将函数调用的结果赋值给 t。

过程内优化接受前端生成的三地址代码中间表示为输入，对其进行程序分析，并将其转换为更优的三地址代码。在这个过程中，优化器可能会对三地址代码进行移动、插入、删除和修改等操作。然后，经过优化后的程序交给编译器的后续阶段继续进行处理。需要特别指出的是，编译器在把经过优化后的三地址代码输出给编译器后端之前，可能需要将这些线性的语句序列重新变换为嵌套的表达式，主要原因是指令选择阶段在只含一个二元运算或数据移动的“原子化的”语句上执行树匹配的效果一般不理想，第 6 章将深入讨论指令选择。

本节采用了一个更简单的控制流图的观点，即流图中每个节点(语句)n 都有指向其他后继的有向边，也就是说，这里的流图基本块都是平凡的。这个约定不影响数据流分析的正确性，但对大型程序不太高效，5.2.5 节将讨论对其加速的若干技术。

5.2.3 数据流分析

本节讨论程序的种数据流分析算法，包括到达定值分析、可用表达式分析、到达表达式分析、活跃分析等。

1. 到达定值分析

许多优化需要提前了解在一个点对一个程序变量 t 进行的特定赋值，是否会直接影响程序中另一点的变量 t 的值。变量 t 有一个无二义性的定值(unambiguous definition)，是指程序中有形如 t=a⊕b 或 t=M[a] 对变量 t 进行赋值的语句。给定一个定值 d，如果存在一条从 d 到语句 u 的由控制流边组成的路径，并且该路径上不包含对 t 的任何其他定值，我们就说定值 d 到达(reach)语句 u。编译器进行的这类分析被称为到达定值(reaching definition)分析。

模糊定值(ambiguous definition)是指这样的语句，它可能给变量 t 赋值，也可能不给变量 t 赋值。例如，如果 t 是一个全局变量，语句 s 是函数调用语句，被调用的这个函数有可能修改 t，也可能不修改 t，那么 s 就是对变量 t 的一个模糊定值。但是，此处讨论的数据流分析暂时不将逃逸变量作为临时变量来对待，而是作为具有存储位置的内存变量来对待。这意味着本节不讨论模糊定值(但这同时也失去了优化这些逃逸变量的机会)。在本节的讨论中，假设所有的定值都是明确的。

到达定值的计算可以表达成对数据流方程的求解。首先给每个赋值语句标记一个标号，并且计算这些标号组成的集合。可以说语句

$$d: t = x \oplus y$$

生成(generate)定值 d，因为无论到达这条语句开始的其他定值是什么，定值 d 都能到达该语句的末尾。并且这条语句杀死(kill)了对变量 t 的其他定值，因为无论到达这条语句开始的变量 t 的其他定值是什么，它们都不能到达这条语句的末尾，即它们不能直接影响这条语句之后的变量 t 的值。

临时变量 t 的所有定值(即定值标号)组成的集合被定义为 defs(t)。表 5-1 给出了到达定值的生成集合和杀死集合。

表 5-1　到达定值的生成集合(gen)和杀死集合(kill)

语句 s	gen[s]	kill[s]
d : t=b ⊕ c	{d}	defs(t)−{d}
d : t=M [b]	{d}	defs(t)−{d}
M [a]=b	{}	{}
if (a⊖ b,L_1,L_2)	{}	{}
goto L	{}	{}
L :	{}	{}
f (a_1,a_2,…,a_n)	{}	{}
d : t=f (a_1,a_2,…,a_n)	{d}	defs(t)−{d}

利用生成集合 gen 和杀死集合 kill，可以计算出到达每个节点 n 的开始的定值集合 in[n]和到达 n 末尾的定值集合 out[n]：

$$\text{in}[n] = \bigcup_{p \in \text{pred}[n]} \text{out}[p]$$

$$\text{out}[n] = \text{gen}[n] \cup (\text{in}[n] - \text{kill}[n])$$

可以通过迭代来解这两个方程：首先，对所有的节点 n，编译器将集合 in[n] 和 out[n] 初始化为空集合 Φ；然后，编译器将每个方程看作一个赋值语句，反复执行赋值操作，直到 in[n] 和 out[n] 不再改变为止。

考虑如下有标号的程序(将这些标号直接作为定值标号)：

```
1.      a = 5
2.      c = 1
3. L1 : if   c > a goto L2
4.      c = c + c
5.     goto L1
6. L2 :  a = c - a
7.       c = 0
```

迭代进行赋值，依次计算每条语句的输入集合 in[ ] 和输出集合 out[ ]，注意到迭代 3 和迭代 2 的 in[n]和 out[n]相同，则算法可结束运行，计算过程如表 5-2 所示。

表 5-2 迭代计算的结果

语句	生成集合和杀死集合		迭代 1		迭代 2		迭代 3	
n	gen[n]	kill[n]	in[n]	out[n]	in[n]	out[n]	in[n]	out[n]
1	1	6		1		1		1
2	2	4,7	1	1,2	1	1,2	1	1,2
3			1,2	1,2	1,2,4	1,2,4	1,2,4	1,2,4
4	4	2,7	1,2	1,4	1,2,4	1,4	1,2,4	1,4
5			1,4	1,4	1,4	1,4	1,4	1,4
6	6	1	1,2	2,6	1,2,4	2,4,6	1,2,4	2,4,6
7	7	2,4	2,6	6,7	2,4,6	6,7	2,4,6	6,7

编译器可将计算得到的到达定值信息用于多种优化。例如,编译器可以尝试进行常数传播(constant propagation)优化,即如果只有一个定值 d：t=c 可以到达程序点 u,且该定值是对变量 t 的常量定值,则可将程序点 u 对变量 t 的使用替换成常量 c。例如,在上面的示例程序中,变量 a 只有一个常量定值 1 到达语句 3,因此,可以通过常量传播将判断条件 c>a 替换为 c> 5。

2. 可用表达式分析

公共子表达式删除(Common-Subexpression Elimination,CSE)是一种重要的编译器优化方法,这种优化对一个多次计算 x⊕y 的程序,删除其中的重复计算。编译器可借助可用表达式(available expression)分析来实施这种优化。

如果从流图的入口节点到节点 n 的每条路径上,表达式 x⊕y 都至少被计算一次,并且在每条路径上 x⊕y 的最近一次出现之后,再没有对变量 x 或 y 的定值,那么称表达式 x⊕y 在节点 n 是可用的(available)。

可以利用集合 gen 和 kill,并结合数据流方程组来表达可用性的概念,这里 gen 和 kill 都是表达式的集合。

每一个计算表达式 x⊕y 的节点都生成{x⊕y},而每一个对变量 x 或 y 的定值都会杀死表达式{x⊕y},可用表达式的集合 gen 和 kill 如表 5-3 所示。

表 5-3 可用表达式的集合 gen 和 kill

语句 s	gen[s]	kill[s]
t=b⊕c	{b⊕c}−kill[s]	包含 t 的表达式
t=M[b]	{M[b]}−kill[s]	包含 t 的表达式
M[a]=b	{}	形如 M[x]的表达式
if(a⊖b,L_1,L_2)	{}	{}
goto L	{}	{}
L：	{}	{}
f(a_1,a_2,…,a_n)	{}	形如 M[x]的表达式
t=f(a_1,a_2,…,a_n)	{}	包含 t 的表达式和形如 M[t]的表达式

一般地，语句 t＝b＋c 生成表达式 b＋c。但是对于 b＝b＋c，由于在 b＋c 之后有对 b 的定值，因此不会生成 b＋c。方程

$$\text{gen}[s]=\{b \oplus c\}-\text{kill}[s]$$

给出了这种情况。

存储操作指令 M[a]＝b 可能修改任意存储位置，因此它杀死所有的从内存中取操作数的表达式 M[x]。如果编译器可以肯定 a≠x，即变量 a 和 x 肯定不指向同一内存地址，则编译器可以得到更精确的结论：认为 M[a]＝b 不会杀死 M[x]，这称为别名分析。

给定 gen 和 kill，编译器可以计算集合 in 和 out：

$$\text{in}[n]=\bigcup_{p\in \text{pred}[n]} \text{out}[p]$$

$$\text{out}[n]=\text{gen}[n] \cup (\text{in}[n]-\text{kill}[n])$$

集合 in 和 out 的计算几乎和到达定值的计算一样，只是在计算 in[n] 时，计算的是节点 n 的所有前驱的 out 集合的交集，而不是并集，即当且仅当每条到达节点 n 的路径上都计算了某个表达式时，这个表达式在节点 n 才是可用的。

为了通过迭代计算 in 和 out，需要首先定义起始节点的集合 in 为空集，将其他所有节点的集合初始化为全集（即所有表达式组成的集合），而不是空集。这是因为交集运算会使集合变小，而不像到达定值计算中的并集那样使集合变大。然后，算法寻找此方程组的最大不动点。

3. 到达表达式分析

如果流图中存在一条从节点 s 到节点 n 的路径，且该路径不存在任何对 x 和 y 的赋值，或者任何对 x⊕y 的计算，就称节点 s 中的表达式 t＝x ⊕y 到达节点 n。和往常一样，能够用集合 gen 和 kill 来表示到达表达式。值得注意的是，在到达表达式的定义中，只要求节点 s 到节点 n 存在一条路径即可；而在可用表达式中，必须考查从节点 s 到节点 n 的所有路径。

在实际中，公共子表达式删除优化需要用到的到达表达式，可能只是程序中所有表达式的一个小的子集。因此，编译器可以更加简化地计算到达表达式：从节点 n 开始后向搜索，一旦发现对表达式 x⊕y 的计算，便停止搜索，也可以在计算可用表达式的过程中计算到达表达式。

4. 活跃分析

如果一个变量 t 在一个给定的程序点 u 后面会被使用到，则称变量 t 在程序点 u 是活跃的。计算变量活跃性的分析过程被称为活跃分析（liveness analysis）。

活跃分析也可以用 gen 和 kill 表示：对变量的任何使用都会使该变量成为活跃的，对变量的任何定值都会杀死该变量的活跃性。对三地址代码计算活跃性集合 gen 和 kill 的规则由表 5-4 给出。

表 5-4 对三地址代码计算活跃性集合 gen 和 kill 的规则

语句 s	gen[s]	kill[s]
t=b ⊕ c	{b,c}	{t}
t=M[b]	{b}	{t}
M[a]=b	{a,b}	{}
if(a⊖b,L_1,L_2)	{a,b}	{}
goto L	{}	{}
L :	{}	{}
f (a_1,a_2,…,a_n)	{a_1,a_2,…,a_n}	{}
t=f (a_1,a_2,…,a_n)	{a_1,a_2,…,a_n}	{t}

集合 in 和 out 的方程组与到达定值和可用表达式的方程组类似，但是活跃分析是向后(Back-ward)的数据流分析，因此 in 和 out 的计算也是向后的：

$$\text{in}[n] = \text{gen}[n] \cup (\text{out}[n] - \text{kill}[n])$$

$$\text{out}[n] = \bigcup_{s \in \text{succ}[n]} \text{in}[s]$$

5.2.4 程序优化

编译器能够利用数据流分析的结果对程序进行优化。本节将讨论几种重要的程序优化：公共子表达式删除、常数传播、复写传播、死代码删除等。

1. 公共子表达式删除

给定流图中的一条语句 d：t=x ⊕ y，如果表达式 x⊕y 在另一条语句 s：u=x ⊕y 处是可用的，那么编译器可以删除 s 中对 x⊕y 的计算。这种优化被称为公共子表达式删除优化。

公共子表达式删除优化的主要步骤如下。

(1) 编译器计算可用表达式(Available Expression)，即寻找形如 d：t=x ⊕y 且满足如下条件的语句：在从语句 d 到语句 s 的所有路径上，既没有计算 x⊕y，也没有对 x 或 y 定值。

(2) 编译器生成一个新的临时变量 w，并将上述语句 d 重写为：

```
d : w = x ⊕ y
e : t = w
```

(3) 编译器将语句 s 改为：

```
s : u = w
```

注意，公共子表达式删除优化引入了额外的数据移动语句(如上面的 t=w 等)，编译器将依靠复写传播优化或接合优化来删除部分或全部的多余赋值。

2. 常数传播

假设有一条语句 d：t=c(其中 c 是常数)和另一条使用 t 的语句 n，例如 n：y=t⊕x，如

果 d 是能够到达 n 的唯一对变量 t 的定值，则在语句 n 中，变量 t 可被替换成常数，即可以将语句 n 重写为 n：y＝c⊕x。

3. 复写传播

复写传播与常数传播类似，但传播的不是常数 c，而是一个变量 z。假设有一条语句 d：t＝z，以及另一条使用 t 的语句 n，例如 n：y＝t⊕x，如果 d 是能够到达语句 n 的唯一定值，同时任何从 d 到 n 的路径（包括多次经过 n 的路径）都没有对 z 定值，那么可以将 n 重写为 n：y＝z⊕x。

复写传播和寄存器分配存在密切联系：基于图着色的寄存器分配器可以在分配过程中完成对节点的接合，即把与数据移动语句 d：x＝y 相关的变量 x 和 y 分配到同一个物理寄存器 r 中，从而编译器能够把得到的无用的数据移动语句 d：r＝r 移除；并且考虑到如果在寄存器分配之前进行复写传播，则有可能增加寄存器溢出的数目。因此，如果只是为了删除冗余的数据移动而做复写传播的话，应该等到寄存器分配之后再进行。

但另一方面，编译器在中端优化阶段进行复写传播有可能识别出更多其他优化机会，如公共子表达式删除等。例如，在下面的程序中：

```
a = y + z
u = y
c = u + z
```

编译器对数据移动语句 u＝y 执行复写传播后，其他两个加法表达式才能被识别为公共子表达式。因此，编译器设计者需要根据优化的目标和组织方式，仔细选择优化进行的时机和顺序。

4. 死代码删除

对于语句 d：a＝b ⊕c 或 d：a＝M[x]，如果 a 不在 s 的出口活跃集合中，则语句 d 可以被删除。这种优化被称为死代码删除。

需要特别注意：程序优化绝对不能更改程序的可观察行为。有些指令有隐含的副作用，如进行输入输出或者抛出异常等，在这种情况下，编译器不应该删除这类代码。

5.2.5　数据流分析的改进

5.2.3 节已经讨论了基于数据流方程和迭代求解的数据流分析方法，本节将讨论几种可以加速数据流分析的重要改进。

1. 位向量

位向量（bit vector）是对有限集合 S 的高效表示，对集合 S 引入一个对应的向量 V（V[i]∈{0，1}，0≤i ≤n）

$$S = \{s_0, s_1, \cdots, s_n\}$$

$$V = \{b_0, b_1, \cdots, b_n\}$$

元素 x∈S（x＝s_i）当且仅当 V[i]＝（b_i＝1），即元素 x 在集合 S 中，当且仅当 x 在对应向量 V 中的值是 1。

在位向量表示中，对两个集合 S 和 T 的操作都可以被表示成其对应位向量上的相应操

作。例如，集合并集可以用位向量的按位或操作获得；集合交集可以通过位向量按位与操作获得；集合的补集可以通过按位补操作获得等。如果计算机的字长为 W 位，向量长度为 N 位，那么用 N /W 条按位或运算指令组成的序列就可以计算出两个集合的并。当然，还必须包括 2N /W 条取指令和 N /W 条存指令，以及索引和循环的开销。

需要注意，如果集合非常稀疏，其对应的位向量中几乎全部是零，这种情况下使用位向量表示会占用更多的内存空间。

2. 基本块

前面我们基于三地址代码基本语句，讨论了数据流方程求解。如果以基本块为单位，也可以加速数据流求解的过程。下面以到达定值为例进行讨论，但这里的讨论也适用于其他数据流分析。

考虑有什么样的定值到达了节点 n 的出口，即节点 n 的 out 集合：

$$out[n] = gen[n] \cup (in[n] - kill[n])$$

因为 in[n]恰好等于 out[p]，因此有

$$out[n] = gen[n] \cup ((gen[p] \cup (in[p] - kill[p])) - kill[n])$$

根据等式

$$(A \cup B) - C = (A - C) \cup (B - C)$$
$$(A - B) - C = A - (B \cup C)$$

有

$$out[n] = gen[n] \cup (gen[p] - kill[n]) \cup (in[p] - (kill[p] - kill[n]))$$

如果希望节点 pn 合并 p 和 n 的作用，那么从最后一个方程可以看出 pn 的正确的 gen 和 kill 集合分别为

$$gen[pn] = gen[n] \cup (gen[p] - kill[n])$$
$$kill[pn] = kill[p] \cup kill[n]$$

可以用这种方法合并一个基本块中的所有语句，从而得到整个基本块的 gen 和 kill 集合。控制流图中基本块的数量比单个语句的数量小得多，所以基于基本块的迭代数据流分析的速度也要快得多。

上述迭代数据流分析算法一旦完成，就可以从整个基本块的 in 集合开始，用一遍迭代来计算基本块中语句 n 的前驱语句的 gen 和 kill 集合，从而重新获得基本块中各个语句的数据流信息。

3. 节点排序

在前向数据流问题中(例如到达定值或可用表达式)，一个节点的 out 集合的信息将传递给它的后继节点的 in 集合。如果能够安排每个节点的计算次序，使得它都先于它的后继节点进行，就能够加速数据流分析的计算过程。

为此，对于有向无环图，可以先对流图进行拓扑排序(这将给出一种顺序，其中每一个节点都在它的后继节点之前)，然后按照排好的顺序计算数据流方程。即便图中含有环，对图进行类似的拓扑排序，仍有助于减少图的迭代次数，在这种伪拓扑的排序中，大多数节点

的计算仍先于它们的后继节点。

算法 21 给出了基于深度优先遍历的图拓扑排序算法。利用该算法给出的 sorted 数组(它给出深度优先遍历计算出的顺序),数据流方程可以按下面的方式迭代求解:

```
repeat
    for i = 1 to N
        n = sorted[i]
        in = ⋃ out[p]
             p∈pred[n]
        out[n] = gen[n] ∪ (in - kill[n])
until out 集合不再改变
```

因为 in 只是为了计算 out 局部使用,所以不需要将 in 设置成全局数组。

算法 21　按照深度优先搜索拓扑排序

输入: 有向图 G=(V,E) 和节点 n
输出: 有向图 G 的伪拓扑排序

```
 1. procedure Topological-sort(V, E, n)
 2.     N = |V|
 3.     for 每个节点 i ∈ V do
 4.         mark[i] = false
 5.     DFS(n)
 6. procedure DFS(i)
 7.     if mark[i] = false then
 8.         mark[i] = true
 9.         for i 的每个后继 s do
10.             DFS(s)
11.         sorted[N] = i
12.         N = N - 1
```

对于后向数据流问题(如活跃分析),只需对遍历顺序稍加修改,即从出口节点开始遍历前驱,而不是从入口节点开始遍历后继。

4. 使用-定值链和定值-使用链

到达定值的相关信息可以作为使用-定值链来保存,即对变量 x 的每一个使用,它的使用-定值链是一张列表,此表记录着能够到达该使用的 x 的所有定值。从技术上讲,使用-定值链并不能加快数据流分析,但是能够更高效地实现那些需要分析结果的优化算法。

第 4 章讨论了静态单赋值形式,静态单赋值形式可看成使用-定值链的一种特例,静态单赋值形式中的使用-定值链中最多只有一个元素,因为每个变量只有唯一的定值。这个性质使得编译器在静态单赋值形式上进行优化会更加高效。

表示活跃分析结果的一种方法是利用定值-使用链,即每一个定值有一张表,此表记录着该定值的所有可能的使用。

5. 工作表算法

在基于迭代求解的算法中,只要 repeat-until 循环的一次迭代中有任意一个 out 集合

发生改变，则所有的方程都需要重新计算，这在很多情况下比较低效，因为大多数方程也许并没有在此次迭代中继续改变。

工作表算法是对基本迭代算法的改进，工作表的核心步骤是记录需要重新计算的 out 集合。只要节点 n 必须重新计算，并且它的 out 集合发生了改变，那么 n 的所有后继节点都将放到工作表中(如果后继节点不在表中的话)。算法 22 以计算到达定值为例，说明了工作表的具体方法。当从工作表 W 中取出一个节点 n 进行处理时，如果所选择的节点是 sorted 数组中最早出现的节点，则工作表算法将收敛得更快。

算法 22 到达定值的工作表算法

```
1. W = 所有节点的集合
2. while  W 不为空 do
3.     n = 从 W 中移除的节点
4.     old = out[n]
5.     in = ⋃ out[p]
          p∈pred[n]
6.     out[n] = gen[n] ⋃ (in - kill[n])
7.     if old ≠ out[n] then
8.         for n 的每个后继节点  s do
9.             if s ∉ W then
10.                将 s 加入 W
```

6. 增量式数据流分析

利用数据流分析的结果，优化器能够执行各种程序转换：移动、修改或删除指令，并且这些优化可以具有以下级联效果。

(1) 删除死代码 a＝b⊕c 可能导致以前的代码 b＝x⊕y 变成死代码。

(2) 删除一个公共子表达式可能会产生另一个可以被删除的公共子表达式。例如程序：

```
x = b + c
y = a + x
u = b + c
v = a + u
```

当 u＝b＋c 被替换成 u＝x 后，复写传播将 a＋u 改变成 a＋x，这样就又出现了一个能够被删除的公共子表达式。

根据这个事实，基于数据流的优化器的一种简单的组织方法是，首先，执行全局的数据流分析，其次，做所有可能的基于数据流的优化，再次重复进行全局数据流分析，最后执行优化，如此迭代反复，直到不能发现更多的优化为止。

通常情况下迭代过程最多执行两三次即可终止，但在最坏的情况下，迭代可能进行很多次。考虑语句 $z=a_1+a_2+a_3+\cdots+a_n$，其中 z 是死代码，该语句被翻译成如下三地址代码：

```
x1 = a1 + a2
x2 = x1 + a3
     ...
xn-2 = xn-3 + an-1
z = xn-2 + an
```

第一轮死代码删除优化移除对 z 的定值，下一轮活跃分析判断出 x_{n-2} 是死代码，然后死代码删除优化再移除 x_{n-2}，以此类推，编译器共计需要 n 轮分析和优化才能删除 x_1。

为了避免这种最坏情况降低编译效率，编译器可以采取以下几种策略。

(1) 设定截止阈值：使分析和优化的执行次数不超过 k 次(例如可取 k 为 3)。尽管这种做法未必能取得最优编译结果，但至少可以保证编译能在合理的时间内结束。

(2) 级联分析：设计一种新的数据流分析，能够预测将要执行的优化的级联效果。

(3) 增量数据流分析：当优化器对程序进行某种转换时(这种转换可能会使数据流信息无效)，优化器并不抛弃原来的数据流信息，而是对它进行“修订”。

接下来，讨论级联分析的一个重要实例：值编号(value numbering)，它只需执行一遍，就能找到一个基本块内所有的级联的公共子表达式。

该算法维护一张表 T，此表将变量映射为值编号，也将形如(值编号，操作符，值编号)的三元组映射为值编号。为了提高效率，应该用哈希表来实现 T。此外，算法还需要一个全局编号 N，用于统计迄今为止已见到了多少个不同的值。

利用 T 和 N，值编号算法(参见算法 23)从头到尾扫描基本块的四元式。当看到表达式 b+c 时，它会查找 b 的值编号和 c 的值编号。然后在 T 中查找 hash(nb，+，nc)，如果找到了，则意味着 b+c 重复了较早时候的计算，将 b+c 标记为可删除的，并且使用以前计算的结果。如果没有找到，则 b+c 继续保留在程序中，同时也将它加入哈希表中。

算法 23　值编号算法

```
1. T = empty
2. N = 0
3. for 块中的每个四元式 a = b ⊕ c do
4.     if (b ↦k) ∈ T 对于某个 k then
5.         n_b = k
6.     else
7.         N = N + 1
8.         n_b = N
9.         将 b ↦n_b 放入 T 中
10.    if 对于某个 k, 有 (c ↦k) ∈ T then
11.      n_c = k
12.    else
13.        N = N + 1
14.        n_c = N
15.        将 c ↦n_c 放入 T 中
16.     if 对于某个 m, 有 ((n_b, ⊕, n_c) ↦m) ∈ T then
17.      将 a ↦m 放入 T 中
18.      将这个四元式 a = b ⊕ c 标记为公共子表达式
19.    else
20.      N = N + 1
21.      将 (n_b, ⊕, n_c) ↦N 放入 T 中
22.      将 a ↦N 放入 T 中
```

5.3 别名分析

如果两个变量指向同一个内存单元，则它们称为别名(alias)；编译器对别名变量进行的分析称为别名分析(alias analysis)。例如，对如下的示例程序：

```
M[p] = 5;
M[q] = 7;
a = M[p];
```

希望到达定值分析指出只有 M[p]的一个定值(即 5)到达了 a 的定值点。但是问题在于无法确定另一个变量(此处是 q)是否是 p 的别名，即变量 q 和 p 是否指向同一个内存地址，如果是，就只有定值(7)能够到达 a。

类似地，如果下面的程序采用引用方式传递参数：

```
void f(int &i, int &j){
     i = 5;
    j = 7;
    return i;
}
```

那么当用 f(x,x)来调用函数 f 时，到达定值分析必须结合别名分析，才能确定 i 可能和 j 是同一个变量这一事实。

一般地，程序中有可能会有别名的变量包括：

(1) 作为传地址参数的变量；

(2) 取了其地址的变量；

(3) 析取指针的左值表达式；

(4) 显式带下标的数组左值表达式；

(5) 内层嵌套过程中使用的变量。

本节讨论的别名分析有两点要注意：第一，本节描述的别名分析只考虑临时变量的值，而不考虑逃逸变量(逃逸变量是被取地址的变量，因此它们必须位于内存中)；第二，本节讨论的是可能别名关系(may-alias)，即如果程序运行时，变量 p 和 q 可能指向相同的内存地址，则 p 和 q 就是可能别名。在大多数数据流分析中，静态(编译时)信息不可能完全精确，因此可能别名关系是保守的，即如果不能证明 p 绝对不是 q 的一个别名，就说 p 和 q 是可能别名。接下来，为简单起见，经常将可能别名简称为别名。

5.3.1 基于类型的别名分析

对于强类型语言，如果两个变量具有不一致的类型，那么它们不可能是同一存储空间的不同名字，因此可以利用类型信息来指导可能别名关系的构建。另外，在这些语言中，程序员不能显式地使指针指向一个局部变量，也可以利用这一点进行别名分析。

根据变量类型将程序使用的所有存储空间划分为一些不相交的集合，这些集合称为别名类(alias class)。别名类的计算涉及类型，而编译器语义分析之后的阶段对类型一无所知，因此必须在语义分析阶段计算并保存足够多的类型信息，供编译器的后续程序分析和优化使用。

5.3.2　基于流的别名分析

除了基于类型的别名类外，还可以基于创建点构造别名类。在下面的程序中：

```
int *p, *q;
int h, i;
p = &h;
q = &i;
*p = 0;
*q = 5;
a = *p;
```

即使 p 和 q 的类型相同，也知道它们指向的是不同的记录。因此，赋给 a 的一定是 0，定值 *q=5 不会影响 a。

为了能自动识别出这些区别，需要为每个创建点构造一个别名类。也就是说，对每个分配记录的不同语句(即在 C 中每次调用 malloc 或在 Java 中每次调用 new)，都构造一个新的别名类。此外，每个不同的被取了地址的局部或全局变量都属于同一个别名类。

指针(或传地址的参数)可以指向多个变量，这些变量可以属于不同的别名类。例如以下程序：

```
1.  p = new List();
2.  q = new List();
3.  if(a == 0)
4.    p = q;
5.  p.head = 4;
```

其中第 5 行，q 只能指向别名类 2，但是 p 指向的别名类既有可能是 1，也有可能是 2，具体取决于 a 的值。

因此，编译器必须给每个内存单元关联一组别名类，而不是只关联某一个别名类。第 2 行后，得到了信息 $p \mapsto \{1\}$，$q \mapsto \{2\}$；第 4 行后，有 $p \mapsto \{2\}$，$q \mapsto \{2\}$。但是当控制流的两条分支汇合时(上述程序中，包括控制边 3→5 和 4→5)，必须合并别名类信息，于是在第 5 行后，有 $p \mapsto \{1,2\}$，$q \mapsto \{2\}$。

数据流算法处理形如(t,d,k)的元组集合，其中 t 为变量，d 为程序位置(记录的分配点)，d 和 k 一起表示在位置 d 分配的记录的第 k 个域的所有实例的别名类。如果 t－k 在语句 s 的开始可能指向别名类为 d 的一个记录，集合 in[s]就包含了(t,d,k)。

这里不使用 gen 和 kill 集合，而是使用一个转移函数(transfer function)：如果 A 是语句 s 的入口处的别名信息(元组集合)，则 $trans_s(A)$是语句 s 的出口处的别名信息。不同类型四元式的转移函数的定义如表 5-5 所示。

表 5-5 别名流分析的转移函数

语句 s	$trans_s(A)$
t=b	$(A-\Sigma_t) \cup \{(t,d,k) \mid (b,d,k) \in A\}$
t=b+k (k 为常量)	$(A-\Sigma_t) \cup \{(t,d,i) \mid (b,d,i-k) \in A\}$
t=b⊕c	$(A-\Sigma_t) \cup \{(t,d,i) \mid (b,d,j) \in A \vee (c,d,k) \in A\}$
t=M[b]	$A \cup \Sigma_t$
M[a]=b	A
if(a⊖b,L_1,L_2)	A
goto L	A
L:	A
f(a_1,a_2,…,a_n)	A
d: t=allocRecord(a)	$(A-\Sigma_t) \cup \{(t,d,0)\}$
t=f(a_1,a_2,…,a_n)	$A \cup \Sigma_t$

初始集合 A_0 包括(FP,frame,0),frame 是当前函数的所有分配在栈帧上的变量的特殊别名类。

使用缩写 Σ_t 表示所有元组(t,d,k)的集合,其中 d、k 是其类型和变量 t 一致的任意记录域的别名类。让编译器前端配合,由它为每个变量 t 提供一个“小的”t 集合,可以使得这种分析更为精确。

别名流分析的集合方程组是

$$in[s_0] = A_0, \text{ 其中 } s_0 \text{ 是起始节点}$$

$$in[n] = \bigcup_{p \in pred[n]} out[p]$$

$$out[n] = trans_n(in[n])$$

可以像通常一样用迭代方法求解此方程组。最后,如果存在 d,k 使(p,d,k)∈in[s] 并且(q,d,k)∈in[s],就说 p 在语句 s 中可能与 q 别名。

5.3.3 别名信息的使用

给定可能别名关系,可以将每个别名类作为数据流分析中的一个“变量”来对待,就像到达定值和可用表达式中的变量一样。

以可用表达式为例,修改表中的一行,设置 gen 和 kill 集合,如表 5-6 所示。

表 5-6 修改后的集合

语句 s	gen[s]	kill[s]
M[a]=b	{}	{M[x] | a 在语句 s 可能与 x 别名}

现在分析下面的程序片段:

```
1. u = M[t]
2. M[x] = r
3. w = M[t]
4. b = u + w
```

没有别名分析时，由于不知道 t 和 x 是否相关，因此会假定第 2 行的存储指令可能会杀死 M[t]的可用性。但是假设别名分析已判断出在语句 2 中 t 不可能和 x 别名，则在第 3 行中 M[t]仍是可用的，于是，可以将它作为公共子表达式删除。经复写传播后，得到：

```
1. z = M[t]
2. M[x] = r
3. b = z + z
```

上面讨论的是过程内的别名分析，过程间的别名分析有助于分析函数调用指令的作用。例如，在下面的程序中：

```
1.  t = fp + 12
2.  u = M[t]
3.  f(t)
4.  w = M[t]
5.  b = u + w
```

如果函数 f 会修改 M[t]，则 M[t]在第 4 行就不是可用的了；否则，M[t]在第 4 行就是可用的。

5.4　过程间分析及优化

过程间分析和优化可以跨越函数边界，进行多个函数的分析和优化。本节将讨论进行过程间分析和优化的主要技术。

5.4.1　分析

有两方面的原因可能导致过程调用低效：一是单过程分析和优化中信息的缺失，这是由分析和变换区域中函数调用引起的；二是为维护过程调用的固有抽象而引入的特定开销。引入过程间分析是为解决由第一个原因引起的过程调用低效。

1. 调用图

编译器在过程间分析中必须解决的第一个问题是构建调用图(call graph)。在最简单的情况下，每个函数调用位置处被调用的函数的名称均由字面常量确定，如在函数调用“call foo(x,y,z)”处，被调用函数的名称为 foo，编译器将创建一个名为 foo 的调用图节点。编译器会为程序中的每个函数创建一个调用图节点，并针对每个调用位置向调用图添加一条边。这一过程花费的时间与程序中的函数数目和调用位置数目成正比。

源语言特性可能增加构建调用图的难度。例如，考虑下面的程序，其精确调用图如图 5-8(a)所示。

```
int compose(int f(), int g())
```

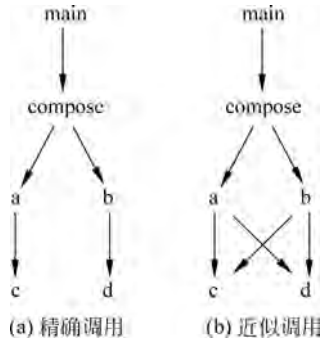

图 5-8　构建具有函数值参数的调用图

```
    return f(g);
int a(int z())
    return z();
int b(int z())
    return z();
int c()
    return … ;
int d()
    return … ;
int main(int argc,char * argv[])
    return compose(a,c) + compose(b,d);
```

1）值为过程的变量

如果程序使用值为过程的变量，则编译器必须分析代码，以估计在每个调用值为过程的调用位置上，潜在被调用者的集合。编译器可以从由使用显式字面常数的调用指定的图开始构建。接下来，它可以跟踪函数值围绕调用图的这个子集的传播，并在需要处添加边。

编译器可以使用全局常量传播的一种简单模拟将函数值从过程入口处传递到使用函数值的调用位置，并在其间使用集合并运算。

正如上述代码所示，直接分析可能会高估调用图的边集。该代码调用 compose 函数计算 a(c)和 b(d)。但是，简单分析将得出的结论是 compose 中的形参 g 可以接收 c 或 d 作为实参，因此程序形成的复合函数可能是 a(c)、a(d)、b(c)、b(d)中的任何一个，如图 5-8(b)所示。为构建精确的调用图，编译器必须跟踪沿同一路径共同传递的参数的集合。接下来，算法可以单独处理每个集合以推导精确的调用图。另外，算法还可以把每个值迁移的代码路径标记到值上，并使用路径信息来避免向调用图添加不合逻辑的边，如(a,d)或(b,c)。

2）根据上下文解析的名字

一些语言允许程序员使用根据上下文解析的名字。在具有继承层次结构的面向对象语言中(如 Java)，方法名到具体实现的绑定依赖于接收对象(receiver)所属的类，还有相关继承层次结构的状态。如果在编译器进行分析时，继承层次结构和所有过程都已经固定下来，那么编译器可以使用针对类结构的过程间分析尽量减小任何给定调用位置上可调用的方法的集合。对于调用位置处可调用的每个过程或方法，编译器都必须向调用图中添加一条边。

如果语言允许程序在运行时导入可执行代码或新的类定义，则编译器必须构建一个保守的调用图，以反映每个调用位置处所有潜在被调用者的全集。为此，编译器可在调用图中构建一个节点表示未知过程，并对其赋予最坏情形的性质。

如果编译器能够减少可能调用多个过程位置的数目，那么它就能够减少调用图的边(对应于运行时不可能存在的调用)，从而提高调用图的精度。更重要的是，在任何调用位置，只要其调用目标的范围能够缩小到只有一个被调用者，那么它就可以实现为一个简单调用。而调用目标包括多个可能被调用者的调用位置，这可能需要在运行时查找，以便将调用分派到具体实现。

在过程内分析中，假定控制流图具有单入口和单出口，如果过程具有多个返回语句，则添加一个人造的出口节点。在过程间分析中，语言特性可能导致类似的问题。例如，Java语言支持初始器(initializer)和终结器(finalizer)。Java虚拟机会在加载并校验某个类之后调用该类的初始器，并在为对象分配空间之后且返回对象的哈希值之前，调用对象的初始器。

2. 过程间常量传播

过程间常量传播会随着全局变量和参数在调用图上的传播，跟踪其已知常数值，这种跟踪会穿越过程体并跨越调用图的边。过程间常量传播的目标在于发现过程总是接收已知常量值或发现过程总是返回已知常量值的情形。当分析发现这样的常量时，编译器可以对与该值相关的代码进行专门化处理。

过程间常量传播由3个子问题组成：发现常量的初始集合、面向调用图传播已知的常数值以及对值穿越过程的传播进行建模。

分析程序必须在每个调用位置识别出哪些实参具有已知的常数值。有大量技术可实现这一目标。最简单的方法是识别用作参数的字面常数值。一种更有效、代价更高的技术是：使用一个全局常量传播算法识别值为常量的参数。

给定常量的初始集合，分析程序需要跨越调用图的边并穿越过程(从入口点到过程中的每个调用位置)传播常数值。这部分分析类似于迭代数据流算法。但与更简单的问题，如活动变量或可用表达式相比，其使用的迭代数目将显著增多。

分析程序每次处理调用图节点时，都必须判断在过程入口点处已知的常数值是如何影响过程中各个调用位置处已知的常数值集合的。为此，编译器需要为每个实参建立一个小模型，名为跳跃函数(jump function)。具有n个参数的调用位置s会有一个跳跃函数向量$\mathcal{J}_s=(\mathcal{J}_s^a,\mathcal{J}_s^b,\mathcal{J}_s^c\cdots,\mathcal{J}_s^n)$，其中a是被调用者中的第一个形参，b是第二个形参，以此类推。这些形参是包含s的过程p的形参，每个跳跃函数$\mathcal{J}_s^x$都依赖于它们的某个子集的值，将该集合记作Support($\mathcal{J}_s^x$)。

现在假定$\mathcal{J}_s^x$由一个表达式树组成，该表达式树的叶节点都是调用者的形参或字面常数。要求对于任何$y\in$Support($\mathcal{J}_s^x$)，如果Value(y)为T，则返回T。

算法24用于处理跨越调用图的过程间常量传播。该算法将字段Value(x)关联到每个过程p的每一个形参x。算法假定每个形参都具有唯一的名字或完全限定名。在初始化阶段，编译器将所有Value字段都设置为T。接下来，算法遍历程序中每个调用位置s处的每个实参a，将a对应的形参f的Value字段更新为Value(f)$\wedge\mathcal{J}_s^f$，并将f添加到Worklist。这一步骤将跳跃函数表示的初始常量集合分解为多个Value字段，并设置Worklist使之包含所有的形参。

第二阶段，编译器重复地从Worklist中选择一个形参并传播它。为传播过程p的形参f，分析程序会找到p中每个调用位置s，以及每一个满足$f\in$Support($\mathcal{J}_s^x$)的形参x(对应于调用位置s的某个实参)。算法对$\mathcal{J}_s^x$求值并将其合并到Value(x)。如果这使Value(x)发生改变，则算法将x添加到Worklist。用于实现Worklist的数据结构应该具有某种性质

(如稀疏集),使得只允许 x 的一个副本出现在 Worklist 中。

算法 24 过程间常量传播算法

```
1. // 初始化
2. Worklist = Φ
3. for 程序中的每个过程 p do
4.     for p 的每个形参 f  do
5.         Value(f ) = T
6.         Worklist = Worklist ⋃ { f }
7. for 程序中的每个过程调用位置 s do
8.     for 调用位置 s 处的每个实参 a 对应的形参 f do
9.         Value(f ) = Value( f ) ∧ J_s^f
10. // 迭代到不动点
11. while Worklist ≠ Φ do
12.     从 Worklist 中选择一个形参 f
13.     // 更新依赖于 f 的每一个参数的值
14.     for 过程 p 中的每个调用位置 s 和每个满足 f ∈Support(J_s^x) 的形参 x do
15.         t = Value(x)
16.         Value(x) = Value(x) ∧ J_s^x
17.         if Value(x) < t then
18.            Worklist = Worklist ⋃ {x}
19. for 每个过程 p do
20.     CONSTANTS(p) = Φ
21.     for p 的每个形参 f do
22.         if Value(f ) = ⊤then
23.             Value(f ) = ⊥
24.         if Value(f ) ≠ ⊥ then
25.             CONSTANTS(p) = CONSTANTS(p) ⋃{( f ,Value( f ))}
```

算法的第二阶段之所以会停止,是因为每个 Value 至多能取到 3 个格值:⊤、某个 c 和⊥。变量 x 只能在计算出其初始 Value 时或其 Value 改变时进入 Worklist。因此每个变量 x 至多只能出现在 Worklist 上 3 次。因而,改变的总数是有限的,迭代最终会停止。在算法的第二阶段停止之后,会有一个后处理步骤来构建每个过程入口处已知的常量集合。

跳跃函数可以通过简单的静态近似实现,也可以通过小的参数化模型实现,还可以通过比较复杂的方案实现,即在每次对跳跃函数求值时进行广泛分析。在上述任何一种方案中,有几个原则都是成立的:如果分析程序判断调用位置 s 处的参数 x 是一个已知常量 c,那么$\mathcal{J}_s^x$=c 且有 Support($\mathcal{J}_s^x$)=Φ;如果 y∈Support($\mathcal{J}_s^x$)且 Value(y)=⊤,那么 $\mathcal{J}_s^x$=⊤;如果分析程序判断的值是无法确定的,那么$\mathcal{J}_s^x$=⊥。例如,如果 Support($\mathcal{J}_s^x$) 包含一个从文件中读取的值,则$\mathcal{J}_s^x$=⊥。

分析程序可以用许多方法实现$\mathcal{J}_s^x$。简单的实现方案:仅当 x 是包含 s 的过程中的一个形参的静态单赋值形式名时才传播常量。较为复杂的方案可以建立由形参和字面常数的静态单赋值形式名组成的表达式。

算法 24 只沿调用图的边正向传播值为常数的实参。可以用一种直截了当的方法扩展

该算法，以处理过程中具有全局作用域的返回值和变量。

既然算法可以建立跳跃函数来模拟值从调用者流动到被调用者，那么它也可以构建回跳函数（return jump function）来模拟值从被调用者返回到调用者。回跳函数对于初始化值的例程特别重要，无论是 FORTRAN 中由公用块填写数据，还是 Java 中为对象或类设置初始值的操作。算法可以用处理普通跳跃函数的方式来处置回跳函数。但要注意的情况是，实现必须避免创建回跳层数的环，这种情况会导致脱节（例如，对于出现尾递归的过程）。

为扩展算法使之涵盖更大的一类变量，编译器只需用一种适当的方法简单地扩展跳跃函数向量。扩展变量集合将增加分析的代价，但有两个因素可以减少代价。首先，在构建跳跃函数时，分析程序可以注意到其中许多变量没有一个能够轻易模拟的值。算法可以将这些变量映射到一个返回 $\perp$ 的通用跳跃函数，从而避免将这些变量置于 Worklist 上。其次，对于可能有常数值的变量来说，格的结构确保了它们至多在 Worklist 上出现两次，因而算法仍然运行得很快。

5.4.2　优化

过程调用将一个程序划分为多个过程，对于编译器生成高效代码的能力具有正反两方面的影响。从正面来看，它限制了编译器在任一时刻需要考虑的代码数量，这使得编译时数据结构保持在比较小的尺寸上，同时通过对问题规模的约束也限制了各种编译时算法的代价。从负面来看，将程序划分为过程，限制了编译器理解被调用过程内部行为的能力。

由过程调用引入的低效性的第二个主要来源是，对每个调用来说，调用者中必定执行一个调用前代码序列和一个返回后代码序列，同时被调用者中必定要执行一个起始代码序列和一个收尾代码序列。实现这些代码序列的操作是要花费时间的，且这些代码序列之间的转移需要跳转（可能是破坏性的）。在一般情形下，这些操作都是为实现源语言中的抽象而引入的开销。但对于任何特定调用来说，编译器也许能对这些代码序列或被调用者进行某种改进，使之适应局部运行时环境并实现更好的性能。

过程调用对于编译时支持和运行时操作的这些影响，会引入过程内优化无法解决的低效性。为减少独立过程引入的低效性，编译器可以使用过程间分析和优化技术，同时对多个过程进行分析和变换。

本节将介绍两种不同的过程间优化技术：过程调用的内联替换和过程置放。因为全程序优化要求编译器能够访问被分析和变换的代码，对编译器结构有一些隐含的要求，所以本节最后将探讨在包含过程间分析和优化机制的编译器系统中会出现的结构性问题。

1. 内联替换

编译器为实现过程调用而必须生成的代码涉及很多操作。生成的代码必须分配一个活动记录、对每个实参求值、保存调用者的状态、创建被调用者的环境、将控制从调用者转移到被调用者（以及与之对应的反向转移）以及把返回值从被调用者传递给调用者。在某种意义上，这些运行时活动是使用编程语言的一部分固有开销，它们维护了编程语言本身的抽象，但严格来

说，这些抽象对于结果的计算并不是必需的。优化编译器试图减少此类开销的代价。

有时候，编译器可以通过将被调用者过程体的副本替换到调用位置上，并根据具体调用位置进行适当的调整，来提高最终代码的效率。这种变换称为内联替换（inline substitution）。内联替换是一种变换，它将一个调用位置替换为被调用者过程体的一个副本，并重写代码以反映参数的绑定。内联替换能够避免大部分过程链接代码，还可以根据调用者的上下文，对被调用者过程体的新副本进行调整。因为该变换代码从一个过程移动到另一个过程中，还会改变程序的调用图，所以内联替换是一种过程间变换。

类似于许多其他优化，内联替换很自然地划分为两个子问题：实际的变换和选择需要内联的调用位置的决策过程。变换本身相对简单，决策过程更为复杂且对性能有直接影响。

1）变换

为进行内联替换，编译器需要用被调用者过程体重写一个调用位置，同时需要适当修改过程体副本，以模拟参数绑定的效果。图 5-9 给出了两个过程 fee 和 fie，二者都调用了另一个过程 foe。图 5-10 描述了将 fie 中对 foe 的调用进行内联之后的控制流。编译器已经创建了 foe 的一个副本并将其移动到 fie 内部，将 fie 中的调用前代码序列直接连接到 foe 副本的起始代码序列，同时用类似的方式将 foe 副本的收尾代码序列直接连接到 fie 中的返回后代码序列。由此生成的代码中，一部分基本程序块是可以合并的，可通过后续的优化继续改进代码。

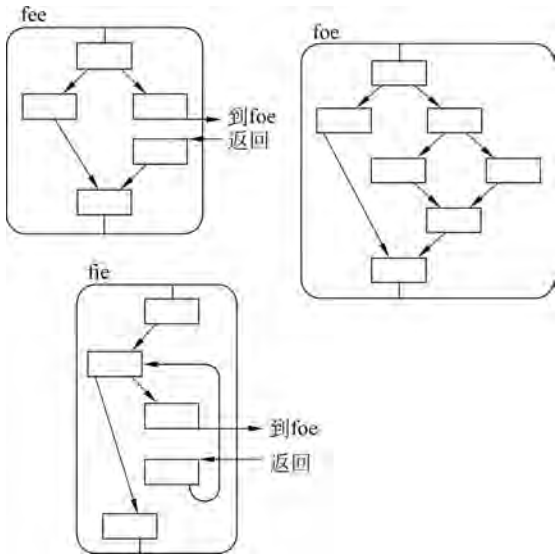

图 5-9　内联替换之前

当然，编译器必须使用一种能够表示内联过程（Inlined Procedure）的中间形式。一些源语言结构能够导致结果代码中出现任意且罕见的控制流结构。例如，如果被调用者带有多个过早的返回语句，则会产生一个复杂的控制流图。类似地，FORTRAN 交错返回

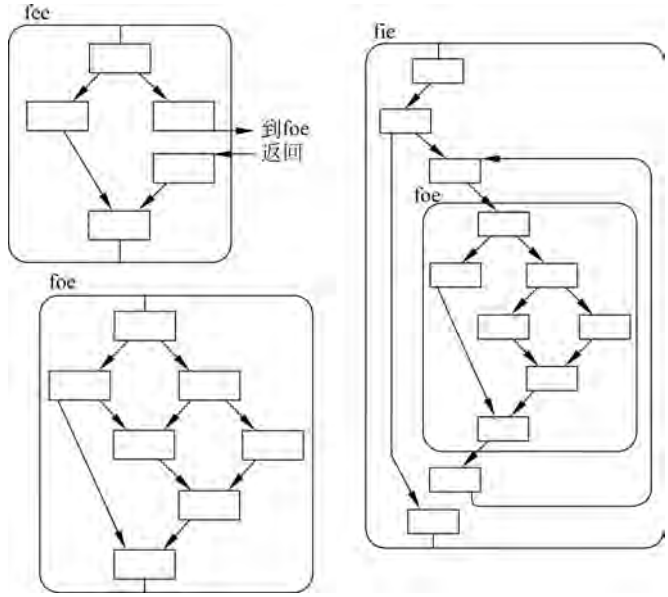

图 5-10　内联替换之后

(alternate return)结构允许调用者向被调用者传递标号,然后被调用者可以使控制返回到这些标号中的任何一个。但不论是哪种情况,最终产生的控制流图可能都很难接近源代码的抽象语法树。

在实现时,编译器编写者应该注意局部变量的“增殖”问题。一个简单的实现策略可能会在调用者中为被调用者中的每个局部变量分别创建一个与之对应的新的局部变量。如果编译器内联了几个过程,或在几个调用位置内联了同一被调用者,那么局部命名空间可能会变得相当庞大。虽然命名空间的增长并不导致正确性问题,但它可能会增加对变换后代码进行编译的代价,有时候也会损害最终代码的性能。关注这一细节,就可以轻易避免该问题,只需让跨越多个内联的被调用者重用名字即可。

2) 决策过程

选择对哪些调用位置进行内联是一个复杂任务。内联一个给定的调用位置可能会提高性能,也可能导致性能下降。为做出明智的选择,编译器必须在一个颇为宽泛的范围内考查调用者、被调用者和调用位置的特征。编译器还必须了解其自身的优势和劣势。

通过内联,代码性能改进的主要来源是直接消除一部分操作,以及提高其他优化的有效性。在剔除了一部分链接代码序列后会出现前一种效应。例如,可以去掉寄存器保存和恢复代码,而由寄存器分配器做出这方面的决策。来自调用者的知识还可用于证明被调用者内部的其他代码是死代码或无用代码。而后一种效应则源自为全局优化提供了更多的上下文信息。

内联替换导致性能降低的主要来源是对结果代码进行代码优化的有效性降低。内联被调用者会增加代码长度和命名空间规模。这会导致在原来的调用位置附近对寄存器需

求的增加。而消除寄存器保存和恢复代码则改变了寄存器分配器“眼中”的问题。实际上，这些效应中的任何一项都可能导致优化有效性的下降。

体系结构中的改变，比如更大的寄存器集合，可能导致过程调用代价增加。这些改变进而又使得内联更具吸引力。

在每个调用位置上，编译器必须决定是否内联该调用。使问题进一步复杂化的是，一个调用位置上所做的决策会影响到其他调用位置上的决策。例如，如果 a 调用 b，b 又调用 c，则选定将 c 内联到 b 中不仅会改变内联到 a 中的过程的特征，还会改变底层程序的调用图。此外，必须从整个程序的角度来考虑内联带来的一些效应，如代码长度的增长，但编译器编写者可能想要限制代码的总长度。

内联替换的决策过程会在每个调用位置根据多条准则来考查，其中包括如下几条：

(1) 被调用者规模。如果被调用者的代码长度小于过程链接代码(调用前代码序列、返回后代码序列、起始代码序列、收尾代码序列)，那么内联此类被调用者应该会减少代码长度，实际执行的操作数也会减少。这种情况经常出现。

(2) 调用者规模。编译器可能会限制任何过程的总长度，以缓解编译时间的增加和优化有效性的降低。

(3) 动态调用计数。与对很少执行的调用位置进行同等改进相比，对频繁执行的调用位置进行改进，能够提供更大的收益。实际上，编译器会使用剖析数据或简单估算来对调用位置的执行次数进行计数。

(4) 常数值实参。在调用位置处使用具有已知常数值的实参，会产生一种对代码进行改进的潜在可能性：将这些常数合并到被调用者的过程体中。

(5) 静态调用计数。编译器通常会跟踪调用同一过程的不同调用位置的数目。只从一个调用位置处被调用的过程可以安全地内联而不会带来代码长度的增长。编译器在进行内联操作的同时应该更新这个度量数据，以检测因内联替换的进行而减少到只余一个调用位置的那些过程。

(6) 参数计数。参数的数目可以充当过程链接代价的一种表示，因为编译器必须生成代码以对每个实参求值并存储。

(7) 过程中的调用。跟踪过程中调用的数目，这提供了一种很容易的方法来检查调用图中的叶节点，即不包含调用的过程，称为叶过程，它通常是良好的内联候选者。

(8) 循环嵌套深度。循环中的调用位置，比循环以外的调用位置执行得更为频繁。它们还破坏了编译器将循环作为单个单元进行调度的能力。

(9) 占执行时间的比例。根据剖析数据计算每个过程占执行时间的比例，可以防止编译器内联那些对性能影响不大的例程。

2. 过程置放

给定一个程序的调用图，其中标注了每个调用位置测量或估算的执行频度，需要重排各个过程以减小虚拟内存工作集的规模，并限制调用引起指令高速缓存中冲突的可能性。

如果过程 p 调用 q，则希望 p 和 q 占用相邻的内存位置。

为解决这个问题，可以将调用图当作可执行代码中各个过程相对位置上的一组约束来处理。调用图的每条边(p,q)规定了可执行代码中应该存在的一组相邻关系。遗憾的是，编译器无法满足全部相邻关系。例如，如果 p 调用 q、r 和 s，则编译器无法将后三者都放置到与 p 相邻的位置上。因而，编译器执行过程置放时，倾向于使用一种贪婪算法来寻找一种良好的置放方式，而非试图计算最优置放方式。

采用过程置放算法建立过程链，可以使用位于过程链中部的过程之间的边，因为该算法的目标就是使各个过程彼此接近，从而减小工作集规模并减少对指令高速缓存的干扰。如果 p 调用 q，且从 p 到 q 的距离小于指令高速缓存的容量，那么这种置放方式就是成功的。

过程置放由两个阶段组成：分析和变换。分析阶段在程序的调用图上运作。算法重复地在调用图中选择两个节点并合并二者。合并的次序由执行频度数据驱动，频度数据是测量或估算而来的。合并的次序决定了最终的代码布局。变换阶段比较直接，此时只需按照分析阶段选择的次序重排各个过程的代码即可。

算法 25 给出了一个用于过程置放分析阶段的贪婪算法。

算法 25　过程置放算法

```
1. 初始化工作
2. 构建调用图 G
3. 初始化优先队列 Q
4. for 图 G 中的每条边 (x, y) do
5.     if (x == y) then
6.         从 G 中删除 (x, y)
7.     else
8.         weight((x, y)) = 边 (x, y) 的估算执行频度
9. for 图 G 中的每个节点 x do
10.    list(x) = {x}
11.    if 从 x 到 y 存在多条边 then
12.        合并它们和它们的权重
13.    for 图 G 中的每条边 (x, z) do
14.        Enqueue(Q, (x, z), weight((x, z))
15. //以迭代方式建立过程置放的一种次序
16. while Q 不为空 do
17.    (x, y) = Dequeue(Q)
18.    for 图 G 中的每条边 (y, z) do
19.        ReSource((y, z), x)
20.    for 图 G 中的每条边 (z, y) do
21.        ReTarget((z, y), x)
22.    将 list(y) 追加到 list(x)
23.    从图 G 中删除 y 和与之相连的边
```

该算法在程序的调用图上运作，按估算的执行频度次序来考查调用图中的各条边，从而以迭代方式构建一种置放方式。算法在第一步建立调用图，为各条边分配与估算的执行频度相对应的权重，然后将两个节点之间的所有边合并为一条边。作为初始化工作的最后一部分，算法为调用图的边建立一个优先队列，按边的权重排序。

算法的后半部分以迭代方式建立过程置放的一种次序。该算法将调用图中的每个节点关联到过程的一个有序表。这个表规定了各个有名字过程之间的一种线性序。在该算法停止时，这个表规定了各个过程上的一个全序，可利用该全序在可执行代码中置放各个过程。

算法使用调用图中各条边的权重来引导这一处理过程。它重复地从优先队列中选择权重最高的边，假定为(x,y)，并合并边的源(source)x 和目标(sink)y。接下来，算法必须更新调用图以反映这种变化：

(1) 算法对每条边(y,z)调用 ReSource，将(y,z)替换为(x,z)并更新优先队列。如果边(x,z)已经存在，则 ReSource 将二者合并。

(2) 算法对每条边(z,y)调用 ReTarget，将(z,y)替换为(z,x)并更新优先队列。如果边(z,x)已经存在，则 ReTarget 将二者合并。

为使过程 y 放置到 x 之后，算法将 list(y)追加到 list(x)。最后，算法从调用图中删除 y 和与之相连的边。

该算法在优先队列为空时停止。在最终形成的图中，每个节点都与原始调用图中的某个连通分量一一对应。如果从表示程序入口点的节点出发可以到达调用图中所有的节点，那么最终的图中将只有一个节点。如果某些过程是不可到达的，因为程序中不存在调用这些过程的代码路径，或者这些路径被具有二义性的调用掩盖，那么最终的图中将包括多个节点。不管是哪种情形，编译器和链接器都可以使用与最终的图中各个节点相关联的列表，来规定各个过程的相对置放次序。

为了解过程置放算法的工作方式，考虑图 5-11 画面 0 中给出的示例调用图。P_5有一条到其自身的边，但是因为环的源和目的相同，所以这种边不影响置换算法。

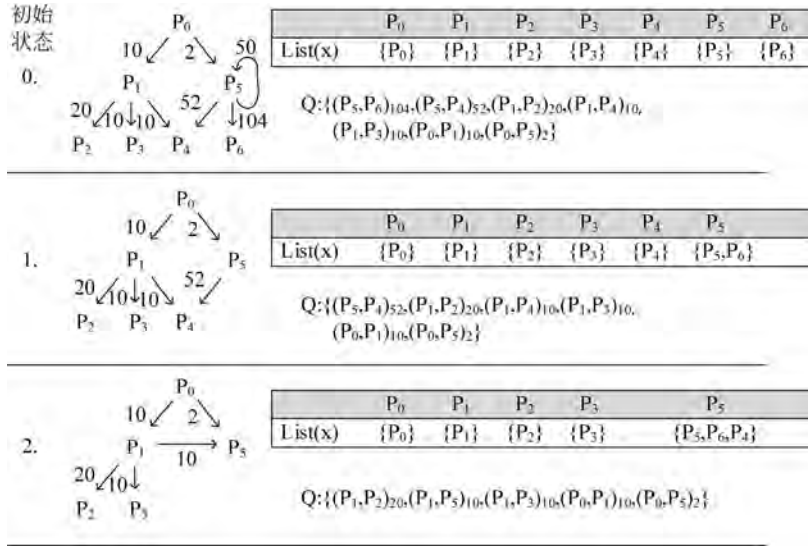

图 5-11 过程置放算法的步骤(1)

画面 0 给出了该算法在迭代归约即将开始时的状态。每个节点对应的列表都是平凡的，只包含其自身的名字。优先队列使图中每条边（环除外）根据执行频度排序。

画面 1 给出了该算法在 while 循环完成第一次迭代之后的状态。算法将 P_6 坍缩（collapse）到 P_5，并更新对应于 P_5 的列表和优先队列。

在画面 2 中，算法已经将 P_4 坍缩到 P_5。它将边（P_1，P_4）的目标重定向到 P_5，并改变了优先队列中对应边的名字。此外，它从图中删除了 P_4 并更新了对应于 P_5 的列表。

如图 5-12 所示，其他迭代以类似的方式进行。画面 4 给出了算法合并边的场景。此时，算法将 P_5 坍缩到 P_1，并将边（P_0，P_5）的目标重定向到 P_1。因为（P_0，P_1）已经存在，算法只是合并了新旧两条边的权重，并相应地更新优先队列：删除（P_0，P_5），并改变（P_0，P_1）的权重。

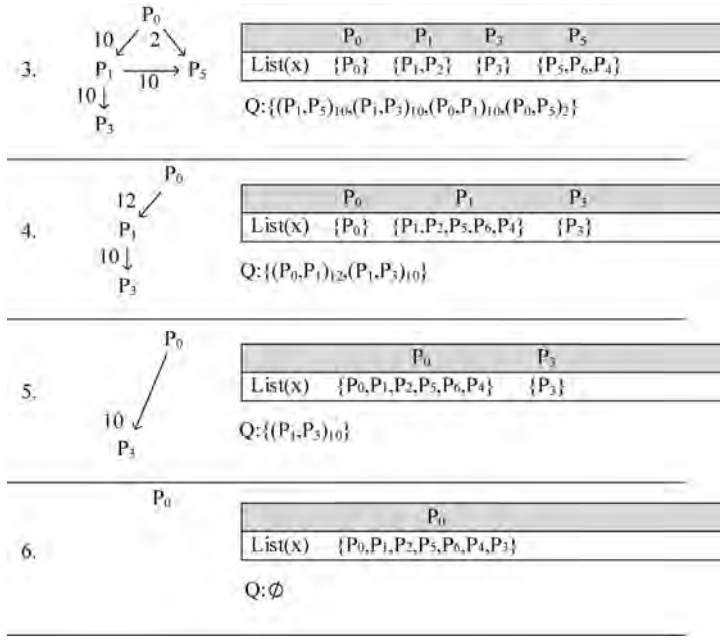

图 5-12　过程置放算法的步骤（2）

在各次迭代结束后，调用图已经坍缩到一个节点 P_0。虽然这个例子构建的布局从入口节点开始，但这是由各条边的权重所致，而非算法设计如此。

3. 针对过程间优化的编译器组织结构

建立一个跨两个或更多过程进行分析和优化的编译器，可以从根本上改变编译器与其所生成代码之间的关系。程序在输入编译器时，通常划分为多个部分，这些部分通常称为编译单元。对于传统的编译器来说，编译单元可能是单个过程、单个类或单个代码文件，编译生成的结果代码完全取决于对应编译单元的内容。一旦编译器使用关于某个过程的知识来优化另一个过程，则结果代码的正确性同时取决于两个过程的状态。

考虑内联替换对优化后代码正确性的影响，假定编译器将 fie 内联到 fee 中，任何后续

对 fie 的编辑修改都必将导致重新编译 fee，这是因优化决策而导致的依赖性，而非源代码中暴露的任何关系所致。

如果编译器收集并使用过程间信息，则可能会出现类似的问题。例如，fee 可能调用 fie，fie 又调用 foe。假定编译器需要依赖一个事实：对 fie 的调用不会改变全局变量 x 的已知常量值。如果程序员后来编辑 foe 导致该过程修改了 x，这一改变会使此前对 fee 和 fie 的编译无效，因为优化所依赖的事实已经发生了改变。因而，对 foe 的修改可能导致必须重新编译程序中的其他过程。

为解决这种固有的问题，并使得编译器能够访问其需要的所有源代码，人们提出了能够进行全程序或过程间优化的几种不同的编译器结构：扩大编译单元、在集成开发环境中嵌入编译器以及在链接时进行优化。

(1) 扩大编译单元。对于过程间优化引入的实际问题，最简单的解决方案是扩大编译单元。如果编译器只在一个编译单元内考虑优化和分析，且这些单元具有某种一致的划分方式，那么编译器可以规避前文提到的问题。这样，编译器只能分析并优化同时编译的代码，因而无法在编译单元之间引入依赖性，也不需要访问其他编译单元的源代码或相关事实。当然，这种方法限制了过程间优化的机会，它还促使程序员创建较大的编译单元，并将彼此调用的过程群集到一起，这在具有多个程序员的系统中会引入实际问题。即便如此，这种组织方式仍然是有吸引力的，因为它对编译器行为模型的干扰最少。

(2) 集成开发环境。如果对编译器组织结构的设计将编译器嵌入一个集成开发环境(Integrated Development Environment，IDE)内部，那么编译器就可以通过 IDE 按需访问源代码。在源代码发生改变时，IDE 可以通知编译器，这样编译器能够确定是否需要重新编译。这种模型将源代码和编译后代码的所有权从开发者移交给 IDE。接下来，IDE 和编译器之间的协作可以确保采取适当的行动来保证一致且正确的优化。

(3) 链接时优化。编译器编写者可以将过程间优化转移到链接器中，其中可以访问所有静态链接的代码。为获得过程间优化的收益，链接器可能还需要执行后续的全局优化。因为链接时优化的结果只记录在可执行文件中，而该可执行文件将在下一次编译时丢弃，这种策略绕过了重新编译问题。与其他方法相比，这种方法会执行更多的分析和优化，但它同时具备简单性和正确性。

5.5 循环优化

循环(loop)是一个指令序列，它重复执行直到满足终止条件为止。循环在计算机程序中广泛存在，程序的大部分执行时间用于执行某个循环。因此，专门优化循环的执行效率非常有价值。直观上讲，循环从序列结尾又跳回到序列开始的指令序列。但是为了高效地优化循环，将给出对循环更准确的定义。

控制流图中的循环是一个包含满足以下性质的头节点(header node)h 的节点集合 S：

(1) S 中的每个节点都有一条由通向 h 的有向边构成的路径；

(2) 从 h 到 S 中的任意节点,都有一条有向边路径;

(3) 除 h 外,不存在任何从 S 外节点到 S 内其他节点的边。

图 5-13 给出了几个循环的示例,其中各例中 1 是头节点。循环的入口(loop entry)节点是有一个前驱位于循环外的节点,循环的出口节点是有一个后继位于循环外的节点。图 5-13(c)、(d)和(e)说明了循环可以有多个出口节点,但是只能有一个入口节点。图 5-13(e)和(f)包含嵌套循环。

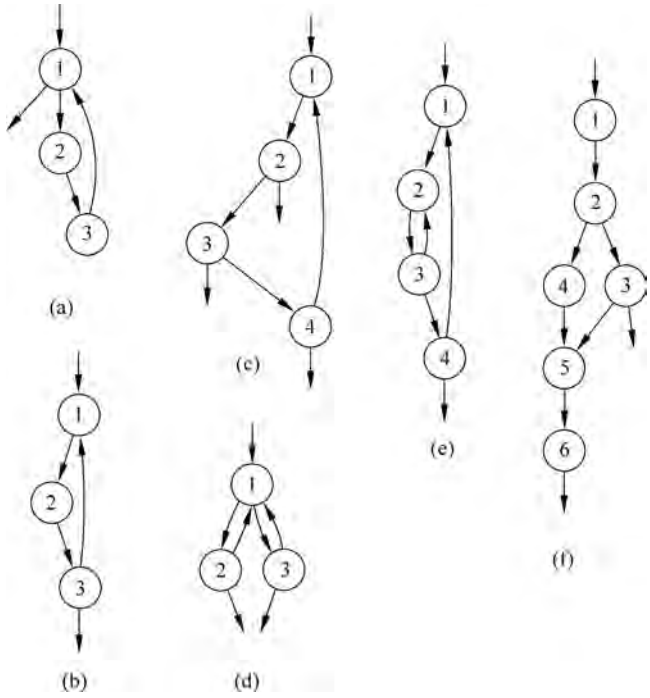

图 5-13　循环示例

图 5-14(a)不包含循环,在强连通部分中的两个节点 (2,3) 都可以不经过对方而到达。图 5-14(c)中分别包含节点 1、2 和 3 的 3 个图形之间的关系和图 5-14(a)相同,如果重复地删除所有 x 是 y 的唯一前驱的边 x→y,并且合并这一对节点 (x,y),则可以看到:删除 6→9,5→4,合并 (7,9)、(3,7)、(7,8)、(5,6)、(1,5)、(1,4),就可以得到图 5-14(a)。虚线指出了通过删除边和折叠节点对图 5-14(c)的归约。

不可归约流图(irreducible flow graph)是指这样的图:合并节点和删除边后,在图中可以找到与图 5-14(a)相同的子图。可归约流图(reducible flow graph)是合并后不包含这种子图的流图。在没有这样的子图时,节点的任何环路都只有唯一的头节点。常见的控制流结构,如 if-then、if-then-else、while-do、repeat-until、for 和 break(甚至多级 break)都只能生成可归约流图。因此,Java 的方法或者不带 goto 语句的 C 函数的控制流图,总是可归约的。

可归约流图的优点是许多数据流分析都能在可归约流图上高效进行。此时,不需要使用不动点迭代(即迭代地执行赋值,直到结果不再发生改变),就可以确定出计算这些赋值

的顺序，并且提前计算出需要多少次赋值，即不再需要检查是否发生了任何改变。

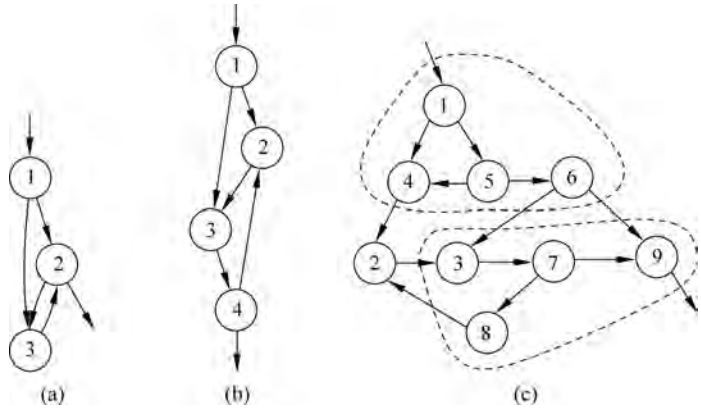

图 5-14 所有流图都不包含循环

5.5.1 循环

图 5-15 展示了一个流图和它的必经节点树。

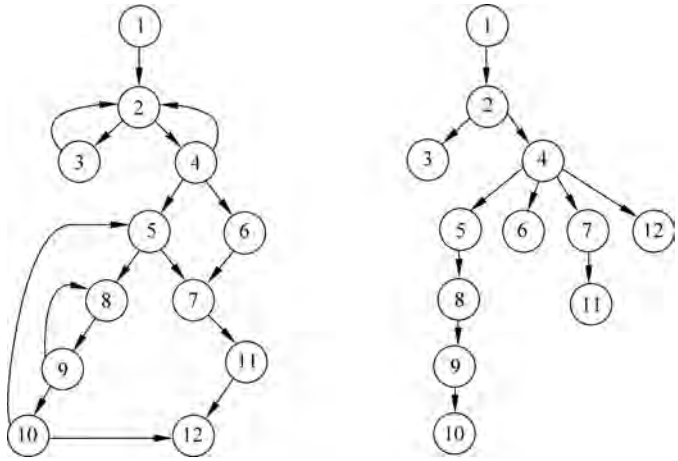

图 5-15 流图和它的必经节点树

流图中从一个节点 n 到它的必经节点 h 的边称为回边，对于每条回边，对应地存在着一个构成循环的子图。回边 n→h，其中 h 是 n 的必经节点，对应的自然循环(natural loop)是由满足下列条件的所有节点 x 组成的集合：x 的必经节点是 h，并且有一条从 x 到 n 的路径不包含 h。这个循环的头(header)是 h。

图 5-15 中流图的回边 10→5 的自然循环包括节点{5,8,9,10}，并且内部有一个由节点{8,9}构成的嵌套循环。

如果有多条回边到达节点 h，那么 h 就是多个自然循环的头。在图 5-15 的流图中，3→

2 对应的自然循环由节点{3,2}组成，4→2 对应的自然循环由节点{4,2}组成。

本节讨论的循环优化适用于任何循环，不管循环是否是自然循环，或者是否和其他循环共享一个循环头。但是，因为内层循环占用了大多数程序执行时间，所以通常希望首先优化内层循环。如果两个循环共享一个循环头，就很难判断应将哪一个循环看作内层循环。解决这一问题的通用方法是合并共享同一个头的所有循环。但合并后的循环不一定是一个自然循环。

如果合并图 5-15 中流图的循环头为 2 的两个循环，得到的这个循环将包括节点 2、3、4，该循环不是一个自然循环。

如果 A 和 B 是头分别为 a 和 b 的两个循环，其中 a≠b，并且 b 在 A 中，则 B 的节点是 A 的节点的真子集，就称 B 嵌套在 A 的内部，或者说 B 是内层循环。

可以构建程序中循环的循环嵌套树(loop-nest tree)，构建流图 G 的循环嵌套树的过程如下：

(1) 计算 G 的必经节点；

(2) 构建必经节点树；

(3) 找出所有的自然循环，以及所有的循环头节点；

(4) 对每个循环头节点 h，将所有头为 h 的自然循环合并成一个循环 loop[h]；

(5) 构建循环头节点 h(以及隐含的循环)的树，如果 h_2 在循环 loop[h_1] 中，则在树中，loop[h_2] 在 loop[h_1] 之下。

这种循环嵌套树的叶子节点是最内层循环。

为了在循环嵌套树中有一个位置放置不属于任何循环的节点，可以将整个过程体看成是一个位于循环嵌套树的根的伪循环。图 5-15 的循环嵌套树如图 5-16 所示，循环头(节点 1、2、5、8)位于每个椭圆的上半部，一个循环包括一个循环头(例如节点 5)、同一个椭圆中的所有其他节点(例如节点 10)以及以该椭圆为根的循环嵌套子树中的所有节点(例如节点 8、9)。

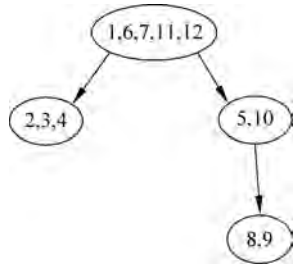

图 5-16　图 5-15 流图的循环嵌套树

许多循环优化需要在紧挨着循环执行之前插入一些语句。例如，循环不变量外提会将一条语句从循环内移动到紧挨循环之前。这些语句应该放在哪里呢？图 5-17(a)举例说明了这个问题：如果想要将语句 s 插入循环之前紧挨着循环的一个基本块中，则需要将 s 同时放到基本块 2 和 3 的末尾。为了有一个统一的位置放置这些语句，在循环外插入了一个新的、初始为空的前置节点(preheader)p 和一条边 p→h。所有从循环内的节点 x 到 h 的边 x→h 都不会发生改变，但是所有从循环外节点 y 到 h 的边 y→h 都将重定向到 p。

如果循环中包含语句 t＝a⊕b，并且在循环的每一轮执行中，a 的值都相同，b 的值也相同，那么 t 每次也会具有相同的值，则 t 称为循环不变量(loop invariant)。可以将这个计算提升到循环之外，这样该计算就只需要执行一次，而不是每次迭代都执行。这种优化称为

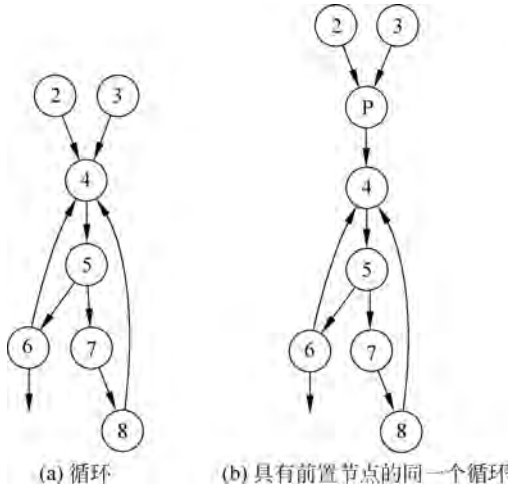

图 5-17 循环及加入前置节点的循环

循环不变量外提(Loop Invariant Code Motion,LICM)。

为了静态估计变量 a 的动态值,编译器需要对 a 的值进行保守估计。如果每个操作数 a,都满足下列条件之一,则循环 L 中的定值

$$d: t = a_1 \oplus a_2$$

是循环不变量,其中:

(1) a_i是常数;

(2) 或者所有到达 d 的 a_i 的定值都在循环之外;

(3) 或者 a_i只有一个定值 e:a_i=…到达 d,并且该定值 e 是循环不变量。

根据上面的条件,可以给出一个迭代算法来找出循环不变量的定值:首先找出所有操作数是常数或者来自循环之外的定值,然后重复地寻找其操作数都为循环不变量的定值。这是一个典型的不动点算法,可以给出一个基于工作表的高效实现。

假设 t=a ⊕b 是循环不变量,能够将它提升到循环之外吗?在图 5-18(a)中,将它外提可以使程序运行较快,并仍能得到相同的计算结果。但是在图 5-18(b)中,外提它虽然可使程序运行得更快,但是结果却不正确:原来的程序并不一定会执行 t=a⊕b,但是转换后的程序却总是执行它,如果一开始就有 i≥N,转换后的程序会产生错误的 x 值。图 5-18(c)中,外提 t=a⊕b 也是不正确的,因为原循环中有多个对 t 的定值,转换后的程序会以不同的交替方式对 t 赋值。在图 5-18(d)中进行外提同样是错误的,因为在此循环不变量定值之前有一个对 t 的使用,因此,将此定值外提后,循环的第一次迭代会使用错误的值。

综合考虑以上问题,可以给出以下将定值 d:t=a_1⊕ a_2 外提到循环前置节点末尾的判定标准。

(1) d 是所有这样的循环出口节点的必经节点:在这些循环出口节点,变量 t 在出口是活跃的;

(2) 在循环中 t 只有一个定值;

```
L0:                L0 :               L0 :               L0 :
   t=0                t = 0              t = 0              t = 0
L1:                L1 :               L1 :               L1 :
   i = i + 1          if i ≥ N           i = i + 1          M[i] = t
   t = a ⊕ b            goto L2          t = a ⊕ b          i = i + 1
   M[i] = t           i = i + 1          M[i] = t           t = a ⊕ b
   if i < N           t = a ⊕ b          t = 0              M[i] = t
     goto L1          M[i] = t           M[i] = t           if i < N
L2 :                  goto L1            if i < N             goto L1
   x = t           L2 :                    goto L1        L2 :
                      x = t            L2 :                  x = t
```

(a) 外提　(b) 不能外提　(c) 不能外提　(d) 不能外提

图 5-18　外提 t＝a ⊕b 的正确候选和不正确候选

(3) 并且 t 不属于循环前置节点的出口活跃集合。

在循环不变式外提中，还必须考虑指令隐含的副作用。例如，如果指令 $t=a_1 \oplus a_2$ 可能引发某类算术异常或者有其他副作用，上述规则就需要做一些修改对这些情况加以限制。

上述条件(1)比较苛刻，它会阻碍从 while 循环中外提许多计算。从图 5-19(a)可以看到，循环体中没有一条语句是循环出口节点(它同时也是循环头节点)的必经节点。为了解决这个问题，可以将 while 循环转换为其前有一条 if 语句的 repeat 循环。这种转换需要复制头节点中的语句，如图 5-19(b)所示。当然 repeat 循环体中的所有语句都是循环出口节

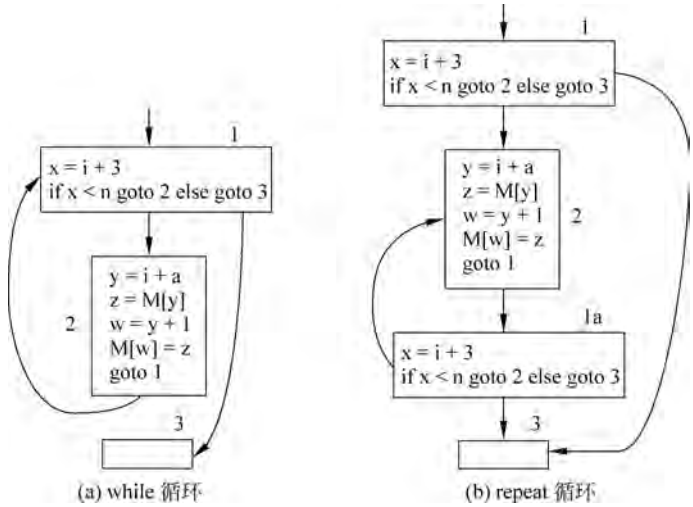

图 5-19　将一个 while 循环转换为 repeat 循环

点的必经节点(如果没有 break 或显式的循环退出语句),这样便能满足条件(1)。

5.5.2 归纳变量

某些循环中,存在一个递增或递减的变量 i,以及一个在循环中被赋值

$$j = i * c + d$$

的变量 j,其中 c 和 d 是循环不变量。于是可以在不引用 i 的情况下计算 j 的值,即只要 i 以 δ 的幅度递增,就可以用 c * δ 递增 j。

例如,表 5-7 中左侧的程序计算一个数组的所有元素之和。利用归纳变量分析(induction-variable analysis),编译器可以发现 i 和 j 是相关的归纳变量,通过强度削弱(strength reduction)操作,编译器可以用加法替代乘以 4 的乘法,然后通过归纳变量删除(induction-variable elimination)可以将 i ⩾n 替换为 k⩾ 4n+a,最后通过各种复写传播,可以得到表 5-7 中右侧的程序。转换后的这个循环包含的四元式更少,运行也会更快。下面分步介绍这一系列的转换。

表 5-7 归纳变量优化之前和之后的循环

优 化 之 前	优 化 之 后
s = 0	s = 0
i = 0	k' = a
L_1 : if i ⩾ n goto L_2	b = n * 4
j = i * 4	c = a + b
k = j + a	L_1 : if k' ⩾ c goto L_2
x = M[k]	x = M[k']
s = s + x	s = s + x
i = i + 1	k' = k' + 4
goto L_1	goto L_1
L_2	L_2

像表 5-7 左侧程序中 i 这样的变量称为基本归纳变量(basic induction variable),j 和 k 是和 i 同族的导出归纳变量(derived induction variable)。在原始循环中,j 被定值后,有

$$j = a_j + i * b_j$$

其中,$a_j = 0$,$b_j = 4$。完全可以用(i,a,b)来刻画 j 在它的定值点的值,其中 i 是基本归纳变量,a 和 b 是循环不变表达式。

如果有另一个导出归纳变量 k,具有定值 k=j+c_k(其中 c_k是循环不变量),则 k 也和 i 同族。可以用三元组(i,c_k,b_j)来刻画 k,即

$$k = c_k + i * b_j$$

也可以以同样的方式用三元组(i,0,1)刻画基本归纳变量 i,这意味着 i=0+i·1。这样,每个归纳变量就都可以用这种三元组来刻画了。

如果一个归纳变量在循环的每次迭代中都改变相同的量(常数或循环不变量),就称它为线性归纳变量(linear induction variable)。在表 5-8 中,左侧的程序中归纳变量 i 不是线性的:在某些迭代中,它递增 b,在其他迭代中,它递增 1。此外,在一些迭代中 j=i·4,而

在另外一些迭代中，导出归纳变量 j 并不随 i 的递增而增加。

表 5-8　非线性归纳变量优化之前和之后的循环

优 化 之 前	优 化 之 后
s = 0 L_1: if s ⩾ 0 goto L_2 i = i + b j = i * 4 x = M[j] s = s − x goto L_1 L_2: i = i + 1 s = s + j if i < n goto L_1	s = 0 j' = i * 4 b' = b * 4 n' = n * 4 L_1: if s ⩾ 0 goto L_2 j' = j' + b' j = j' x = M[j] s = s − x goto L_1 L_2: j' = j' + 4 s = s + j if j' < n' goto L_1

1. 发现归纳变量

如果在以 h 为头节点的循环 L 中，变量 i 只有一个形如 i=i+c 或者 i=i−c 的定值，其中 c 是循环不变量，那么 i 就是循环 L 的一个基本归纳变量。

如果变量 k 满足下列条件，那么 k 是循环 L 中的导出归纳变量。

(1) L 中 K 只有一个形如 k=j * c 或者 k=j+d 的定值，其中 j 是一个归纳变量，c 和 d 是循环不变量；

(2) 并且当 j 是和 i 同族的导出归纳变量时，有

① 到达 k 的 j 的唯一定值是 j 在循环中的那个定值；

② 并且在 j 的定值到 k 的定值之间的任何路径上都没有 i 的定值。

假设 j 是用(i,a,b)来刻画的，则根据 k 的定值是 j·c 还是 j+d，可以用(i,a* c,b* c)或者(i,a+d,b) 来描述 k。

为了进行归纳变量分析，形如 k=j −c 的语句可以作为 k=j+(−c) 来对待，除非−c 是不可表示的，在 2 的补码算术中有时可能会发生这种情况。

形如 k=j/c 的语句可以重写为 k=j*(1/c)，因此 k 可以看成是一个归纳变量。这样的重写只适合于浮点计算，因为如果这是一个整数除法，就不能表示为 1/c。

2. 强度削弱

在许多机器上，乘法比加法的代价高得多，因此希望找出定值形式为 j=i * c 的导出归纳变量，并用一个加法来替代它。

强度削弱需要对三元组为(i,a,b)的每一个导出归纳变量 j，构造一个新的变量 j′(尽管具有相同三元组的不同导出归纳变量可以共享同一个 j′)。在每个赋值 i=i+c 之后，构造一个赋值 j′=j′+c * b，其中 c * b 是可以在循环前置节点计算的循环不变量表达式。如果 c 和 b 都是常数，则乘法 c * b 可以在编译时完成计算。用 j=j′ 替换循环中这个对 j 的唯一

赋值。最后，需要在循环前置节点的末尾用 $j'=a+i*b$ 初始化 j'。

对于 i 族的两个归纳变量 x 和 y，在循环的执行期间，除了在语句序列 $z_i=z_i+c_i$ 中（c_i 是循环不变量）之外，如果每次都有

$$\frac{x-a_x}{b_x}=\frac{y-a_y}{b_y}$$

就说 x 和 y 是协调的（coordinated）。

显然，i 族中由强度削弱引入的所有新变量都是相互协调的，也都和 i 协调。

当一个归纳变量 j 的定值 j=…被替换为 j=j′时，就知道 j′是协调的，但是 j 可能不是协调的。不过，只要期间没有插入对 j′的定值，标准的复写传播算法就可以用 j′的使用替换 j 的使用。因此，可以不使用流分析去了解 j 是否协调，只要复写传播认为使用 j′是合法的就可以了。

在强度削弱后，程序中仍然有乘法，但乘法已经在循环之外了。如果循环执行多个迭代，则在许多机器上使用加法的循环应该比使用乘法的运行速度快得多。但是，在能够通过指令调度而隐藏乘法延迟的处理器上，强度削弱的效果有可能会令人失望。

对表 5-7 中左侧的程序执行强度削弱，发现 j 是三元组为(i,0,4)的一个导出归纳变量，k 的三元组为(i,a,4)。对 j 和 k 执行强度削弱后，有：

```
    s = 0
    i = 0
    j' = 0
    k' = a
L1: if i ⩾ n goto L2
    j = j'
    k = k'
    x = M[k]
    s = s + x
    i = i + 1
    j' = j' + 4
    k' = k' + 4
    goto L1
L2:
```

可以执行死代码删除来删除语句 j=j′，还希望删除无用变量 j′的所有定值，但是从技术上讲 j′并不是死的，循环的每个迭代中都使用它。

3. 删除

强度削弱后，一些归纳变量在循环中根本不被使用，另一些也只是用于和循环不变量做比较，这些归纳变量都可以删除。如果一个变量在循环 L 的所有出口都是死的，并且它只用于对自身的定值，那么在循环 L 中这个变量是无用的（useless），无用变量的所有定值都可以被删除。

在上述例子中，删除 j 后，变量 j′就成了无用变量，可以删除 $j'=j'+4$。如此一来，在前置节点中，j′的定值也能够通过死代码删除来移除。

4. 重写比较

如果变量 k 只是用于与循环不变量进行比较，或者只是用于自身的定值，并且在同一

族归纳变量中，还存在着另外某个不是无用的归纳变量，那么 k 就是一个几乎无用的变量。通过修改循环不变量与这个几乎无用的变量的比较，使之与相关的归纳变量进行比较，可以使一个几乎无用的变量变成 一个无用变量。

如果有一个比较 $k<n$，并且 j 和 k 是 i 族中的协调归纳变量，n 是循环不变量，则有

$$\frac{j-a_j}{b_j}=\frac{k-a_k}{b_k}$$

因此，比较 $k<n$ 可以写成：

$$a_k+\frac{b_k}{b_j}(j-a_j)<n$$

现在，可以将两边都减去 a_k，然后都乘以 b_j/b_k。如果 b_j/b_k 是正的，则这个比较变为：

$$j-a_j<\frac{b_j}{b_k}(n-a_k)$$

如果 b_j/b_k 是负的，这个比较则变为：

$$j-a_j>\frac{b_j}{b_k}(n-a_k)$$

最后，在两边都加上 a_j（这里只给出 b_j/b_k 为正的情况）：

$$j<\frac{b_j}{b_k}(n-a_k)+a_j$$

这个比较的整个右部是一个循环不变量，可以外提到循环前置节点中并只计算一次。

这里存在几条重要的限制：

（1）如果 $b_j(n-a_k)$ 不能被 b_k 整除，则不能使用上面的转换，因为不能在整数变量中保存一个小数值。

（2）如果 b_j 或者 b_k 不是常数，而是一个不能确定其正负值的循环不变量，则也不能使用上述转换，因为不知道应该使用哪种比较（小于还是大于）。

在前面的例子中，比较 $i<n$ 可以用 $k'<a+4*n$ 来替换。当然 $a+4*n$ 是一个循环不变量，应该外提。于是 i 将成为一个无用变量，并可以被删除。转换后的程序变为：

```
    s = 0
    k' = a
    b = n * 4
    c = a + b
L1: if k' < c goto L2
    k = k'
    x = M[k]
    s = s + x
    k' = k' + 4
    goto L1
L2:
```

最后，复写传播可以删除 $k=k'$，最终得到了表 5-7 中左侧的程序。

5.5.3 数组边界检查

安全的程序设计语言要求对每个数组下标操作进行数组边界检查，这会降低程序的执行性能。为了让安全的语言能够获得与不安全语言一样的执行效率，编译器可以尝试移除能够证明是冗余的边界检查。

静态移除所有可能冗余的边界检查是不可计算问题，但许多数组下标的访问都具有a[i]的语法形式，其中 i 是归纳变量，编译器可以对这种形式的下标进行有效的优化。

数组的边界通常具有形式 $0 \leqslant i \wedge i < N$，其中 N 是数组的大小，因此 N 总是非负的，可以将上述检查变形为 $i \leqslant_M N$，其中 $\leqslant_M$ 是一个无符号比较操作符。

1. 删除数组边界检查的条件

尽管直观地来看，一个归纳变量似乎一定会位于某个范围内，并且应该能知道这个范围是否超出了数组的边界，但实际上从循环 L 中删除一个边界检查的判别标准相当复杂。

(1) 有一个语句 s_1，它含有一个归纳变量 j 和一个循环不变量 u，且该语句具有下列形式之一：

```
if j < u goto L1 else goto L2
if j ⩾ u goto L1 else goto L1
if j >u goto L1 else goto L2
if j ⩽ u goto L1 else goto L1
```

其中，L_2 在循环之外。

(2) 有一个具有如下形式的语句 s_2：

```
if k <u n   goto L3   else goto   L4
```

其中，k 是和 j 协调的归纳变量，n 是循环不变量，s_1 是 s_2 的必经节点。

(3) L 中不存在包含 k 的定值的嵌套循环。

(4) 当 j 增加时，k 也增加，即 $b_j/b_k > 0$。

n 常常是数组长度。在有静态数组的语言中，数组长度 n 是常数。在许多具有动态数组的语言中，数组长度是循环不变量。在 Java 和 ML 中，数组一旦被分配，就不能动态地修改其长度。典型情况下，数组长度 n 可以通过读取某个数组指针 v 的 length 域来获得。为了便于阐述，假设 length 域位于数组对象中偏移为 0 的位置。为了避免进行复杂的别名分析，编译器的语义分析阶段需要将表达式 M[v] 标记为不变的，这意味着不会有其他存储指令能够修改数组 v 的 length 域的内容。如果 v 是循环不变量，则 n 也是循环不变量。即使 n 不是数组长度，而是其他某个循环不变量，也仍然能够优化比较 $k <_u n$。

要在循环前置节点中放一个测试，该测试要表达的意思是：每次迭代都有 $k \geqslant 0 \wedge k < n$。令 k_0 是前置节点末尾处 k 的值，令 $\Delta k_1, \Delta k_2, \cdots, \Delta k_m$ 是在循环内给 k 增加的所有的循环不变量值。于是，通过在前置节点的末尾进行如下测试来确保 $k \geqslant 0$：

$$k \geqslant 0 \wedge \Delta k_1 \geqslant 0 \wedge \cdots \wedge \Delta k_m \geqslant 0$$

令 $\Delta k_1, \Delta k_2, \cdots, \Delta k_p$ 是在 s_1 和 s_2 之间不再经过 s_1 的任意路径上给 k 增加的所有的循环不

变量值的集合。于是，只要保证在 s_1 处有

$$k < n - (\Delta k_1 + \Delta k_2 + \cdots + \Delta k_p)$$

就足以确保在 s_2 处有 $k<n$。由于知道 $(j-a_j)/b_j=(k-a_k)/b_k$，所以这个测试变为：

$$j < \frac{b_j}{b_k}(n - (\Delta k_1 + \Delta k_2 + \cdots + \Delta k_p) - a_k) + a_j$$

因为测试 $j<u$ 是测试 $k<n$ 的必经节点，所以当下面的测试成立时，上面这个测试将总是为真：

$$u < \frac{b_j}{b_k}(n - (\Delta k_1 + \Delta k_2 + \cdots + \Delta k_p) - a_k) + a_j$$

由于比较的两边都是循环不变量，所以可以按下面的方法将它移到前置节点中。首先，要确保所有循环不变量的定值都已提升到了循环之外。然后，按如下方法重写循环 L：复制 L 中的所有语句，由此创建一个头为 L'_h 的新循环 L'。在 L'中，将语句

```
if  k < n  goto  L'3  else  goto  L'4
```

替换为

```
goto L'3
```

在 L 的前置节点的末尾，加入与下列语句等价的语句：

```
if k≥ 0∧ k1≥ 0∧ … ∧ km≥ 0∧u < (bj/bk)(n-(Δk1 + Δk2 + … + Δkp) - ak) + aj
    goto L'h
else goto L'h
```

这个条件对 goto 语句测试 k 是否总在 0 和 n 之间。

有时已知的信息足以在编译时便计算出这一复杂的条件。至少在下面两种情况下可以做到这一点：

(1) 测试中出现的所有循环不变量都是常数；

(2) n 和 u 是同一个临时变量，$a_k=a_j$，$b_k=b_j$，并且在 s_1 和 s_2 之间没有给 k 增加 Δk。在类似 Java 这样的语言中，如果程序员编写的是如下程序，则可能会出现这种情况：

```
n = A.length
i = 0
while i < n do
    sum = sum + A[i]
    i = i + 1
```

假设数组 A 的长度是从相对此数组指针偏移为 0 的域中读取的，length(A)对应的四元式就将包括 n=M[A]。同时，A[i]的四元式为了用 n 进行边界检查，也会包括 n=M[A]。假设已标记表达式 M[A]为不变的，那么就不会有其他的存储指令修改存储单元 M[A]的内容，因此，现在对 u 和 n 定值的这两个表达式都是公共子表达式。

如果能够在编译时计算出前面那个复杂的比较，就能够无条件地使用循环 L 或循环

L′，并且删除没有使用的另一个循环。

2. 清理

优化后，程序中可能会遗留一些没有解决的小问题：标号 L′ 后面的语句可能是不可到达的；在循环 L′ 中可能有 n 和 k 的若干无用计算。前者可以通过不可到达代码删除来清理，后者可以通过死代码删除来清理。

3. 拓广

为了使这个算法在实际中有用，还需要从几个方面进行拓广。

(1) 循环出口比较可以是下列形式之一：

```
if  j ≤ u'  goto  L1   else goto  L2
if  j >u'   goto  L2   else goto  L1
if  u' ≥ j  goto  L1   else goto  L2
if  u' <j   goto  L2   else goto  L1
```

其中，比较 j≤u′替换了 j<u。

(2) 循环出口可以发生在循环体的底部，而不是数组边界检查之前。可以将这种情况描述如下：存在着一个测试

```
s2 : if  j ≤ u  goto  L1   else goto  L2
```

其中 L_2 在循环之外，并且 s_2 是所有循环回边的必经节点。则感兴趣的 Δk_i 位于 s_2 和任意回边之间，以及循环头和 s_1 之间。

(3) 应该处理 $b_j/b_k<0$ 的情况。

(4) 应该处理 j 的计数向下减少而不是向上增加的情况，此时循环出口测试类似 j≥l，l 是循环不变量的下界。

(5) 归纳变量的递增可能是“不规则的”，例如：

```
while i < n - 1 do
    if sum < 0
        then i = i + 1;sum = sum + i;i = i + 1;
    else i = i + 2;
    sum = sum + a[i];
```

这里，有 3 个 Δi(分别是 1、1 和 2)。这三个增加可能全都起作用，也可能只有某一个起作用，还可能全都不起作用。但显然在这里的两条路径上都有 i=i+2，这种情况下，将 i 的递增提到 if 之前(并合并它们)的分析对此会有益处。

5.5.4 循环展开

有些循环的循环体较小，循环大部分的执行时间用在了递增循环计数变量和测试循环退出条件上。编译器尝试展开这些循环，将循环体连续地复制两次或多次，可以使这些循环更加高效，这类优化称为循环展开(loop unrolling)。

给定一个头节点为 h，回边为 $s_i \to h$ 的循环 L，可以按以下方式展开：

(1) 复制节点，构建一个头节点为 h′，回边为 $s' \to h'$ 的循环 L′；

(2) 将循环 L 中所有从 $s_i \to h$ 的回边改为 $s_i \to h'$；

(3) 将循环 L′中所有从 $s' \to h'$ 的回边改为 $s'_i \to h$。

例如，表 5-9 中左侧的程序展开后得到表 5-9 中右侧的程序。但是这并没有完成什么有用的优化，每个“原始”迭代仍然有一个递增分支和一个条件分支。

表 5-9　无用的循环展开

循环展开前	循环展开后
L_1 : x = M[i] s = s + x' i = i + 4 if i < n goto L_1 else L_2 L_2 :	L_1 : x = M[i] s = s + x i = i + 4 if i < n goto L'_1 else L_2 L'_1 : x = M[i] s = s + x i = i + 4 if i < n goto L_1 else goto L_2 L_2 :

利用归纳变量的有关信息可以做得更好，需要一个归纳变量 i，i 的每次递增 $i=i+\Delta$ 是循环的每条回边的必经节点。于是每次迭代对 i 的递增恰好是所有 Δ 的和，因此可以将所有的递增和循环出口测试积累到一起，得到表 5-10 中左侧的程序。但是这样的循环展开仅在原始循环迭代次数为偶数时才正确。为解决这个问题，可以在展开的循环之后再增加一个结语(epilogue)来执行“奇数”迭代，如表 5-10 中右侧的程序所示，这样就可以在原始循环迭代次数任意的情况下工作。

这里仅给出了展开因子为 2 的展开情况。当使用展开因子 K 进行展开时，循环的结尾是一个 K−1 次迭代的循环(和原始循环类似)。

表 5-10　有用的循环展开

脆　弱　的	健　壮　的
L_1 : x = M[i] s = s + x x = M[i + 4] s = s + x i = i + 8 if i < n goto L_1 else L_2 L_2 :	if i < n − 8 goto L_1 else L_2 L_1 : x = M[i] s = s + x x = M[i + 4] s = s + x i = i + 8 if i < n − 8 goto L_1 else L_2 L_2 : x = M[i] s = s + x i = i + 4 if i < n goto L_2 else L_3 L_3 :

5.5.5 毕昇编译器的其他循环优化

一段时期以来，处理器速度的增速要远高于内存速度，如果程序中的所有访存操作以内存的速度执行，程序性能将受到极大的影响。人们为此设计了高速缓存用于缓解二者不匹配导致的性能问题。然而高速缓存的有效性极大地依赖于程序暴露出的数据局部性。

局部性原则是计算机体系结构设计过程中一个重要的观察和指导，即程序倾向于使用近期被使用过的数据和指令，局部性分为时间局部性和空间局部性。时间局部性指最近被访问过的内容近期内很可能被再次访问。空间局部性表示地址相近的内容往往在相近的时间内都被使用。当程序中的循环存在局部性时，晚一些的迭代可以复用在前面迭代中被加载到高速缓存中的数据，而不需要再去相对低速的内存中获取数据。为了充分发挥鲲鹏体系结构的优势，毕昇编译器实现了一系列特性的循环优化，下面将介绍几类重要的循环变换并从局部性的视角对这些变换展开分析。

1. 基础概念

循环的优化过程大致可以分为 3 个阶段：相关性/依赖分析、收益分析和循环变换，首先分别对这 3 个阶段涉及的一些重点概念进行介绍。

相关性/依赖分析是准确地识别出程序中存在的依赖关系(dependence)的过程。若依赖关系在变换前后被破坏，程序的正确性将无法得到保证。假设 Si 与 Sj 分别为程序中的两条访存指令，且 Si 在 Sj 之前执行。通过对两个访存操作可能的类型(写或读)进行排列组合，可以将依赖的类型分为以下 4 种。

(1) 流依赖(flow/true dependence)：Sj 读取 Si 输出的值，即 Read After Write (RAW)。

(2) 反依赖(anti dependence)：Si 重写 Sj 输出的值，即 Write After Read(WAR)。

(3) 输出依赖(output dependence)：Si 输出的值将被 Sj 重写，即 Write After Write (WAW)。

(4) 输入依赖(input dependence)：Sj 和 Si 读取同一个值，即 Read After Read(RAR)。

其中反依赖和输出依赖又被称为资源依赖，程序员可以通过消除代码中存储单元的复用来消除它们，典型的优化手段有寄存器重命名。输入依赖与其他三者不同，并不会阻碍 Si 与 Sj 的并行执行，因此通常所提的数据依赖并不将其包含在内。

从循环的角度入手，还可以进一步将依赖关系进行划分。

(1) 循环间依赖(loop-carried dependence)：当更晚的迭代中的数据访问依赖于更早的迭代中产生的数据时，称该依赖为循环间依赖。

(2) 循环无关依赖(loop-independent dependence)：与循环间迭代相对应，当依赖以静态的形式存在于迭代内部时，称该依赖为循环无关依赖。

如果循环中不存在循环间依赖，那么该循环的各个迭代可以并行地执行。

收益分析，即判断程序在经过一系列的变换后是否会在实际运行时获得性能等收益的过程，决定了一个变换最终是否应该被执行。本节将重点从数据局部性的角度分析一个变

换是否会带来收益，数据复用(data reuse)是局部性分析中的重要概念，当同一个数据在循环的多个迭代中被使用时，称该数据被复用(reused)。

从上述定义可知，当程序存在数据依赖时，与之对应的数据显然被复用了。但反之并不成立，因为数据依赖必须伴有写操作的发生，而复用则没有该要求。

复用是循环本身的内在属性，其存在并不意味着程序在真实的执行过程中体现了局部性。例如，当对相同数据的两次使用之间跨越了过多的迭代时，由于实际机器的高速缓存大小是有限的，在更早的迭代中被加载到高速缓存中的数据，在第二次使用时可能已经被其他数据换出，因此晚些的使用仍需要重新从更低层次的存储结构中加载数据。本节也将介绍一些能够将复用机会转换为程序局部性的循环变换。

数据复用还可以进一步划分如下。

(1) 自时间复用(self-temporal reuse)：当循环内的一个数据引用在不同的迭代中访问了相同的数据，称该引用存在自时间复用。

(2) 自空间复用(self-spatial reuse)：当循环中的一个数据引用在不同的迭代中访问了相同高速缓存行上的数据，称该引用存在自空间复用。

(3) 集合时间复用(group-temporal reuse)：当循环中的一些数据引用在不同迭代中访问了相同的数据，称该引用集合存在集合时间复用。

(4) 集合空间复用(group-spatial reuse)：当循环中的一些数据引用在不同迭代中访问了相同高速缓存行上的数据，称该引用集合存在集合空间复用。

可以看出，时间复用是空间复用的子集，因为相同的数据必然位于同一条高速缓存行上。下面以表 5-11 中的简单循环为例帮助读者理解各种类型的复用。

表 5-11　示例代码

包含多种类型复用的双重循环

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        f (A[i], A[j]);
    }
}
```

A[i]在最内层循环中存在自时间复用，因为在最内层循环(i 不变，j 从 0 到 n－1)的迭代中，A[i]始终访问相同的数组元素；同理 A[j]在最外层循环中(j 不变，i 从 0 到 n－1)存在自时间复用。此外，A[j]在最内层循环中还存在自空间复用，且一共被复用了 n 次；同理 A[i]在最外层循环中也存在自空间复用。循环中的复用可以分为 3 类：跨嵌套复用(across loop nests reuse)，嵌套内复用(loop nest reuse)以及最内层嵌套复用(innermost loop reuse)。

在识别到循环迭代空间中存在不同的复用机会后，可以采用恰当的变换改变循环对数据的处理顺序，从而利用局部性来减少对内存的访问次数，这样可以提高性能。下面是一些与循环变换相关的概念。

(1) 完美嵌套循环(perfectly-nested loop)：所有操作指令都在最内层循环中的嵌套循环。

(2) 距离向量(distance vector)：k 层嵌套循环的距离向量是一个 k 维整数向量 $\vec{d}=<d1,d2,\cdots,dk>$，对每一个索引向量 i，索引向量 $\vec{i}+\vec{d}=(i1+d1,\cdots,ik+dk)$ 对应的迭代依赖于索引向量 $\vec{i}$ 对应的迭代。

(3) 字典序为正(lexicographically positive)：如果一个距离向量至少有一个非零元素，且第一个非零元素为正，那么称该距离向量的字典序为正。

(4) 幺模矩阵(unimodular matrix)：行列式为 -1 或 1 的整型方阵。

(5) 幺模变换(unimodular transformation)：作用于嵌套循环的迭代空间，可用幺模矩阵表示的线性变换。

由于并不是所有嵌套循环的依赖关系都能用有限的距离向量集合表示，为简化起见，本节把讨论限制在循环依赖关系都能用距离向量表示的场景。

引入幺模变换的重要意义在于，幺模变换的效果可以形式化地用幺模矩阵和距离向量的乘积表示。当幺模变换作用于嵌套循环后，若产生的新循环能够按新的迭代次序正确执行，则称该幺模变换是合法(legal)的。可以证明，对于一个可用词典序为正的距离向量表示其依赖关系的嵌套循环，一个幺模变换是合法的当且仅当其距离向量经过该幺模变换后生成的向量仍是正词典序的。

因为幺模矩阵的乘积同样是幺模矩阵，所以一系列的幺模变换可以用一系列幺模矩阵相乘最终得到的矩阵来表示，这样可以快速判断一个或一系列幺模变换的正确性，并且一系列变换也可以用一次变换来完成，大大地简化了合法性检查和变换所需的工作。循环优化问题也被抽象表示为在一系列限制条件下，找到能最大化目标函数的幺模矩阵的过程。

幺模变换也有其局限性，其中最重要的是其只能对完美嵌套循环进行变换。多面体变换(polyhedral transformation)是一个针对非完美嵌套循环的研究方向，其能够很好地处理非完美嵌套循环，目前开源 LLVM 项目中也包含了多面体优化框架 Polly。由于大部分程序的循环为完美嵌套循环，或者可以通过循环外提等其他编译器变换转换为完美嵌套循环。

2. 循环反转

循环反转(loop reversal)是一种逆转循环迭代的执行顺序的幺模变换，其幺模矩阵是将一个单位矩阵部分对角线上的 1 用 -1 替换后得到的矩阵。如 $\begin{bmatrix}-1 & 0\\ 0 & 1\end{bmatrix}$ 对应表 5-12 中将一个双层嵌套循环的外层循环迭代方向逆转的操作。

原循环的距离向量为[1,0]，将幺模矩阵与距离向量相乘得到新的迭代空间内的距离向量，$\begin{bmatrix}-1 & 0\\ 0 & 1\end{bmatrix}\begin{bmatrix}1\\ 0\end{bmatrix}=\begin{bmatrix}-1\\ 0\end{bmatrix}$，新的距离向量字典序不为正，可知该变换是不合法的，读者也可以通过简单的程序分析确认。

表 5-12　循环反转优化

优　化　前	优　化　后

for (i = 0; i < n; i++) {
　for (j = 0; j < n; j++) {
　　a[i, j] = a[i - 1, j];
　}
}
	for (i = n - 1; i >= 0; i--) {
	　for (j = 0; j < n; j++) {
	　　a[i, j] = a[i - 1, j];
	　}
	}

循环反转并不能直接增强程序的局部性，更多的情况下是作为其他循环变换（如循环交换）的使能者（enabler）。

3. 循环交换

循环交换（loop interchange）是将两个相邻循环的嵌套顺序互换的幺模变换，在部分书籍或编译器实现中，对循环交换概念做了进一步的扩展，允许两个以上的非相邻循环进行交换，后者有时也称为循环置换（loop permutation），本书对二者不做区分。

继续利用幺模变换矩阵以及距离向量进行合法性推导。表 5-13 的例子中，变换前距离向量为 $\begin{bmatrix}1\\0\end{bmatrix}$，该循环交换对应的幺模矩阵为 $\begin{bmatrix}0 & 1\\1 & 0\end{bmatrix}$，变换后距离向量为 $\begin{bmatrix}0 & 1\\1 & 0\end{bmatrix}\begin{bmatrix}1\\0\end{bmatrix}=\begin{bmatrix}0\\1\end{bmatrix}$，仍为正字典序，因此此变换合法。

表 5-13　循环交换优化

优　化　前	优　化　后

for (i = 0; i < n; i++) {
　for (j = 0; j < n; j++) {
　　a[j, i] = a[j - 1, i + 1];
　}
}
	for (j = 0; j < n; j++) {
	　for (i = 0; i < n; i++) {
	　　a[j, i] = a[j - 1, i + 1];
	　}
	}

循环交换的意义在于，可以通过减少数据复用的间隔（Stride），提高程序的局部性。

4. 循环倾斜

循环倾斜（Loop Skewing）是一种改变循环迭代空间形状的幺模变换。表 5-14 中是一个典型的循环倾斜优化用例，嵌套对应的距离向量集合为 $\left\{\begin{bmatrix}1\\0\end{bmatrix},\begin{bmatrix}0\\1\end{bmatrix}\right\}$。

表 5-14　循环倾斜优化

优　化　前	优　化　后

for (i = 0; i < n; i++) {
　for (j = 0; j < m; j++) {
　　a[j, i] = a[j + 1, i - 1];
　}
}
	for (i = 0; i < n; i++) {
	　for (j = i; j < i + m; j++) {
	　　a[j - i, i] = a[j - i - 1, i - 1];
	　}
	}

变换后，循环的迭代空间发生了如图 5-20 所示的变化。

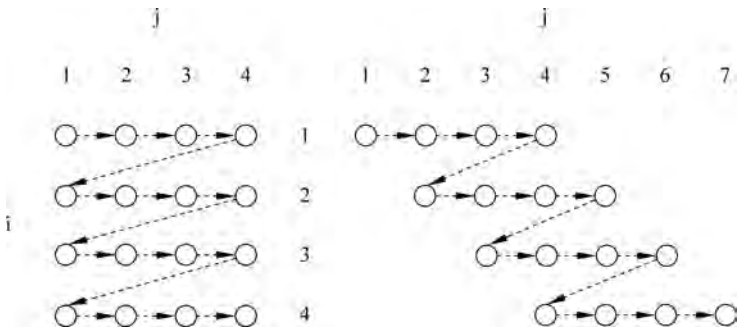

图 5-20 循环倾斜后的迭代空间变化

变化对应的幺模矩阵为 $\begin{bmatrix}1 & 0\\1 & 1\end{bmatrix}$，通过与原距离向量集合相乘，可以得到新的距离向量集合为 $\left\{\begin{bmatrix}1\\1\end{bmatrix},\begin{bmatrix}0\\1\end{bmatrix}\right\}$，可知该变换是合法的。尽管变换后内外层循环仍存在依赖，但此时再进行循环交换，就可以将循环进一步转换，如表 5-15 所示。

表 5-15 在循环倾斜变换后再进行循环交换

优 化 前	优 化 后

`for (i = 0; i < n; i++) {`
`  for (j = i; j < i + m; j++) {`
`  a[j - i, i] = a[j - i - 1, i - 1];`
`  }`
`}`
	`for (j = 0; j < m + n - 1; j++) {`
	`  for (i = max(0, j - m + 1); i < min(n,j); i++) {`
	`  a[j - i, i] = a[j - i - 1, i - 1];`
	`  }`
	`}`

这时最内层循环将可以并行。由此可见，循环倾斜与循环反转类似，其自身无法增强程序局部性从而带来性能收益，因此多被用于把嵌套循环转换为更利于其他变换发挥的结构。可以证明，循环倾斜变换始终是合法的。

5. 条带挖掘

条带挖掘(strip mining)是将一个 n 层嵌套循环的迭代空间重新划分，转换为嵌套层次在 n+1～2n 范围内的新循环的非幺模变换，如表 5-16 所示。

原一维的迭代空间被分割成了新的二维迭代空间，就像是被分割成了 floor(M/32) 条迭代次数为 32 的“迭代条带”，产生的外层循环称为控制循环(controlling loops)。由于条带挖掘只对迭代空间进行了切割，并不改变迭代的实际执行顺序，故可以证明其始终是合法的。仅靠条带挖掘并不能提高数据局部性，其往往结合循环交换一起出现，其收益将会在循环分块中做更具体的介绍。值得一提是，条带挖掘对于支持向量操作的机器而言是十分重要的一类变换。

表 5-16　条带挖掘优化

优　化　前	优　化　后

for (i = 0; i < n; i++) {
　A[i] = B[i] + 1;
　D[i] = B[i] − 1;
}
	for (j = 0; j < n; j += 32) {
	　for (i = j; i < min(M, j + 31); i++) {
	　　A[i] = B[i] + 1;
	　　D[i] = B[i] − 1;
	　}
	}

6. 循环分块

循环分块(loop tiling)是将条带挖掘和循环交换相结合的非幺模变换，如表 5-17 所示。

表 5-17　循环分块优化

优　化　前	条带挖掘变换后	循环交换优化后

for (i = 0; i < n; i++)
　for (i = 0; i < n; i++)
　　C[i] = A[j, i];
	for (ii = 0; ii < n; ii += 2)
	　for (jj = 0; jj < n; jj += 2)
	　　for (i = ii; i < min(ii + 2, n); i++)
	　　for (j = jj; jj < min(jj + 2, n); j++)
	　　C[i] = A[j, i];
		for (ii = 0; ii < n; ii += 2)
		　for (jj = 0; jj < n; jj += 2)
		　　for (j = jj; j < min(jj + 2, n); i++)
		　　for (i = ii; i < min(ii + 2, n); i++)
		　　C[i] = A[j, i];

表 5-17 中的循环分块首先将原嵌套循环的内外层均做了条带宽度为 2 的条带挖掘变换，由外到内的前两层循环为条带变换新增的控制循环。新循环的迭代执行顺序如图 5-21 所示，像是被划分成了大小为 2×2 的方块。

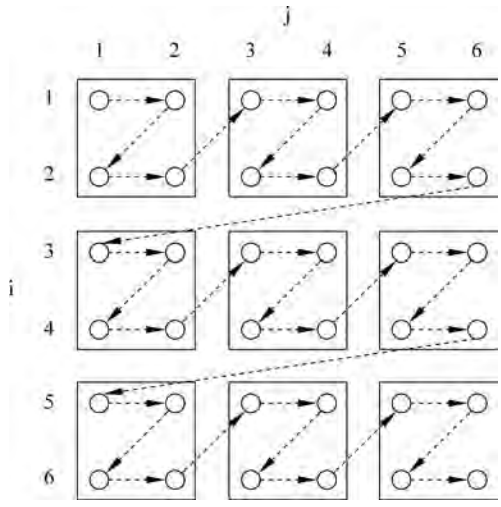

图 5-21　表 5-17 中条带挖掘变换后的循环对应的迭代空间

之后最内两层循环通过循环交换，得到表 5-17 中右侧的循环，假设数组 A 的元素在内存中按行优先(row-major)排布，上述变换能够有效减少数据复用之间跨越的迭代间隔，提高了程序局部性。合适的分块尺寸(tile size)需要结合机器的寄存器、高速缓存大小来进行选择。由前文可知，条带挖掘始终是合法的，因此循环分块的合法性检查与循环交换一致，在此略过。

7. 循环合并

循环合并(loop fusion)是将两个相邻且具有相同迭代空间的循环体，合并为一个循环体的非幺模变换，如表 5-18 所示。

表 5-18 循环合并优化

优 化 前	优 化 后

`for (i = 0; i < n; i++)`
`  A[i] = 0;`
`for (i = 0; i < n; i++)`
`  B[i] = 0;`
	`for (i = 0; i < n; i++) {`
	`  A[i] = 0;`
	`  B[i] = 0;`
	`}`

只有当待合并的两个循环拥有相同的迭代界限(iteration bounds)，同时原来的两个循环中的指令在合并后的新循环中不存在任何形式的数据依赖时，循环合并才是合法的。其合法性检查可以抽象为：合并不能引入任何新的字典序后向数据依赖(lexically backward data dependence)。

循环合并可以带来的收益包括：

(1) 提高数据局部性。如果合并前的循环存在数据复用，合并后数据的局部性将提高。

(2) 减少循环控制流开销。两个相邻循环合并后整体迭代次数减半，控制循环执行的整体开销(overhead)也将减半。

(3) 提高计算颗粒度。循环合并将细粒度的运算合并为更大粒度的代码整体，代码间边界的消失往往会暴露出更多的优化机会，如常量传播、死代码删除等。

但在某些场景下，循环合并也可能带来性能损失，如合并后的循环中寄存器压力过大产生溢出。

8. 循环展开合并

循环展开合并(loop unroll-and-jam)是在外层循环展开的基础上，对内层循环进行合并的非幺模变换。可以将其理解为循环展开和循环合并的结合形式，并且它的合法性检查也是后两类检查的结合，在此不做重复阐述。该变换首先通过循环展开，创造出相邻循环的嵌套结构，从而暴露出循环合并机会以试图获取循环合并的一系列收益。变换后的循环往往有更好的数据局部性，更密集的数据流也给像硬件预取一类的优化提供了更大的发挥空间，如表 5-19 所示。

表 5-19　循环展开合并优化

优　化　前	优　化　后

for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
  a[i, j] = a[i, j - 2];
	for (i = 0; i < n; i += 3)
	  for (j = 0; j < n; j++) {
	  a[i, j] = a[i, j - 2];
	  a[i + 1, j] = a[i + 1, j - 2];
	  a[i + 2, j] = a[i + 2, j - 2];
	  }

从表 5-19 中例子可见，循环展开合并还可以看作是对嵌套循环的最内层做的循环分块。

9. 循环拆分

循环拆分(loop distribution/fission)是将一个循环拆分为多个循环的非幺模变换，可以理解为循环合并的逆过程。拆分后的循环与原循环拥有相同的迭代空间。循环拆分的合法性检查可以抽象为：被拆分的循环不能包含任何字典序后向的循环间数据依赖。

表 5-20 中所表示的循环拆分过程，是将一个非完美嵌套循环拆分为完美嵌套循环和由剩余部分构成的循环。新产生的完美嵌套循环可以应用一系列幺模变换如循环交换，来提高程序的局部性获得性能收益。此外，循环拆分还可以应用于将循环拆分为包含循环间依赖的循环和不包含循环间依赖的循环，后者可以并行执行，若在支持向量指令的机器上，通过自动向量化优化可以获得性能收益。

表 5-20　循环拆分优化

优　化　前	优　化　后

for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++) {
  aa[j][i] = aa[j][i] + bb[j][i] * cc[j]
[i];
  }
  a[i] = b[i] + c[i] * d[i];
}
	for (i = 0; i < n; i++)
	  for (j = 0; j < n; j++)
	  aa[j][i] = aa[j][i] + bb[j][i] * cc[j]
	[i];
	for (i = 0; i < n; i++)
	  a[i] = b[i] + c[i] * d[i];

10. 循环剥离

循环剥离(loop peeling)是将循环的头部或尾部的几次迭代从原循环剥离，剥离出的部分与剩余循环独立执行，是一类非幺模变换。绝大部分 SIMD 处理器对 SIMD 读写操作的数据地址有着对齐上的要求，非对齐访问的效率要远低于对齐访问。以表 5-21 中的代码为例，假设数组 a[0] 元素对应的地址满足对齐要求，执行机器的每一条 SIMD 读指令可以同时读取两个数组元素大小宽度的内存上的内容。原循环的访存模式为 a[1]，a[3]，a[5]，…，可以看到每一次访存操作均为非对齐访问。

表 5-21 循环剥离优化

优 化 前	优 化 后

for (i = 1; i < n; i++)
  a[i] = i;
	a[1] = 1;
	for (i = 2; i < n; i++)
	  a[i] = i;

但将首次迭代从原循环中剥离出来后，访存模式将变为 a[2]，a[4]，a[6]，…的对齐访问形式，循环实际执行效率将显著提高。

11. 循环断开

循环断开(loop unswitching)是当循环中存在条件分支且分支的控制条件是循环不变量时，在分支位置将循环断开，生成两份不包含原分支的非幺模变换。考虑表 5-22 中的程序，若编译器判断 b[j] 为循环不变量，可以把条件判断语句从循环内提取到循环外，生成的新循环中将不再包含条件分支。该变换显而易见的好处有，分支判断的执行次数从原来的 n 次减少到了 1 次，减少了实际执行指令的开销；另外鲲鹏处理器是一款支持向量指令的处理器，减少循环中的分支也可以帮助编译器更好地进行自动向量化优化，提高程序的并行效率。但循环断开和循环展开、循环拆分等优化类似，大部分情况下会增加代码规模，因此在内存尺寸敏感的场景下需要谨慎使用。

表 5-22 循环断开优化

优 化 前	优 化 后

for (int i = 0; i < n; i++) {
  a[i] = 1;
  if (b[j] > 10)
  a[i] += 1;
}
	if (b[j] > 10)
	  for (int i = 0; i < n; i++)
	  a[i] += 1;
	else
	  for (int i = 0; i < n; i++)
	  a[i] = 1;

5.6 多级存储优化

2.3.1 节介绍了鲲鹏微架构的多级存储系统，高速缓存的访问速度比主存快得多，但是容量要小得多。为了使程序获得更好的性能，需要让尽可能多的数据访问发生在缓存中。而程序员在编写业务代码时主要针对业务逻辑抽象出对应的数据结构和算法，一般很少考虑程序底层的访存特征。因此需要编译器分析程序的访存特征，结合底层硬件的存储结构，对程序进行缓存和内存优化。常见的提升系统访存性能的优化有数据预取、数据重排。

5.6.1　数据预取

在程序运行的过程中，可以提前将所需的数据从主存读入高速缓存中，当 CPU 需要读取数据时，数据刚好已在离 CPU 最近的缓存中，这种优化技术称作数据预取（data prefetching）。

数据预取可以通过硬件、软件和软硬件协同来实现。软件预取依赖编译器分析程序的访存特征并插入和调度预取指令。预取指令的作用是告知 CPU 某些数据可能很快会被用到，期望 CPU 能将数据加载到缓存中。软件预取的优势是编译器可以利用编译时的信息，例如结构体、数组的内存布局等，从而获得较为准确的访存信息，但是软件预取指令本身会带来开销。硬件预取通过专用的硬件单元来监测程序的访存特征，自动预取数据。它的优势是没有指令开销，但是硬件预取受到如下因素的限制：硬件预取有启动延迟，需要收集一段访存模式后，才能启动预取数据；由于缺失了编译时信息，硬件预取可能不如软件预取那样准确；硬件预取所支持的访存模式比较有限，不支持不规则的内存访问。

鲲鹏微架构支持硬件和软件的协同预取，通过硬件预取支持基本的连续数据访问和规则步长数据访问，结合毕昇编译器的软件预取优化覆盖嵌套循环、间接访存等更复杂的访存场景，实现对大多数访存特征的预取支持。以下着重讨论软件预取。

有效的软件预取可以大幅度减小程序访存时延，预取的有效性需要满足下列几个条件。

（1）恰当的时间：预取指令应该能够把所需数据恰好在 CPU 所需要的时刻读入缓存。如果数据太早或太晚到达，则 CPU 仍需等待数据，并且会造成缓存污染。

（2）精准的数据：预取指令请求的数据应该是程序实际所需的数据，避免缓存污染。

（3）较少的指令开销：预取指令本身是有开销的，编译器应该尽量减少冗余预取指令带来的开销。

为了满足以上条件，数据预取需要识别瓶颈访存操作（delinquent load）。研究表明，极少数的访存操作往往造成了绝大部分的缓存缺失（cache miss），成为系统的性能瓶颈。因此瓶颈访存的识别对有效的缓存优化十分重要。识别瓶颈访存可以通过编译器的静态分析（如访存模式分析和缓存行为分析）和动态信息收集（如编译器插桩和性能分析工具）获得，或通过动静相结合的方法获得。

对于瓶颈访存，编译器可以根据成本模型计算预取的提前量。考虑表 5-23 中的循环。

表 5-23　软件预取优化——简单循环

软件预取之前	软件预取之后

`for (i = 0; i < N; i++) {`
`  sum += A[i] * B[i];`
`}`
	`for (i = 0; i < N; i++) {`
	`  prefetch(&a[i + k]);`
	`  prefetch(&b[i + k]);`
	`  sum += A[i] * B[i];`
	`}`

为了计算合适的提前量，编译器需要对两方面的信息进行建模，一个是系统的访存延迟(l)，即从一个预取指令发出，到所需的数据被取回到缓存中所需的时间。另一个是循环体运行一次的执行时间(s)。假设 l/s=k，则循环执行 k 次迭代，预取的数据刚好到达。如果编译器希望 A[i+k] 的数据在 i+k 次迭代开始的时候刚好读入缓存，那么需要在第 i 次迭代发出对 A[i+k] 预取指令。基于这个简单的模型，循环中数据预取的提前量(k)可以用 k=l/s 计算。

插入预取指令后的循环如表 5-23 右侧的程序所示。同样的模型可以扩展到外层循环的场景，当内存循环的指令数较少，且访存瓶颈在外层循环时，可以对外层循环进行预取，如表 5-24 所示。

表 5-24 软件预取优化——外层循环

软件预取之前	软件预取之后

for (i = 0; i < N; i++) {
  sum += A[i] * B[i];
  for (int j = 0; j < M; j++) {
  // code for inner loop
  }
}
	for (i = 0; i < N; i++) {
	  prefetch(&a[i + k]);
	  prefetch(&b[i + k]);
	  sum += A[i] * B[i];
	  for (int j = 0; j < M; j++) {
	  // code for inner loop
	  }
	}

对于间接访存场景，目标数据的内存地址由内存中另一个数据读取而来，研究和实验表明，此时采用表 5-25 中的预取模型往往取得较好效果。

表 5-25 软件预取优化——间接访存

软件预取之前	软件预取之后

for (i = 0; i < N; i++) {
  TargetArr[OffsetArr[i]]++;
}
	for (i = 0; i < N; i++) {
	  prefetch(&TargetArr[OffsetArr[[i + k]]);
	  prefetch(&OffsetArr[i + 2 * k]);
	  TargetArr[OffsetArr[i]]++;
	}

如果应用程序的访存模式复杂，例如包含复杂的图、树表等数据结构或访存行为呈现非规则形式，则传统的数据预取技术可能失效，此时可以考虑使用帮助线程(helper thread)来实现预取。帮助线程预取技术使用一个专门的预取线程提前于原有程序运行，这个预取线程会执行一个简化后的源程序，但足以覆盖源程序的热点区域和关键路径。在预取线程执行时，数据会被提前读入缓存中。

5.6.2 数据重组

缓存感知数据重组技术可以设计新的数据布局改善数据局部性，从而提升缓存的利用率。数据重组(data reorganization)可为其他编译优化提供机会，如自动向量化、使能硬件

数据预取、减少缓存冲突和错误共享等。数据重组由 3 个重要部分组成：数据重组合法性分析，盈利成本分析和数据布局变换。

（1）数据重组合法性分析涉及访存指针别名分析，数组的归类、数组维度和数据类型分析。编译器必须保证整个应用程序中的数据指针都能识别出所指向的数组。

（2）数据重组盈利分析根据访问数据单元之间亲和性、访问的频率等，确定如何做数据重组，并评估带来的性能收益。

（3）数据布局变换对数据布局进行变换，如数据结构体拆分、数据压缩、数据扁平化、数据填充等。

1. 结构体拆分

考虑如下的代码：

```
struct {
    double F0;
    double F1;
    double F2;
    double F3;
    double F4;
} s[N];      // N 是常量

void foo() {
    // loop1
    for (i = 0; i < N; i++) {
        … = s[i].F0 …;
        … = s[i].F3 …;
    }
    …
    // loop2
    for (i = 0; i < M; i++) {
        … = s[i].F2 …;
    }
    …
    // loop3
    for (i = 0; i < K; i++) {
        … = s[i].F1 …;
        … = s[i].F4 …;
    }
}
```

其中 s 是一个连续的结构体数组，其内存布局如图 5-22 所示。

图 5-22　结构体数组的内存布局

对 s 的访问呈现如下特征：loop1 中结构体单元 F0 和 F3 的访问同时发生，loop2 中 F2 的访问单独发生，loop3 中 F1 和 F4 的访问同时发生。硬件架构多级存储层次间是以高

速缓存行(cache line)大小为单位进行数据搬移的,鲲鹏的高速缓存行大小为 64 字节。Loop1 中每次仅访问 F0 和 F3,而 F1、F2 和 F4 并未使用,导致没有充分利用缓存,影响程序性能。为了解决这个问题,编译器可进行结构体拆分(data splitting),把常一起被访问的数据尽量放在一起,提高其空间局部性。

常一起被访问的数据具有访问亲和性(access affinity),例如 F0 和 F3,F1 和 F4。亲和性分析将结构体的字段按照访问亲和性进行聚类,得到亲和组(affinity group),然后编译器可以将每个亲和组内的字段重组为独立的数据结构,如图 5-23 所示。

图 5-23 重组后的内存布局

在拆分之后,F0 和 F3 的访问都落在一个高速缓存线内,数据有了更好的局部性,程序的访存性能得到了大幅提升。

对于动态链表,链表的每一个结构体节点可能分配在内存中不连续的空间,且可能存在节点的动态增加和删除,其内存布局如图 5-24 所示。

图 5-24 动态链表的内存布局

此时不能对结构体链表进行全局重排,但是仍然可以根据数据亲和性和冷热,在单个结构体节点内进行拆分。这样在分配单个节点时,可以保证冷热字段分开,避免在访问较热字段时将较冷字段读入高速缓存,提高缓存利用率,同时较热字段仍然按照亲和性排列,保证数据的空间局部性。重排后的内存布局如图 5-25 所示。

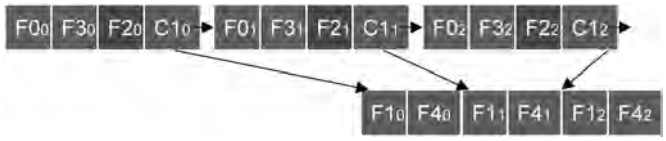

图 5-25 重排后的内存布局

2. 结构体压缩

结构体压缩是将较大类型的字段压缩为较小类型的字段。考虑表 5-26 中左侧的结构体数组。

其中字段 p 是一个指向结构体本身类型的指针,在 64 位机器上其大小为 8 字节,可以覆盖整个地址空间。假设编译器通过访存行为分析得到所有 p 值都落在结构体数组 arr 内,则 p 实际的取值仅仅落在有限的范围内,可以使用一个较小的数据类型来压缩字段 p。

压缩后的代码如表 5-26 中右侧的代码所示,原来的字段 p 通过外提的全局结构体指针 p_head 和压缩后的字段 p_offset 来访问。其中 p_head 为所有可能 p 值的最小值,即 arr[0] 的地址,p_offset 为相对 p_head 的偏移。此时对原有数组的字段 p 的访问 arr[i].p 可

以等价变换为 p_head＋arr[i]. p_offset。原有的结构体的指针成员被压缩为较小的数据类型，每个结构体的内存占用减小，从而提高了缓存利用率，提高了程序性能。

表 5-26　结构体压缩优化

结构体压缩前	结构体压缩后
struct S { 　　double F0; 　　S * p; 　　double F1; } arr[3];	struct S_compressed { 　　double F0; 　　unsigned int p_offset; 　　double F1; } arr[3]; S_compressed * p_head;

3. 数组扁平化

数组扁平化是指将动态分配在非连续空间的多维数组降维排布为连续的一维数组。扁平化可以提高数据的局部性，将间接内存访问简化为直接访问，提高程序的访存性能。考虑表 5-27 中左侧的代码。

表 5-27　数组扁平化优化

数组扁平化前	数组扁平化后

float ** A = (float **)malloc(N * sizeof(float *));
for (i = 0; i < N; i++)
　　A[i] = (float *) malloc (N * sizeof(float));
	float * A = (float *)malloc(N * N * sizeof(float));
	A[i * N + j] = … // 对应原来的 A[i][j]

这是一种常见的动态二维数组初始化的写法，其结果是数组 A 的第一维存储了一个一维数组的指针，每个一维数组指针指向一行连续的数据，但行与行之间的数据在内存中随机分布，如图 5-26 所示。

这样的内存布局会导致对 A 进行访问时局部性较差，跨行访问时尤其如此。同时访问 A 的一个元素需要进行两次访存，一次获得行首地址，一次获得实际数据，造成了额外的访存开销。对 A 进行扁平化后可以解决这两个问题，因为整个 A 数组都分配在一个连续的内存空间，其内存布局如图 5-27 所示。

图 5-26　扁平化前的内存分布

图 5-27　扁平化后的内存分布

相应地，编译器要对源码中的数组初始化和访问部分进行变换，原有的 A[i][j]可以等价变换为 A[i * N+j]，如表 5-27 中右侧的代码所示。

4. 数组重排

与结构体重排类似，数组的内存布局也可以重排，称为数组重排(Data Regrouping)。借鉴基于访问亲和性的分析，可以把常一起被访问的数组元素排列在一起。考虑表 5-28 中左侧的代码。

表 5-28 数组重排优化

数组重排前	数组重排后

for (int i = 200; i < 2000; i = i + 10) {
B[i] = A[i] + A[i - 200] + A[i + 200];
B[i+1] = A[i+1] + A[i-109] + A[i+251];
B[i+2] = A[i+2] + A[i-158] + A[i+232];
…
B[i+9] = A[i+9] + A[i-111] + A[i+509];
}
	#define N X/10
	for (int i = 20; i < 200; i = i + 1) {
	B[10 * i] = A[i] + A[i-20] + A[i+20];
	B[10 * i+1] = A[i+N] + A[i-11 + N] + A[i+25+N];
	B[10 * i+2] = A[i+2 * N] + A[i-16 + 2 * N] + A[i+23+2 * N];
	…
	B[10 * i+9] = A[i+9 * N] + A[i-12 + 9 * N] + A[i+50+9 * N];
	}

在这段代码中循环体的每一行需要访问三个 A 的元素，这三个访问看起来完全随机，空间局部性很差。但是经过分析可以发现，每行中三个访问的下标(设为 x)都满足 x mod 10 等于相同的数(设为 k)，即满足 x mod 10＝k 的元素具有更高的访问亲和性。在此前提下，编译器可以把数组的内容重新排列，让 A[x]，A[x+10]，A[x+20]等在内存中相邻排布。假设数组的大小为 X，则原始代码可以转换为表 5-28 中右侧的形式。

经过重排变换，循环中第一行的内存访问从 A[i]，A[i－200]，A[i+200]变为 A[i]，A[i－20]，A[i+20]，循环的迭代步长从 10 变为 1，数组访问的空间局部性得到了极大改善。假设系统的高速缓存线大小为 64 字节，A 的每个元素大小为 4 字节，则一个 A[i]的访问会将后续的 16 个元素都读入缓存，故重排之后 16 次迭代中 A[i]对应的数据都已经在缓存中。而在数组重排之前，由于每次循环步长为 10，两次迭代以后的数据就会产生缓存缺失。

5. 数组填充

数组填充(data padding)可以用于处理伪共享问题或提升基于 SIMD 的向量化效果。考虑如下例子：

```
int A[2];
int read() {
    return A[0];
}
void write(int x) {
```

```
    A[1] = x;
}
```

在这个例子中，数组 A 的两个元素位于同一个高速缓存线内，如果有两个线程同时执行 read 和 write 函数，由于 write 函数会改写 A[1] 的值，导致 read 函数每次执行时需要重新加载整个高速缓存线（即便 A[0] 的值没有变化），这造成了巨大的访存开销，这种现象称作伪共享。为了消除伪共享，可以在数组 A 中填充额外数据，将原有的 A[0] 和 A[1] 分隔于两个不同的高速缓存线。

数组填充还可以帮助获得更好的向量化效果。当数组的长度不是 SIMD 向量槽（vector lane）数量的整数倍时，向量化往往会产生需要单步执行的剩余循环（remainder loop），此时可以填充数组使得数组大小与向量槽数量对齐，从而移除剩余循环。

5.7　反馈式优化

传统的编译优化技术主要基于源代码的静态程序分析，在不实际执行程序的情况下，实现程序性能改进和优化。但在很多场景下，编译器往往会使用一些启发式的优化算法，对代码执行做出一些最大可能性的猜测。比如循环展开优化（loop unroll），编译器需要考虑循环执行的次数，循环内指令的数量；函数内联优化（inlining），编译器需要根据函数的代码体积大小决定是否对函数进行内联。同时为了使程序获得更好的性能，程序员必须具备一些编译器和与系统设计相关的知识，以便可以向编译器提供特定的提示（hints）或注释（annotations），但是这些优化手段并非总能奏效。本节将介绍一种现代编译器常用的、具有更好优化效果的性能优化技术——反馈式优化（Profile-Guided Optimizations，PGO）。

PGO 是一个动态的优化过程，编译器通过使用程序运行时生成的反馈数据（profile data），编译生成更优的代码。这些数据直接来自实际运行的程序，所以编译器可以利用这些数据进行更为精确的优化。现代编译技术已经可将反馈数据作用于编译过程的各个阶段，比如静态编译时、链接时、链接后/运行时等，如图 5-28 所示。

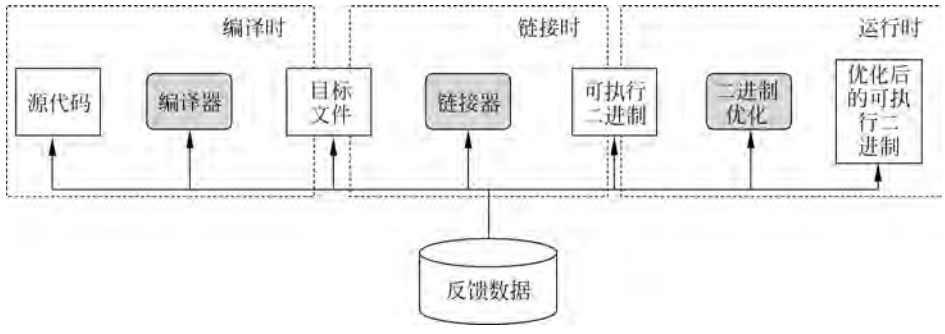

图 5-28　反馈数据作用示意图

典型的 PGO 优化分 3 个步骤完成。第一步是插桩编译。插桩编译会生成一个包含插桩信息的可执行程序，该程序在编译文件的每个基本块中都插入了探针(probes)。每个探针会统计当前基本块运行的次数。如果基本块产生了分支跳转，探针则会记录该分支跳转的方向。编译器一般会提供插桩编译选项，比如毕昇编译器插桩编译的命令如下：

```
clang++ -O2 -fprofile-instr-generate code.cc -o code
```

第二步是运行插桩可执行程序。在运行过程中，插桩程序会生成一个数据文件，包含了程序具体运行过程中的执行计数。

第三步是反馈优化。编译器读取来自第二步的反馈数据文件，从数据文件中解析出程序分析执行的详细信息，该信息用于更好地评估程序的控制流轨迹，从而指导编译器开展一系列的反馈优化，如基本块重排优化、函数内联等，并最终生成优化后的可执行文件。编译器一般会提供反馈优化选项，比如毕昇编译器反馈优化的命令如下：

```
clang++ -O2 -fprofile-instr-use=code.profdata code.cc -o code
```

反馈式优化会尽可能地对程序中最频繁运行的代码部分进行优化。如果每次程序运行过程中输入数据集基本一致，且执行路径及程序行为相近，PGO 的优化效果往往就是最好的。那反馈式优化提供了哪些具体的优化手段呢？

5.7.1 基本块重排优化

基本块重排优化(basic block reordering)会根据 PGO 的反馈信息，尝试找到代码中执行次数最频繁的路径，并将这些路径上的基本块在物理位置上排布得更为紧密。同时将一些在运行过程中未被执行到的代码块移动到代码段的最底部(远离热点代码块)。这会帮助优化指令缓存的局部性和提升分支预测概率。考虑图 5-29 的代码场景。

如果不考虑分支的实际执行情况，通过编译器静态分析优化，可以得到如图 5-30 所示的初始函数基本块及调用关系图。

```
funcD()
{
  do something
}
funcA()
{
  if (cond)
     call funcB
  else    // 热分支，根据profile结果
     call funcC
}
funcB()
{
  call funcD
}
funcC()
{
  call funcD
}
```

图 5-29 初始函数文件示意图

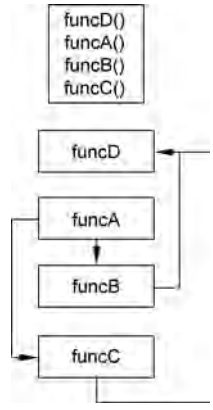

图 5-30 初始函数基本块及调用关系图

编译器从 PGO 反馈信息中识别到分支条件的执行概率(比如图 5-29 中 cond 值总是为 false,即 funcA→funcC→funcD 调用链为热分支),以此为基础进行基本块重排,将 funcA、funcC 和 funcD 排列在一起。完成重排优化后的基本块及函数调用关系图如图 5-31 所示。

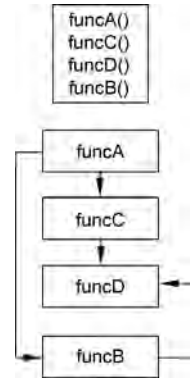

图 5-31　基本块重排后函数调用关系图

5.7.2　函数内联优化

函数内联优化(function inlining)是过程间优化最常使用的一种技术,主要的目的是将一些高频访问的函数内联到调用函数中,以减少函数调用开销。通常场景下,内联决策是一个非常复杂的过程,如 5.4.2 节的决策过程中所描述,编译器会在每个调用位置根据多条准则来决定是否做内联优化,比如根据代码块大小等做出决定。但遗憾的是,受限于编译器静态分析能力,并非所有的内联优化都具有正向的收益,有些反而可能导致性能下降。比如内联后导致函数体过大,寄存器溢出,或者在一些冷分支上做了大量非必要的内联操作,影响了指令访存。

```
funcA()
{
 if (!cond)
    do something  //funcC和funcD被内联
 else
    call funcB
}
funcB()
{
 do something
}
```

图 5-32　内联优化及基本块重排后函数示意图

如果有了反馈数据,编译器就可以获取函数在每次程序运行过程中的调用频次,从而可以做出更准确的内联决策。经过内联决策分析和基本块重排后,图 5-29 的函数调用关系会变成如图 5-32 所示(热点路径上的 funcC 及 funcD 调用会被内联)。

5.7.3　寄存器溢出策略

关于寄存器溢出的细节,可以参考 6.8.5 节。寄存器溢出策略(register spill placement)是一种非常精细的溢出策略,是指编译器(或者运行时系统)在程序运行过程中动态收集程序的动态运行概要信息(data profiling),并根据变量的访问信息(如访问次数)来决定溢出哪个变量。除此之外,还可以根据反馈信息中的冷热分支及基本块的信息,决定不同路径上寄存器的优先级(通常会将寄存器溢出在冷分支上,以减少对程序性能的影响)。

5.8　小结

编译器基于程序的中间表示对程序进行优化。程序优化主要分成两个步骤进行:程序分析和程序变换。在程序分析阶段,编译器分析并得到关于程序的运行时(保守)信息;程序优化依赖这些信息对程序进行变换,以改进程序的性能。本章讨论了程序分析和程序优化的主要技术,包括控制流分析、数据流分析及基于数据流的程序优化、别名分析、过程间

分析和优化、循环优化、内存优化等。

5.9 深入阅读

Lengauer 和 Tarjan 对计算必经节点的方法进行了研究；Tarjan 对其所使用的路径压缩算法进行了更完整的研究并给出了查找强连通分量的算法。

Knuth 对 FORTRAN 程序中的可归约性进行了研究。Allen 和 Cocke 对极大区间进行了定义。Aho、Sethi 和 Ullman 使用了该定义并给出了归约流图到极大区间的算法。

Sharir 最早将结构分析进行公式化。Rosen 给出了基于语法树的数据流分析方法。

Ershov 开发了值编号算法。Allen 和 Cocke 设计了第一个全局数据流分析算法。Kildall 最早提出了数据流分析的不动点迭代方法。Landi 和 Ryder 给出了一个过程间别名分析的算法。

Richardson 和 Ganapathi 对过程间优化效果进行了研究。Callahan 指出过程间优化可能更有助于并行编译器。

Weihl 证明了构造含过程变量的递归程序的调用图是 PSPACE-困难的。系统 ParaFrase 提及了其他一些相关问题。

Cooper 和 Kennedy 对计算流不敏感副作用的方法进行了论述，但 Cooper 等也指出了这种方法的明显的缺点，即它声称可以独立地解全局变量和形式参数子问题，同时 Cooper 解释了解形式参数问题必须先于非局部变量的处理。Myers 也对计算流敏感副作用的方法进行了介绍。

Callahan、Cooper、Kennedy 和 Torczon 论述了执行过程间常数传播的方法。Grove 和 Torczon 讨论了转移函数和返回转移函数、它们的计算代价以及所提供的信息。

Cooper 和 Kennedy 介绍了过程间别名分析的方法，Deutsch 介绍了一种更有效的方法。

5.10 习题

1. 为图 5-33 构造其深度优先表示、宽度优先表示、前序次序和后序次序。

2. 给出到达表达式的数据流方程。要特别注意被定值的临时变量同时也出现在四元式右边时的情况，例如四元式 t←t⊕b 或 t←M[t]。和到达定值一样，gen 和 kill 集合的元素可以是定值 ID。

3. 画出下面程序的基本块（基本块可能包含多条语句）的控制流图，给出每个基本块关于到达定值的 gen 和 kill 集合。

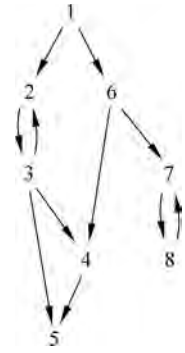

图 5-33 一个有根的有向图

```
1.      a = 5
2.      c = 1
3. L1 : if c > a goto L2
4.      c = c + c
5.      goto L1
6. L2 : a = c - a
7.      c = 0
```

4. 考虑下面程序的到达定值计算：

```
x = 1;
y = 1;
if z!= 0
    then x = 2;
else y = 2;
    w = x + y;
```

(1) 画出该程序的控制流图。

(2) 给出对该程序运行算法 21 得到的 sorted 数组。

(3) 计算到达定值，给出每次迭代的结果。总共需要多少次迭代？

5. 考虑以下代码片段。其中给出了一个过程 fee，还有两个调用了 fee 的调用位置。

```
static int A[1000,1000], B[1000];
        ...
x = A[i,j] + y;
call fee(i,j,1000);
        ...
call fee(1,1,0);
        ...
fee(int row; int col; int ub) {
    int i, sum;
    sum = A[row.col];
    for(i = 0; i < ub; i++) {
        sum = sum + B[i];
    }
}
```

(1) 将 fee 内联到各个调用位置，预期会有何种优化收益？请估算在内联和后续的优化之后，fee 中代码保留下来的比例。

(2) 概略描述一个较高层次的算法，用于估算内联某个具体调用位置带来的收益。应该考虑到调用位置和被调用者。

6. 对于过程置放算法，在处理边(p,q)时总是将 p 放置在 q 之前。

(1) 阐述该算法的一种变体，将边的目标置于源之前。

(2) 构造一个例子，使得这种算法在处理该例子时，能够将两个过程放置得比原来的算法更接近。假定所有过程的代码都具有同样的长度。

7. G 是一个控制流图，h 是 G 中的一个节点，A 是以 h 为头节点的一个循环的节点集

合，B 是以 h 为头节点的另一个循环的节点集合。证明其节点属于 A ∪ B 的子图也是一个循环。

8. 当流图中包含不可到达的节点时，直接必经节点定理将不再正确。

(1) 画出一个包含节点 d、e 和 n 的图，使得 d 是 n 的必经节点，e 是 n 的必经节点，但是 d 不是 e 的必经节点，e 也不是 d 的必经节点。

(2) 指出这个定理的证明中哪一步对包含不可到达节点的流图是无效的。

(3) 用 3 个左右的词，命名一个有助于寻找不可到达节点的算法。

9. 若为了某些目的，需要每个循环头节点恰好只有两个前驱，一个在循环外，一个在循环内。我们可以通过插入一个循环前置节点确保循环头只有一个循环外的前驱节点。解释如何插入一个循环体后置节点，以确保循环头只有一个循环内的前驱节点。

10. 假设任何算术溢出或除以零都会引发运行时的异常。如果我们将 t←a⊕b 提到循环外，而这个循环原来不会执行这条语句，那么在原始程序不产生异常的情况下，转换后的程序则有可能产生异常。修改循环不变量外提的标准，加入对上述问题的考虑。

11. 本章描述了将 while 循环转换为 repeat 循环的方法。说明如何(使用必经节点)刻画基本块控制流图中的 while 循环，以便优化器能够识别它。这种循环的循环体可能包含显式退出循环的 break 语句。