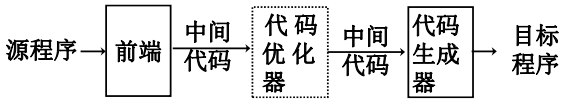
第7章　代码生成和代码优化



本章内容

- 一个简单的代码生成算法
  - 涉及存储管理、指令选择、寄存器分配和计算次序选择等基本问题
- 通过一个完整的例子来介绍代码优化

7.1 代码生成器设计中的问题

7.1.1 目标程序

- 绝对机器语言程序
  - 目标程序将装入到内存的固定地方
  - 粗略地说，相当于现在的可执行目标模块（第8章介绍）
- 可重定位目标模块
  - 代码中含重定位信息，以适应重定位要求

7.1 代码生成器设计中的问题

7.1.1 目标程序

- 可重定位目标模块

```
.L7:
        testl %eax,%eax
        je .L3
        testl %edx,%edx
        je .L7
        movl %edx,%eax
        jmp .L7
.L3:
        leave
        ret
```

可重定位目标模块中，需要有绿色部分的重定位信息

7.1 代码生成器设计中的问题

7.1.1 目标程序

- 绝对机器语言程序
  - 目标程序将装入到内存的固定地方
  - 粗略地说，相当于现在的可执行目标模块（第8章介绍）
- 可重定位目标模块
  - 代码中含重定位信息，以适应重定位要求
  - 允许程序模块分别编译
  - 允许从目标模块调用其他预先编译好的程序模块

7.1 代码生成器设计中的问题

7.1.1 目标程序

- 绝对机器语言程序
- 可重定位目标模块
  - 代码中含重定位信息，以适应重定位要求
  - 允许程序模块分别编译
  - 允许从目标模块调用其他预先编译好的程序模块
- 汇编语言程序
  - 免去编译器重复汇编器的工作
  - 增加可读性

7.1 代码生成器设计中的问题

7.1.2 指令选择

- 目标机器指令集的性质决定了指令选择的难易程度，指令集的统一性和完备性是重要的因素
- 指令的速度和机器特点是另一些重要的因素

7.1 代码生成器设计中的问题

- 若不考虑目标程序的效率，指令的选择是直截了当的

- 例 三地址语句x = y + z (x、y和z：静态分配)

```
MOV   y,   R0   /* 把y装入寄存器R0 */
ADD   z,   R0   /* z加到R0上 */
MOV   R0,  x    /* 把R0存入x中 */
```

逐个语句地生成代码，常常得到低质量的代码

7.1 代码生成器设计中的问题

语句序列 a = b + c
d = a + e
的代码如下

```
MOV   b,   R0
ADD   c,   R0
MOV   R0,  a
MOV   a,   R0
ADD   e,   R0
MOV   R0,  d
```

7.1 代码生成器设计中的问题

语句序列 a = b + c
d = a + e
的代码如下

```
MOV   b,   R0
ADD   c,   R0
MOV   R0,  a
MOV   a,   R0   -- 多余的指令
ADD   e,   R0
MOV   R0,  d
```

7.1 代码生成器设计中的问题

语句序列 a = b + c
d = a + e
的代码如下

```
MOV   b,   R0
ADD   c,   R0
MOV   R0,  a
MOV   a,   R0   -- 多余的指令
ADD   e,   R0   -- 若a不再使用，第三条指
MOV   R0,  d    --令也多余
```

7.1 代码生成器设计中的问题

7.1.3 寄存器分配

运算对象处于寄存器中的指令通常比运算对象处于内存的指令要短一些，执行也快一些

- 寄存器分配
  选择驻留在寄存器中的一组变量
- 寄存器指派
  为要驻留在寄存器的变量选择具体寄存器

7.1 代码生成器设计中的问题

7.1.4 计算次序选择

- 例

某种计算次序可能会比其他次序需要较少的寄存器来保存中间结果（见后面例题3）

7.2 目 标 语 言

7.2.1 目标机器的指令集

- 选择可作为几种微机代表的寄存器机器
- 4个字节组成1个字，有*n* 个通用寄存器R0, R1, …, R*n*-1。
- 二地址指令： *op* 源，目的

  MOV {源传到目的}

  ADD {源加到目的}

  SUB {目的减去源}

7.2 目 标 语 言

- 地址模式和它们的汇编语言形式及附加代价

模式	形式	地址	附加代价
绝对地址	M	M	1
寄存器	R	R	0
变址	*c*(R)	*c* + *contents*(R)	1
间接寄存器	*R	*contents*(R)	0
间接变址	**c*(R)	*contents*(*c* + *contents*(*R*))	1
立即数	#*c*	*c*	1

7.2 目 标 语 言

- 例 指令实例

  MOV R0, M

  MOV 4(R0), M

  4(R0)：*contents*(4 + *contents*(R0))

  MOV *4(R0), M

  *4(R0)：*contents*(*contents* (4 + *contents*(R0)))

  MOV #1, R0

7.2 目 标 语 言

7.2.2 指令代价

- 指令代价简化为

  1 + 指令的源和目的地址模式的附加代价

指令	代价
MOV R0，R1	1
MOV R5，M	2
ADD #1， R3	2
SUB 4(R0), *12(R1)	3

7.2 目 标 语 言

- 例 a = b + c， a、b和c都静态分配内存单元
  - 可生成

    MOV b, R0

    ADD c, R0 代价= 6

    MOV R0, a
  - 也可生成

    MOV b, a

    ADD c, a 代价= 6

7.2 目 标 语 言

- 例 a = b + c， a、b和c都静态分配内存单元
  - 若R0，R1和R2分别含a，b和c的地址，则可生成

    MOV *R1, *R0

    ADD *R2, *R0 代价= 2
  - 若R1和R2分别含b和c的值，并且b的值在这个赋值后不再需要，则可生成

    ADD R2, R1

    MOV R1, a 代价= 3

7.3 基本块和流图

- 怎样为三地址语句序列生成目标代码？

```
prod = 0;                          (1) prod = 0
i = 1;                             (2) i = 1
do {                               (3) t1 = 4 * i
                                   (4) t2 = a[t1]
    prod = prod + a[i] * b[i];     (5) t3 = 4 * i
    i = i +1;                      (6) t4 = b[t3]
} while (i <= 20);                 (7) t5 = t2 * t4
                                   (8) t6 = prod + t5
                                   (9) prod = t6
                                  (10) t7 = i +1
                                  (11) i = t7
                                  (12 ) if i <= 20 goto (3)
```

7.3 基本块和流图

7.3.1 基本块

- 基本块：连续的语句序列，控制流从它的开始进入，并从它的末尾离开，没有停止或分支的可能性（末尾除外）
- 流图：用有向边表示基本块之间的控制流信息，基本块作为结点，就能得到程序的流图

7.3 基本块和流图

- 把三地址语句序列划分成基本块

(1) 首先确定所有的入口语句

- 序列的第一个语句是入口语句
- 能由条件转移语句或无条件转移语句转到的语句是入口语句
- 紧跟在条件转移语句或无条件转移语句后面的语句是入口语句

(2) 每个入口语句到下一个入口语句之前（或到程序结束）的语句序列构成一个基本块

7.3 基本块和流图

```
(1)   prod = 0
(2)   i = 1
(3)   t1 = 4 * i
(4)   t2 = a[t1]
(5 )  t3 = 4 * i
(6 )  t4 = b[t3]
(7 )  t5 = t2 * t4
(8 )  t6 = prod + t5
(9 )  prod = t6
(10)  t7 = i +1
(11)  i = t7
(12 ) if i <= 20 goto (3)
```

```
(1)prod = 0
(2) i = 1
```

B_1

```
(3) t1 = 4 * i
(4) t2 = a[t1]
(5) t3 = 4 * i
(6) t4 = b[t3]
(7) t5 = t2 * t4
(8) t6 = prod + t5
(9) prod = t6
(10) t7 = i +1
(11) i = t7
(12) if i <= 20 goto (3)
```

B_2

7.3 基本块和流图

7.3.2 基本块的优化

- 术语：
  - 三地址语句x = y + z引用y和z并对x定值
  - 若一个名字的值在基本块的某一点以后还要被引用，则说这个名字在该点是活跃的
- 基本块的等价
  - 两个基本块的出口点有同样的活跃变量集合
  - 对其中每个活跃变量，代表其值的两个表达式相等
- 有很多等价变换可用于基本块
  - 局部变换
  - 全局变换

7.3 基本块和流图

- 删除局部公共子表达式

```
a = b + c          a = b + c
a = b + c          a = b + c
b = a – d          b = a – d
c = b + c          c = b + c
d = a – d          d = b
```

- 删除死代码

定值x = y + z以后不再引用x，则称x为死变量

7.3 基本块和流图

- 交换相邻的独立语句

  $t_1 = b + c$　　$t_2 = x + y$

  $t_2 = x + y$　　$t_1 = b + c$

当且仅当t_1和t_2不相同，x和y都不是t_1，并且b和c都不是t_2

- 代数变换

  $x = x + 0$　可以删除

  $x = x * 1$　可以删除

  $x = y ** 2$　改成$x = y * y$

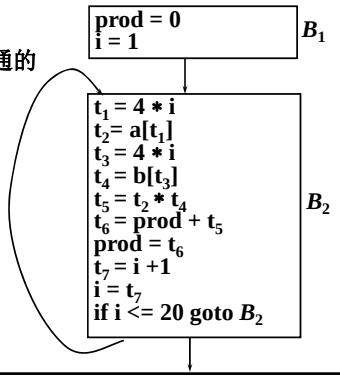
7.3 基本块和流图

7.3.3 流图

- 把控制流信息加到基本块集合，形成一个有向图来表示程序
- 流图中的节点
  初始结点、前驱、后继

7.3 基本块和流图

- 什么是循环?
  - 所有结点是强连通的
  - 唯一的循环入口
- 外循环和内循环



7.3 基本块和流图

7.3.4 下次引用信息

- 为每个三地址语句x = y op z决定x、y和z的下次引用信息

i: x = y op z
. . .
j: … = x …
. . .
k: … = … x

—没有对x的赋值

—没有对x的赋值

7.3 基本块和流图

- 对每个基本块从最后一个语句反向扫描到第一个语句，可以得到下次引用信息

i: x = y op z
. . .
j: … = x …
. . .
k: … = … x

—没有对x的赋值

—没有对x的赋值

- 利用下次引用信息，可以压缩临时变量需要的空间

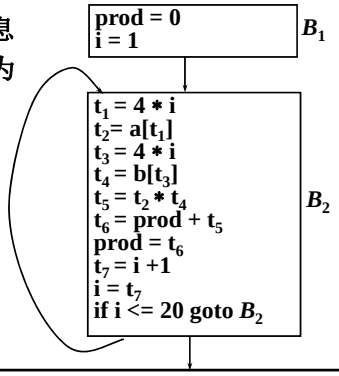
7.4 一个简单的代码生成器

基本考虑：

- 依次考虑基本块的每个语句，为其产生代码
- 假定三地址语句的每种运算都有对应的目标机器指令
- 假定计算结果尽量长时间地留在寄存器中，除非：
  - 该寄存器要用于其他计算，或者
  - 到基本块的出口点

7.4 一个简单的代码生成器

在没有收集全局信息前，暂且以基本块为单位来生成代码



7.4 一个简单的代码生成器

7.4.1 寄存器描述和地址描述

- 例 对a = b + c
  - 如果寄存器Ri含b，Rj含c，且b此后不再活跃
    产生ADD Rj, Ri，结果a在Ri中
  - 如果Ri含b，但c在内存单元，b仍然不再活跃
    产生　　ADD c, Ri，
    或者产生　MOV c, Rj
    　　ADD Rj, Ri
    若c的值以后还要用, 第二种代码较有吸引力

7.4 一个简单的代码生成器

- 在代码生成过程中，需要跟踪寄存器的内容和名字的地址
  - 寄存器描述记住每个寄存器当前存的是什么
    在任何一点，每个寄存器保存若干个(包括零个)名字的值

```
例
                // 语句前，R0保存变量a的值
b = a           // 不为该语句产生任何指令
                // 语句后，R0保存变量a和b的值
```

7.4 一个简单的代码生成器

- 在代码生成过程中，需要跟踪寄存器的内容和名字的地址
  - 寄存器描述记住每个寄存器当前存的是什么
    在任何一点，每个寄存器保存若干个(包括零个)名字的值
  - 名字的地址描述记住运行时每个名字的当前值可以在哪个场所找到
    这个场所可以是寄存器、栈单元、内存地址, 甚至是它们的某个集合
    例　产生MOV c, R0后，c的值可在R0和c的存储单元找到

7.4 一个简单的代码生成器

- 在代码生成过程中，需要跟踪寄存器的内容和名字的地址
  - 寄存器描述记住每个寄存器当前存的是什么
    在任何一点，每个寄存器保存若干个(包括零个)名字的值
  - 名字的地址描述记住运行时每个名字的当前值可以在哪个场所找到
    这个场所可以是寄存器、栈单元、内存地址, 甚至是它们的某个集合
  - 名字的地址信息存于符号表，另建寄存器描述表
  - 这两个描述在代码生成过程中是变化的

7.4 一个简单的代码生成器

7.4.2 代码生成算法

- 对每个三地址语句x = y *op* z
  - 调用函数*getReg*决定放y *op* z计算结果的场所L
  - 查看y的地址描述, 确定存放y的当前值的一个场所y′.如果y的值还不在L中，产生指令MOV y′，L
  - 产生指令*op* z′，L，其中z′是z的当前场所之一
  - 如果y和/或z的当前值不再引用，在块的出口点也不活跃，并且还在寄存器中，那么修改寄存器描述

7.4 一个简单的代码生成器

7.4.3 寄存器选择函数

- 函数***getReg***返回保存**x = y *op* z**的**x**值的场所**L**
  - 如果名字**y**在**R**中，这个**R**不含其他名字的值,并且在执行**x = y *op* z**后**y**不再有下次引用，那么返回这个**R**作为**L**
  - 否则，返回一个空闲寄存器，如果有的话
  - 否则，如果**x**在块中有下次引用，或者***op***是必须用寄存器的算符，那么找一个已被占用的寄存器**R**(可能产生**MOV R**，**M**指令，并修改 **M**的描述)
  - 否则，如果**x**在基本块中不再引用，或者找不到适当的被占用寄存器，选择**x**的内存单元作为**L**

7.4 一个简单的代码生成器

- 赋值语句**d = (a – b) + (a –c) + (a – c)**

编译产生三地址语句序列：

$t_1 = a - b$

$t_2 = a - c$

$t_3 = t_1 + t_2$

$d = t_3 + t_2$

7.4 一个简单的代码生成器

语 句	生成的代码	寄存器描述	名字地址描述
		寄存器空	
$t_1 = a - b$	**MOV a, R0** **SUB b, R0**	R0含t_1	t_1在R0中
$t_2 = a - c$	**MOV a, R1** **SUB c, R1**	R0含t_1 R1含t_2	t_1在R0中 t_2在R1中
$t_3 = t_1 + t_2$	**ADD R1,R0**	R0含t_3 R1含t_2	t_3在R0中 t_2在R1中
$d = t_3 + t_2$	**ADD R1,R0**	R0含d	d在R0中
	MOV R0, d		d在R0和内存中

7.4 一个简单的代码生成器

语 句	生成的代码	寄存器描述	名字地址描述
		寄存器空	
$t_1 = a - b$	**MOV a, R0** **SUB b, R0**	R0含t_1	t_1在R0中
$t_2 = a - c$	**MOV a, R1** **SUB c, R1**	R0含t_1 R1含t_2	t_1在R0中 t_2在R1中
$t_3 = t_1 + t_2$	**ADD R1,R0**	R0含t_3 R1含t_2	t_3在R0中 t_2在R1中
$d = t_3 + t_2$	**ADD R1,R0**	R0含d	d在R0中
	MOV R0, d		d在R0和内存中

7.4 一个简单的代码生成器

语 句	生成的代码	寄存器描述	名字地址描述
		寄存器空	
$t_1 = a - b$	**MOV a, R0** **SUB b, R0**	R0含t_1	t_1在R0中
$t_2 = a - c$	**MOV a, R1** **SUB c, R1**	R0含t_1 R1含t_2	t_1在R0中 t_2在R1中
$t_3 = t_1 + t_2$	**ADD R1,R0**	R0含t_3 R1含t_2	t_3在R0中 t_2在R1中
$d = t_3 + t_2$	**ADD R1,R0**	R0含d	d在R0中
	MOV R0, d		d在R0和内存中

7.4 一个简单的代码生成器

语 句	生成的代码	寄存器描述	名字地址描述
		寄存器空	
$t_1 = a - b$	**MOV a, R0** **SUB b, R0**	R0含t_1	t_1在R0中
$t_2 = a - c$	**MOV a, R1** **SUB c, R1**	R0含t_1 R1含t_2	t_1在R0中 t_2在R1中
$t_3 = t_1 + t_2$	**ADD R1,R0**	R0含t_3 R1含t_2	t_3在R0中 t_2在R1中
$d = t_3 + t_2$	**ADD R1,R0**	R0含d	d在R0中
	MOV R0, d		d在R0和内存中

7.4 一个简单的代码生成器

语 句	生成的代码	寄存器描述	名字地址描述
		寄存器空	
$t_1 = a - b$	MOV a, R0 SUB b, R0	R0含t_1	t_1在R0中
$t_2 = a - c$	MOV a, R1 SUB c, R1	R0含t_1 R1含t_2	t_1在R0中 t_2在R1中
$t_3 = t_1 + t_2$	ADD R1,R0	R0含t_3 R1含t_2	t_3在R0中 t_2在R1中
$d = t_3 + t_2$	ADD R1,R0	R0含d	d在R0中
	MOV R0, d		d在R0和内存中

7.4 一个简单的代码生成器

语 句	生成的代码	寄存器描述	名字地址描述
		寄存器空	
$t_1 = a - b$	MOV a, R0 SUB b, R0	R0含t_1	t_1在R0中
$t_2 = a - c$	MOV a, R1 SUB c, R1	R0含t_1 R1含t_2	t_1在R0中 t_2在R1中
$t_3 = t_1 + t_2$	ADD R1,R0	R0含t_3 R1含t_2	t_3在R0中 t_2在R1中
$d = t_3 + t_2$	ADD R1,R0	R0含d	d在R0中
	MOV R0, d		d在R0和内存中

7.4 一个简单的代码生成器

- 前三条指令可以修改，使执行代价降低

修改前	修改后
MOV a, R0	MOV a, R0
SUB b, R0	MOV R0，R1
MOV a, R1	SUB b, R0
SUB c, R1	SUB c, R1
. . .	. . .

7.4 一个简单的代码生成器

7.4.4 为变址和指针语句产生代码

- 变址与指针运算的三地址语句的处理和二元算符的处理相同

7.4 一个简单的代码生成器

7.4.5 条件语句

- 实现条件转移有两种方式
  - 根据寄存器的值是否为下面6种情况之一进行分支：负、零、正、非负、非零和非正
  - 用条件码来表示计算的结果或装入寄存器的值是负、零还是正

7.4 一个简单的代码生成器

1、根据寄存器的值是否为下面6种情况之一进行分支：负、零、正、非负、非零和非正

- 例 **if x < y goto L**
  - 把x减y的值存入寄存器R
  - 如果R的值为负，则跳到L

7.4 一个简单的代码生成器

2、用条件码的例子

• 例 若if x < y goto L 的实现是：

```
CMP     x,   y
CJ<     L
```

则：x = y + w
if x < 0 goto L
的实现是：

```
MOV     y,   R0
ADD     w,   R0
MOV     R0,  x
CJ<     L
```

7.5 代码优化概述

- 代码优化
  - 通过程序变换（局部变换和全局变换）来改进程序，称为优化
- 代码改进变换的标准
  - 代码变换必须保程序的含义
  - 采取安全稳妥的策略
  - 变换减少程序的运行时间平均达到一个可度量的值
  - 变换所作的努力是值得的
- 本节通过实例介绍独立于机器的优化

7.5 代码优化概述

7.5.1 优化的主要源头

- 程序中存在许多程序员无法避免的冗余运算
  - 对于A[i][j]和X.f1这样访问数组元素和结构体的域的操作
  - 随着程序被编译，这些访问操作展开成多步低级算术运算
  - 对同一个数据结构的多次访问带来许多公共的低级运算
  - 程序员没有办法删除这些冗余

7.5 代码优化概述

7.5.2 一个实例

```
i = m –1; j = n; v = a[n];
while (1) {
  do i = i +1; while(a[i]<v);
  do j =j –1;while (a[j]>v);
  if (i >= j) break;
  x=a[i]; a[i]=a[j]; a[j]=x;
}
x=a[i]; a[i]=a[n]; a[n]=x;
```

```
(1)  i = m –1
(2)  j = n
(3)  t1 = 4 * n
(4)  v = a[t1]
(5)  i = i + 1
(6)  t2 = 4 * i
(7)  t3 = a[t2]
(8)  if t3 < v goto (5)
```

7.5 代码优化概述

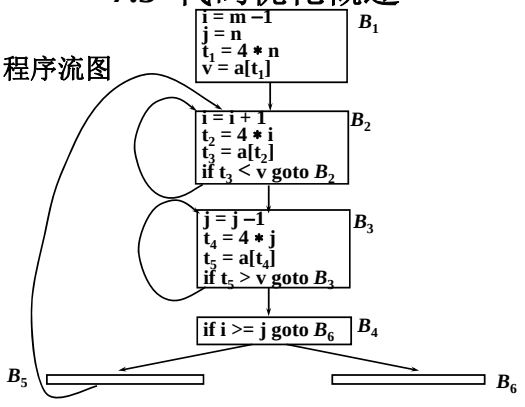
7.5.2 一个实例

```
i = m –1; j = n; v = a[n];
while (1) {
  do i = i +1; while(a[i]<v);
  do j =j –1;while (a[j]>v);
  if (i >= j) break;
  x=a[i]; a[i]=a[j]; a[j]=x;
}
x=a[i]; a[i]=a[n]; a[n]=x;
```

```
(9)   j = j –1
(10)  t4 = 4 * j
(11)  t5 = a[t4]
(12)  if t5>v goto (9)
(13)  if i >=j goto (23)
(14)  t6 = 4 * i
(15 ) x = a[t6]
. . .
```

7.5 代码优化概述

• 程序流图



7.5 代码优化概述

7.5.3 公共子表达式删除

B_5 x=a[i]; a[i]=a[j]; a[j]=x;

B_5

$t_6 = 4 * i$
$x = a[t_6]$
$t_7 = 4 * i$
$t_8 = 4 * j$
$t_9 = a[t_8]$
$a[t_7] = t_9$
$t_{10} = 4 * j$
$a[t_{10}] = x$
goto B_2

7.5 代码优化概述

局部公共子表达式删除、复写传播、 删除死代码

B_5 x=a[i]; a[i]=a[j]; a[j]=x;

B_5

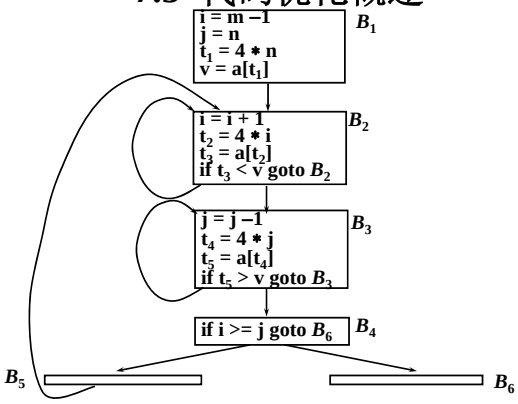
$t_6 = 4 * i$
$x = a[t_6]$
$t_7 = 4 * i$
$t_8 = 4 * j$
$t_9 = a[t_8]$
$a[t_7] = t_9$
$t_{10} = 4 * j$
$a[t_{10}] = x$
goto B_2

$t_6 = 4 * i$
$x = a[t_6]$

$t_8 = 4 * j$
$t_9 = a[t_8]$
$a[t_6] = t_9$

$a[t_8] = x$
goto B_2

7.5 代码优化概述



7.5 代码优化概述

全局公共子表达式删除、复写传播、 删除死代码

B_5 x=a[i]; a[i]=a[j]; a[j]=x;

B_5

$t_6 = 4 * i$
$x = a[t_6]$
$t_7 = 4 * i$
$t_8 = 4 * j$
$t_9 = a[t_8]$
$a[t_7] = t_9$
$t_{10} = 4 * j$
$a[t_{10}] = x$
goto B_2

$t_6 = 4 * i$
$x = a[t_6]$
$t_8 = 4 * j$
$t_9 = a[t_8]$
$a[t_6] = t_9$
$a[t_8] = x$
goto B_2

7.5 代码优化概述

全局公共子表达式删除、复写传播、 删除死代码

B_5 x=a[i]; a[i]=a[j]; a[j]=x;

B_5

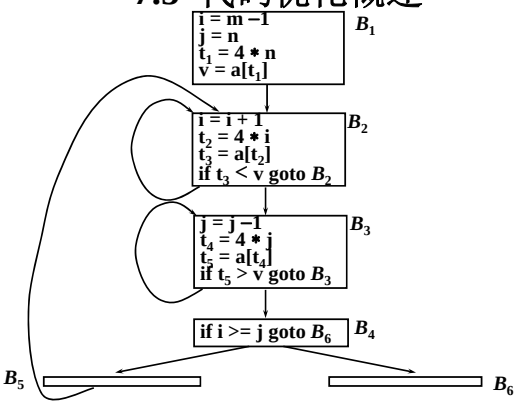
$t_6 = 4 * i$
$x = a[t_6]$
$t_7 = 4 * i$
$t_8 = 4 * j$
$t_9 = a[t_8]$
$a[t_7] = t_9$
$t_{10} = 4 * j$
$a[t_{10}] = x$
goto B_2

$t_6 = 4 * i$
$x = a[t_6]$
$t_8 = 4 * j$
$t_9 = a[t_8]$
$a[t_6] = t_9$
$a[t_8] = x$
goto B_2

$x = a[t_2]$

$t_9 = a[t_4]$
$a[t_2] = t_9$
$a[t_4] = x$
goto B_2

7.5 代码优化概述



7.5 代码优化概述

全局公共子表达式删除、复写传播、 删除死代码

B_5 x=a[i]; a[i]=a[j]; a[j]=x;

B_5

$t_6 = 4 * i$
$x = a[t_6]$
$t_7 = 4 * i$
$t_8 = 4 * j$
$t_9 = a[t_8]$
$a[t_7] = t_9$
$t_{10} = 4 * j$
$a[t_{10}] = x$
goto B_2

$t_6 = 4 * i$
$x = a[t_6]$
$t_8 = 4 * j$
$t_9 = a[t_8]$
$a[t_6] = t_9$
$a[t_8] = x$
goto B_2

$x = a[t_2]$
$t_9 = a[t_4]$
$a[t_2] = t_9$
$a[t_4] = x$
goto B_2

7.5 代码优化概述

全局公共子表达式删除、复写传播、 删除死代码

B_5 x=a[i]; a[i]=a[j]; a[j]=x;

B_5

$t_6 = 4 * i$
$x = a[t_6]$
$t_7 = 4 * i$
$t_8 = 4 * j$
$t_9 = a[t_8]$
$a[t_7] = t_9$
$t_{10} = 4 * j$
$a[t_{10}] = x$
goto B_2

$t_6 = 4 * i$
$x = a[t_6]$
$t_8 = 4 * j$
$t_9 = a[t_8]$
$a[t_6] = t_9$
$a[t_8] = x$
goto B_2

$x = a[t_2]$
$t_9 = a[t_4]$
$a[t_2] = t_9$
$a[t_4] = x$
goto B_2

$x = t_3$
$a[t_2] = t_5$
$a[t_4] = x$
goto B_2

7.5 代码优化概述

公共子表达式删除、复写传播、删除死代码

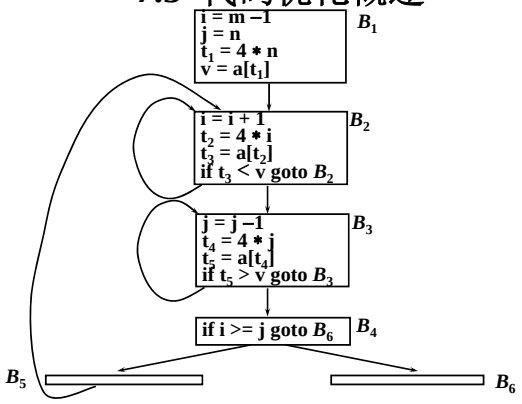
B_6 x = a[i]; a[i] = a[n]; a[n] = x;

B_6

$t_{11} = 4 * i$
$x = a[t_{11}]$
$t_{12} = 4 * i$
$t_{13} = 4 * n$
$t_{14} = a[t_{13}]$
$a[t_{12}] = t_{14}$
$t_{15} = 4 * n$
$a[t_{15}] = x$

$x = t_3$
$t_{14} = a[t_1]$
$a[t_2] = t_{14}$
$a[t_1] = x$

7.5 代码优化概述



7.5 代码优化概述

B_6 x = a[i]; a[i] = a[n]; a[n] = x;

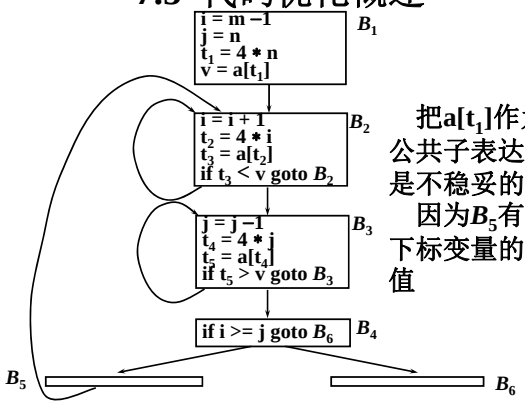
$a[t_1]$能否作为公共子表达式？

B_6

$t_{11} = 4 * i$
$x = a[t_{11}]$
$t_{12} = 4 * i$
$t_{13} = 4 * n$
$t_{14} = a[t_{13}]$
$a[t_{12}] = t_{14}$
$t_{15} = 4 * n$
$a[t_{15}] = x$

$x = t_3$
$t_{14} = a[t_1]$
$a[t_2] = t_{14}$
$a[t_1] = x$

7.5 代码优化概述



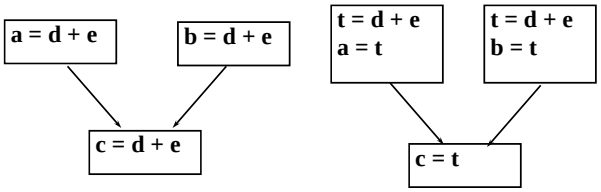
把$a[t_1]$作为公共子表达式是不稳妥的

因为B_5有对下标变量的赋值

7.5 代码优化概述

7.5.4 复写传播

- 复写语句：形式为**f = g**的赋值
- 优化过程中会大量引入复写



删除局部公共子表达式期间引进复写

7.5 代码优化概述

7.5.4 复写传播

- 复写语句：形式为**f = g**的赋值
- 优化过程中会大量引入复写
- 复写传播变换的做法是在复写语句**f = g**后，尽可能用**g**代表**f**

B_5

$x = t_3$
$a[t_2] = t_5$
$a[t_4] = x$
goto B_2

$x = t_3$
$a[t_2] = t_5$
$a[t_4] = t_3$
goto B_2

7.5 代码优化概述

7.5.4 复写传播

- 复写语句：形式为**f = g**的赋值
- 优化过程中会大量引入复写
- 复写传播变换的做法是在复写语句**f = g**后，尽可能用**g**代表**f**
- 复写传播变换本身并不是优化，但它给其他优化带来机会
  - 常量合并（编译时可完成的计算）
  - 死代码删除

7.5 代码优化概述

7.5.5 死代码删除

- 死代码是指计算的结果决不被引用的语句
- 一些优化变换可能会引起死代码

例： 为便于调试，可能在程序中加打印语句，测试后改成右边的形式

```
debug = true;          |   debug = false;
. . .                  |   . . .
if (debug) print …     |   if (debug) print …
```

靠优化来保证目标代码中没有该条件语句部分

7.5 代码优化概述

7.5.5 死代码删除

- 死代码是指计算的结果决不被引用的语句
- 一些优化变换可能会引起死代码

例：复写传播可能会引起死代码删除

B_5

$x = t_3$
$a[t_2] = t_5$
$a[t_4] = x$
goto B_2

$x = t_3$
$a[t_2] = t_5$
$a[t_4] = t_3$
goto B_2

$a[t_2] = t_5$
$a[t_4] = t_3$
goto B_2

7.5 代码优化概述

7.5.6 代码外提

- 代码外提是循环优化的一种
- 循环优化的其他重要技术
  - 归纳变量删除
  - 强度削弱

例：**while (i <= limit – 2) …**

代码外提后变换成

```
t = limit – 2;
while (i <= t ) …
```

7.5 代码优化概述

7.5.7 强度削弱和归纳变量删除

- j和t_4的值步伐一致地变化，这样的变量叫做归纳变量
- 在循环中有多个归纳变量时，也许只需要留下一个
- 这个操作由归纳变量删除过程来完成
- 对本例可以先做强度削弱它给删除归纳变量创造机会

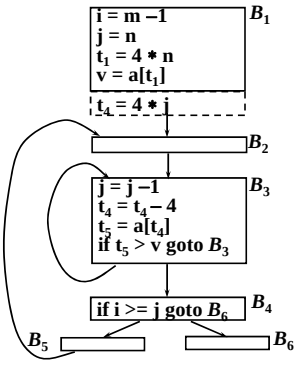
B_3

```
j = j –1
t4 = 4 * j
t5 = a[t4]
if t5 > v goto B3
```

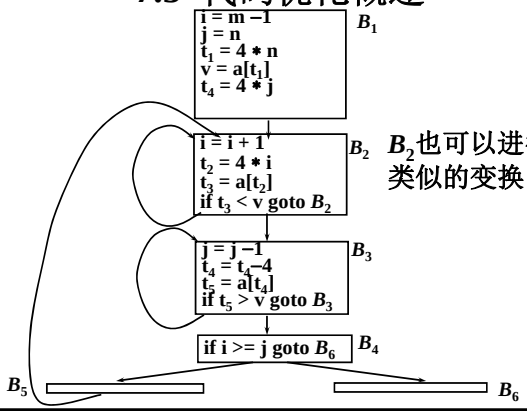
7.5 代码优化概述

```
j = j –1
t4 = 4 * j
t5 = a[t4]
if t5 > v goto B3
```
B_3

除第一次外，
$t_4 == 4 * j$在B_3的入口一定保持
在j = j –1后，
关系$t_4 == 4 * j + 4$也保持

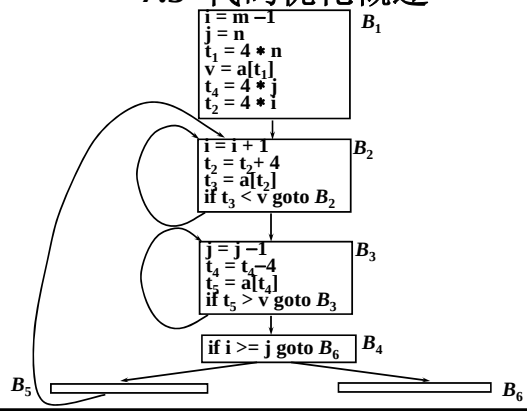


7.5 代码优化概述

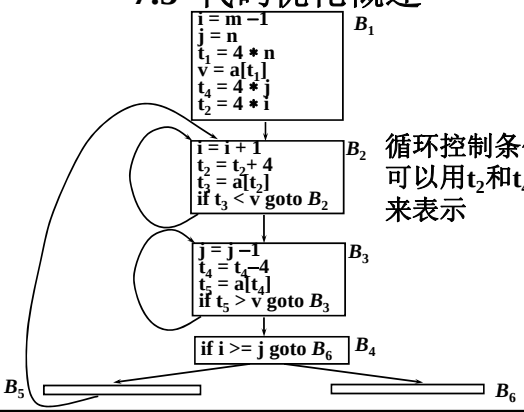


B_2也可以进行类似的变换

7.5 代码优化概述

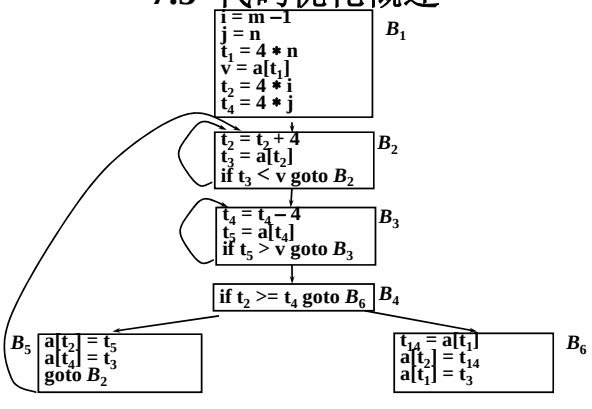


7.5 代码优化概述



循环控制条件可以用t_2和t_4来表示

7.5 代码优化概述



本 章 要 点

- 代码生成器设计中的主要问题：存储管理、计算次序的选择、寄存器的分配、指令的选择等
- 目标机器中几种常用的地址模式和一些常用的指令
- 基本块和程序流图
- 简单的代码生成算法
- 代码优化中主要有哪些代码改进的方式

例 题 1

在SPARC/SunOS上，经某编译器编译，程序的结果是120。把第4行的abs(1)改成1的话，则程序结果是1

```
int fact( ){
   static int i=5;
   if ( i==0 ) { return(1); }
   else { i = i-1; return( (i+abs(1)) * fact( )); }
}
main( ){
   printf("factor of 5 = %d\n", fact( ));
}
```

先完成有函数调用的子表达式的计算

例 题 2

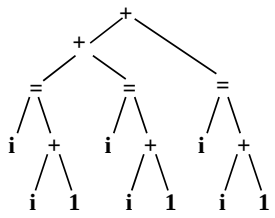
下面的程序在x86/Linux机器上编译后的运行结果是7，而在SPARC/SunOS机器上编译后的运行结果是6。试分析运行结果不同的原因。

```
main( ) {
   long i;

   i = 0;
   printf("%ld\n", (++i)+(++i)+(++i) );
}
```

例 题 2

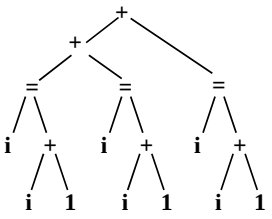
按一般的代码生成，i = i +1的计算结果保留在寄存器中，因此这三个i = i +1的计算次序不会影响最终的结果。结果应该是6



例 题 2

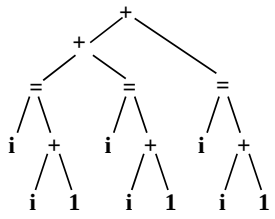
按一般的代码生成，i = i +1的计算结果保留在寄存器中，因此这三个i = i +1的计算次序不会影响最终的结果。结果应该是6

结果是7的话，一定是某个i = i +1的结果未保留在寄存器中。上层计算对它的引用落在计算另一个i = i +1的后面



例 题 2

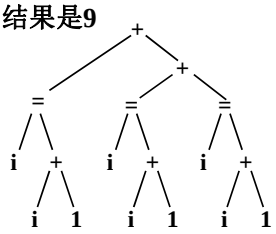
- 如果机器有INC指令的话，编译器极可能产生一条INC指令来完成i = i +1
- x86/Linux机器上果真是这么做的



例　题　2

将表达式改成(++i)+((++i)+(++i)), 结果会怎样？

- 在SPARC/SunOS机器上的结果仍然是6
- 在x86/Linux机器上的结果是9



例　题　3

下面C语言程序如下, 运行时输出105, 为什么？

```
main( ) {
  long i;
  i=10;
  i=(i+5) + (i=i*5);
  printf("%d\n",i); );
}
```

二元运算 需几个R

表达式　　表达式

至少需1个R　至少需2个R

（除基址寄存器外）

寄存器分配策略可能导致先计算右子表达式

注：ubuntu12.04+gcc4.6.3 结果为65

例　题　4

下面是一个C语言程序和在x86/Linux机器上编译（未使用优化）该程序得到的汇编代码见下页（为便于理解，略去了和讨论本问题无关的部分，并改动了一个地方）

```
main( ) {
        long i, j;
        if ( j )
                i++;
        else
                while ( i ) j++;
}
```

例　题　4

```
main( ) {
  long i, j;
  long i, j;
  if ( j )
      i++;
  else
      while ( i ) j++;
}
pushl %ebp
movl %esp,%ebp
subl $8,%esp
cmpl $0,-8(%ebp)
je .L2
```

```
        incl -4(%ebp)
        jmp .L3
.L2: .L4:(写在一行以省空间)
        cmpl $0, -4(%ebp)
        jne .L6
        jmp .L5
.L6:
        incl -8(%ebp)
        jmp .L4
.L5: .L3: .L1:
        leave
        ret
```

例　题　4

- 为什么会出现一条指令前有多个标号的情况，如.L2和.L4，还有.L5、.L3和.L1？从控制流语句的中间代码结构加以解释
- 条件语句和循环语句的中间代码结构如下：

```
if (E) then S1 else S2          while (E) do S
    E的代码                L4:  E的代码
    假转 L2                     真转 L6
    S1的代码                    转  L5
    转 L3                  L6:  S的代码
L2: S2的代码                    转 L4
L3:                        L5:
```

- 当while语句作为条件语句的S2时，会出现所说情况

例　题　4

- 每个函数都有这样的标号.L1，它的作用是什么，为什么本函数没有引用该标号的地方？

.L1标号定义的入口是返回调用者时该执行的指令，在函数内部有return语句时就会跳转到.L1

例 题 5

一个C语言程序

```
main( )
{
  long i,j;

  while (i) {
     if (j) { i = j; }
  }
}
```

生成的汇编码见右边

为什么会有连续跳转？

```
    pushl %ebp
    movl %esp,%ebp
    subl $8,%esp
.L2:
    cmpl $0,-4(%ebp)
    jne .L4
    jmp .L3
.L4:
    cmpl $0,-8(%ebp)
    je .L5
    movl -8(%ebp),%eax
    movl %eax,-4(%ebp)
.L5:
    jmp .L2
.L3:
```

例 题 5

```
while E1 do S1
L2:  E1的代码
     真转 L4
     无条件转 L3
L4:  S1的代码
     JMP L2
L3:
if E2 then S2
     E2的代码
     假转 L5
     S2的代码
L5:
```

例 题 5

```
while E1 do S1
L2:  E1的代码
     真转 L4
     无条件转 L3
L4:  S1的代码
     JMP L2
L3:
if E2 then S2
     E2的代码
     假转 L5
     S2的代码
L5:
```

当它们嵌套时，代码结构变成

```
L2:  E1的代码
     真转 L4
     无条件转 L3
L4:  E2的代码
     假转 L5
     S2的代码
L5:  JMP L2
L3:
```

例 题 5

```
while E1 do S1
L2:  E1的代码
     真转 L4
     无条件转 L3
L4:  S1的代码
     JMP L2
L3:
if E2 then S2
     E2的代码
     假转 L5
     S2的代码
L5:
```

当它们嵌套时，代码结构变成

```
L2:  E1的代码
     真转 L4
     无条件转 L3
L4:  E2的代码
     假转 L5
     S2的代码
L5:  JMP L2
L3:
```

例 题 5

```
while E1 do S1
L2:  E1的代码
     真转 L4
     无条件转 L3
L4:  S1的代码
     JMP L2
L3:
if E2 then S2
     E2的代码
     假转 L5
     S2的代码
L5:
```

当它们嵌套时，代码结构变成

```
L2:  E1的代码
     真转 L4
     无条件转 L3
L4:  E2的代码
     假转 L5
     S2的代码
L5:  JMP L2
L3:
```

例 题 5

```
while E1 do S1
L2:  E1的代码
     真转 L4
     无条件转 L3
L4:  S1的代码
     JMP L2
L3:
if E2 then S2
     E2的代码
     假转 L5
     S2的代码
L5:
```

可优化为假转

当它们嵌套时，代码结构变成

```
L2:  E1的代码
     真转 L4
     无条件转 L3
L4:  E2的代码
     假转 L5
     S2的代码
L5:  JMP L2
L3:
```

例　题　5

一个C语言程序

```
main()
{
   long i,j;

   while (i) {
        if (j) { i = j; }
   }
}
```

优化编译的汇编码见右边

```
    pushl %ebp
    movl %esp,%ebp
.L7:
    testl %eax,%eax
    je .L3
    testl %edx,%edx
    je .L7
    movl %edx,%eax
    jmp .L7
.L3:
```

例　题　6

UNIX 下的C编译命令cc的选择项g和O的解释如下, 其中dbx的解释是“dbx is an utility for source-level debugging and execution of programs written in C”。试说明为什么用了选择项g后，选择项O便被忽略。

-g Produce additional symbol table information for dbx(1) and dbxtool(1) and pass -lg option to ld(1) (so as to include the g library, that is: /usr/lib/libg.a). When this option is given, the -O and -R options are suppressed.

-O[level] Optimize the object code. Ignored when either -g, -go, or -a is used. ...

例　题　7

一个C语言程序如下，右边是优化后的目标代码

```
main( )
{
   int i, j, k;
   i=5;
   j=1;
   while (j<100) {
        k=i+1;
        j=j+k;
   }
}
```

完成了哪些优化？

```
    pushl %ebp
    movl %esp,%ebp
    movl $1,%eax      -- j=1
    movl $6,%edx      -- k=6
L4:
    addl %edx,%eax -- j=j+6
    cmpl $99,%eax
    jle .L4           -- while(j≤99)
```

例　题　7

一个C语言程序如下，右边是优化后的目标代码

```
main()
{
   int i,j,k;
   i=5;
   j=1;
   while(j<100){
        k=i+1;
        j=j+k;
   }
}
```

```
    pushl %ebp
    movl %esp,%ebp
    movl $1,%eax      -- j=1
    movl $6,%edx      -- k=6
L4:
    addl %edx,%eax -- j=j+6
    cmpl $99,%eax
    jle .L4           -- while(j≤99)
```

复写传播、常量合并、代码外提、删除无用赋值

对i，j和k分配内存单元也成为多余，从而被取消

例　题　8

求最大公约数的函数

```
long gcd(p,q)
long p,q;
{
   if (p%q == 0)
        return q;
   else
        return gcd(q, p%q);
}
```

- 其中的递归调用称为尾递归
- 对于尾递归，编译器可以产生和一般的函数调用不同的代码，使得目标程序运行时，这种递归调用所需的存储空间大大减少，也缩短了运行时间
- 对于尾递归，编译器应怎样产生代码?

例　题　8

求最大公约数的函数

```
long gcd(p,q)
long p,q;
{
   if (p%q == 0)
        return q;
   else
        return gcd(q, p%q);
}
```

- 计算实在参数q和p%q，存放在不同的寄存器中
- 将上述寄存器中实在参数的值存入当前活动记录中形式参数p和q的存储单元
- 转到本函数第一条语句的起始地址继续执行

例　题　8

求最大公约数的函数

```
long gcd(p,q)
long p,q;
{
   if (p%q == 0)
        return q;
   else
        return gcd(q, p%q);
}
```

```
   movl 8(%ebp),%esi     p
   movl 12(%ebp),%ebx  q
.L4:
   movl %esi,%eax
   cltd                扩展为64位
   idivl %ebx
   movl %edx,%ecx       p%q
   testl %ecx,%ecx      p%q
   je .L2
   movl %ebx,%esi       q⇒p
   movl %ecx,%ebxp%q⇒q
   jmp .L4
.L2:
```

例　题　9

C语言程序引用sizeof（求字节数运算符）时，该运算是在编译该程序时完成，还是在运行该程序时完成？说明理由。

习　题

- 第1次：7.4（只为7.1(e)生成代码）
- 第2次：7.9, 7.10, 7.16