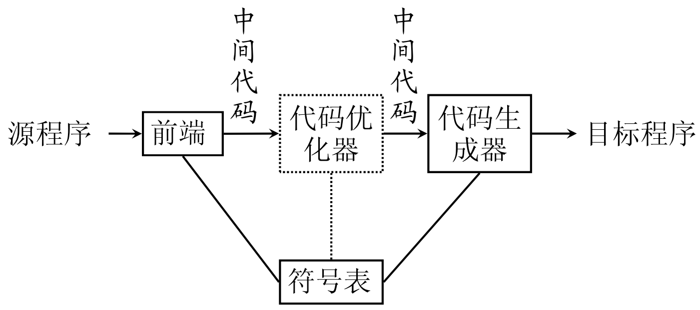
第十章 代码生成与优化概述

- 概述
- 代码生成的基本概念
- 基本块的优化（局部优化）
- 寄存器分配与指派
- 流图中的循环

10.1 概述



Basic Block Optimizations

- Common Sub-Expression Elimination删除公共子表达式
  - a = (x+y)+z; b = x+y;
  - t = x+y; a = t+z; b = t;
- Constant Propagation常数传播/常量折叠
  - x = 5; b = x+y;
  - b = 5+y;
- Algebraic Identities代数恒等式/代数化简
  - a = x * 1;
  - a = x;

Basic Block Optimizations

- Copy Propagation复写传播
  - a = x+y; b = a; c = b+z;
  - a = x+y; b = a; c = a+z;
- Dead Code Elimination删除无用代码
  - a = x+y; b = a; c = a+z;
  - a = x+y; c = a+z
- Strength Reduction强度削弱
  - t = i * 4;
  - t = i << 2;

循环优化

- 代码外提
- 删除归纳变量
- 强度削弱*

代码生成器设计中的问题

- 代码生成器的输入
- 目标程序
- 存储管理
- 指令选择
- 寄存器分配
- 计算次序的选择
- 代码生成方法

代码生成器的输入

- 源程序的中间表示
  - 可有多种表示方法
  - 名字的值可为目标机器直接操作
  - 已完成必要的类型检查，插入了类型转换操作
  - 一般没有语义错误
- 符号表信息
  - 中间表示中名字所代表的数据对象的运行时地址

目标程序

- 绝对机器语言
  - 可立即执行
- 可重定位的机器语言
  - 可分块编译，并链接
- 汇编代码
  - 代码生成容易

指令选择

- **IR**映射为目标代码序列
  - **IR**的层次
  - 指令集体系结构本身的特性
  - 想要达到的生产代码的质量

寄存器分配

- 在寄存器分配期间，在程序的某一点选择要驻留在寄存器中的变量集
- 在随后的寄存器指派阶段，挑出变量将要驻留的具体机器
- 选择最优的寄存器指派方案是**NP**完全的
- 目标机器对寄存器使用的某些约定使分配更复杂

计算次序的选择

- 计算执行的次序影响目标代码效率
- 也会影响使用寄存器的多寡
- 选择最佳次序也是**NP**完全问题

10.2 代码优化及生成基本概念

- CS164: Programming Languages and Compilers, Spring 2008
- **University of Berkeley**

- 6.035 Computer Language Engineering (SMA 5502) Fall 2005
- ***MIT*** OpenCourseWare

定义：基本块

- **A *basic block* is a maximal sequence of instructions with:**
  - **no labels (except at the first instruction), and**
  - **no jumps (except in the last instruction)**
- 基本块是具有如下性质的指令序列
  - 基本块的中间不会有分支转出
  - 也没有转入到基本块中间的分支
  - 基本块应当是最大化的
- 基本块的执行是从它的第一条指令开始

Idea about Basic Blocks

- **Cannot jump in a basic block (except at beginning)**
- **Cannot jump out of a basic block (except at end)**
- **Each instruction in a basic block is executed after all the preceding instructions have been executed**

Basic Block Example

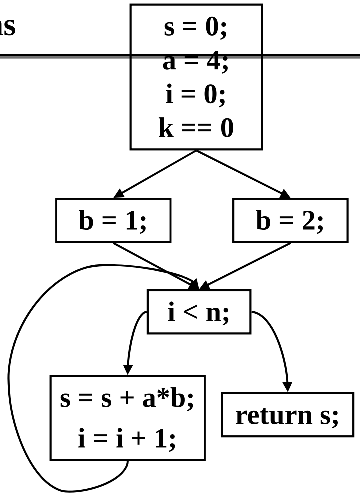
- **Consider the basic block**
  1. **L:**
  2. **t := 2 * x**
  3. **w := t + x**
  4. **if w > 0 goto L′**
- **No way for (3) to be executed without (2) having been executed right before**
  - **We can change (3) to w := 3 * x**
  - **Can we eliminate (2) as well?**

定义. 控制流图

- **A *control-flow graph* is a directed graph with**
  - **Basic blocks as nodes**
  - **An edge from block A to block B if the execution can flow from the last instruction in A to the first instruction in B**
  - 例**. A**中最后一条指令是 **jump L_B**
  - 例**.** 从块**A**到块**B**的执行可能不成功
- 通常缩写为 **CFG**

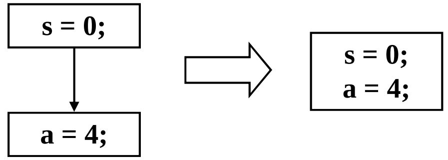
Control Flow Graphs

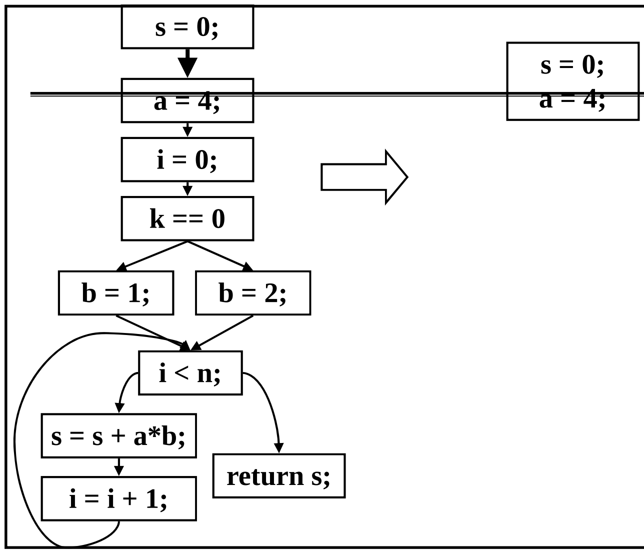
int add(n, k) {
s = 0; a = 4; i = 0;
if (k == 0) b = 1;
else b = 2;
while (i < n) {
s = s + a*b;
i = i + 1;
}
return s;
}

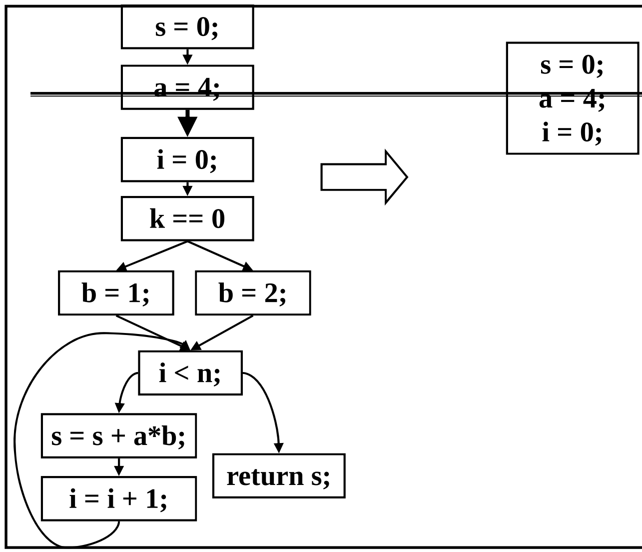


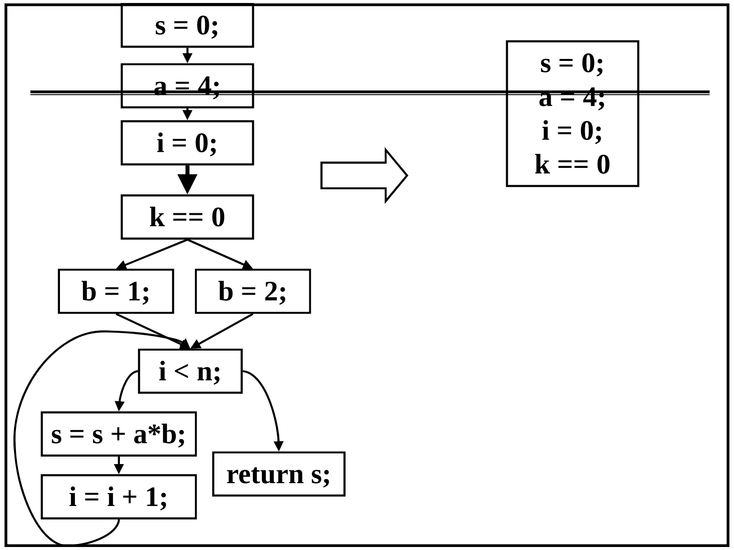
Basic Block Construction

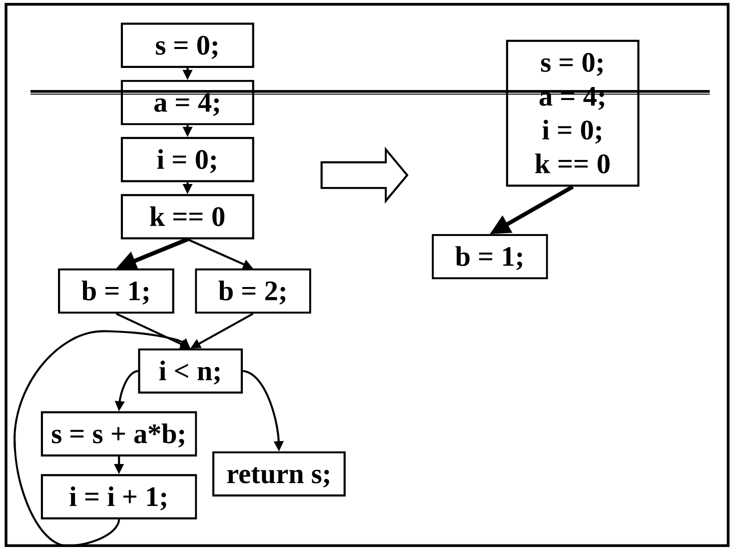
- Start with instruction control-flow graph指令级的CFG
- Visit all edges in graph
- Merge adjacent nodes if
  - Only one edge from first node
  - Only one edge into second node

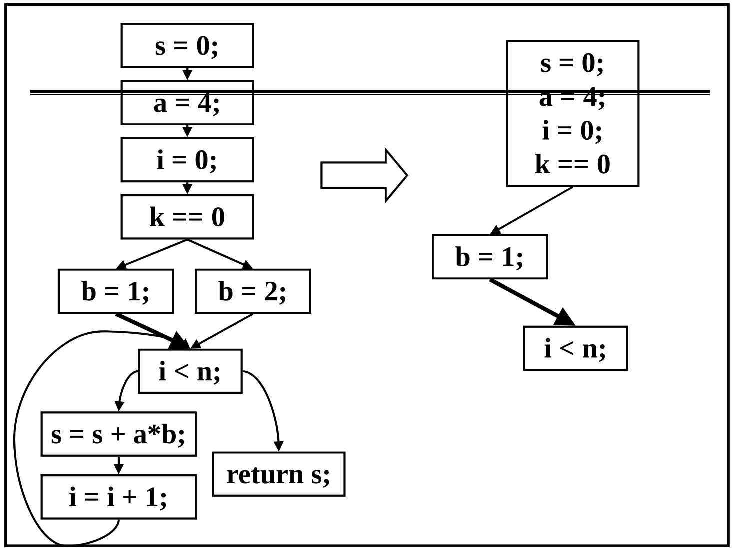


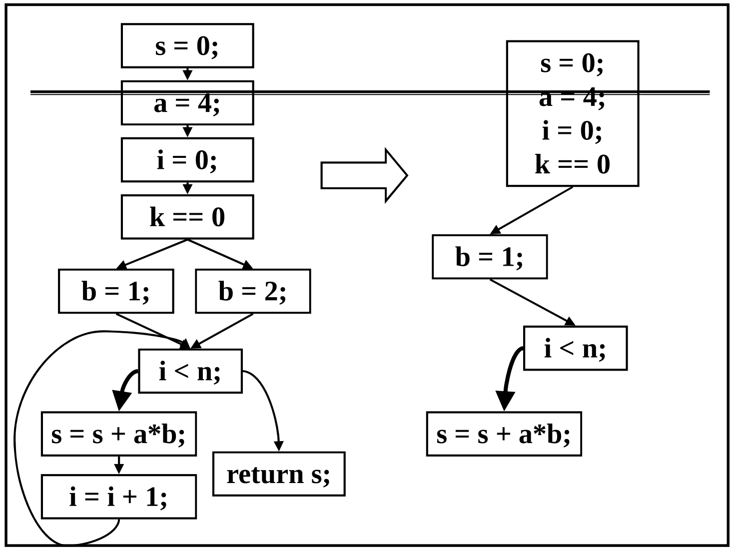


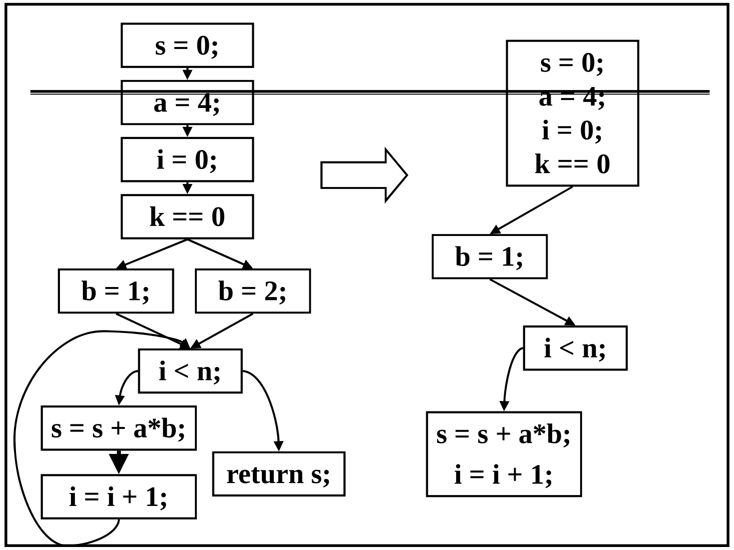


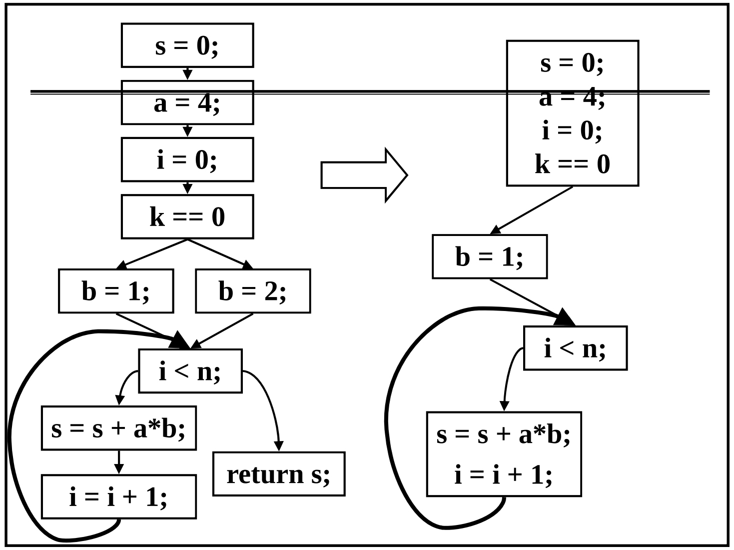

s = 0;
a = 4;
i = 0;
k == 0
b = 1;
b = 2;
i < n;
s = s + a*b;
i = i + 1;
return s;
s = 0;
a = 4;
i = 0;
k == 0

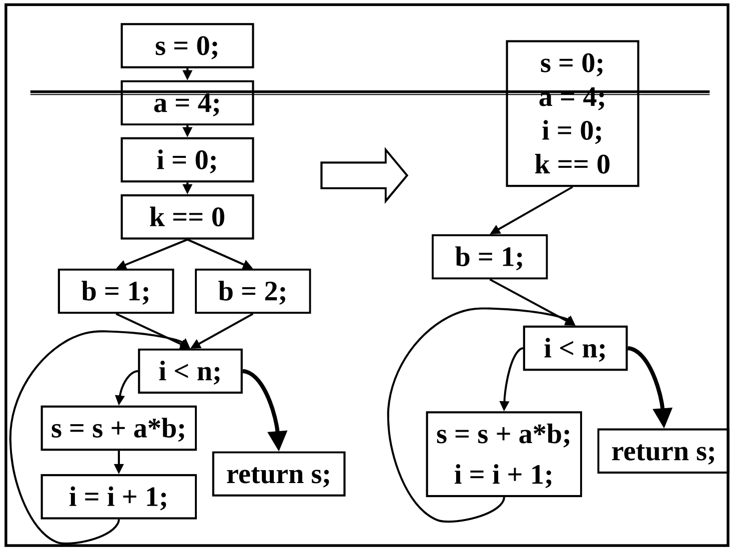

s = 0;
a = 4;
i = 0;
k == 0
b = 1;
b = 2;
i < n;
s = s + a*b;
i = i + 1;
return s;
s = 0;
a = 4;
i = 0;
k == 0
b = 1;

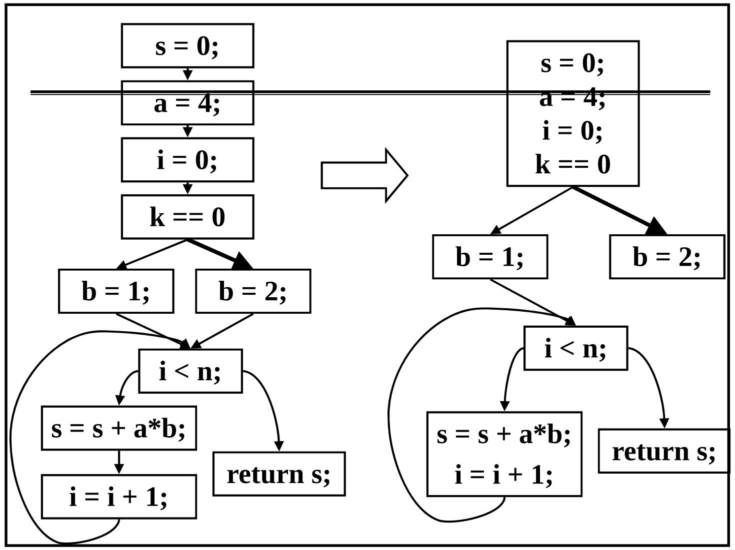

s = 0;
a = 4;
i = 0;
k == 0
b = 1;
b = 2;
i < n;
s = s + a*b;
i = i + 1;
return s;
s = 0;
a = 4;
i = 0;
k == 0
b = 1;
i < n;

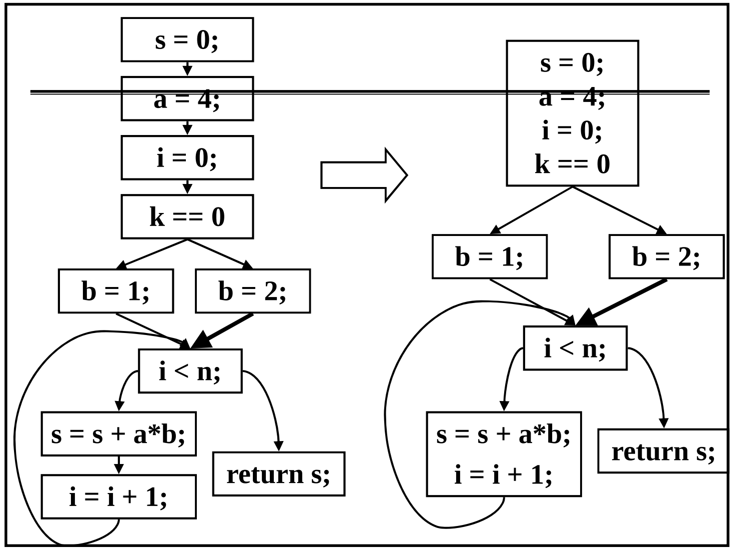

s = 0;
a = 4;
i = 0;
k == 0
b = 1;
b = 2;
i < n;
s = s + a*b;
i = i + 1;
return s;
s = 0;
a = 4;
i = 0;
k == 0
b = 1;
i < n;
s = s + a*b;

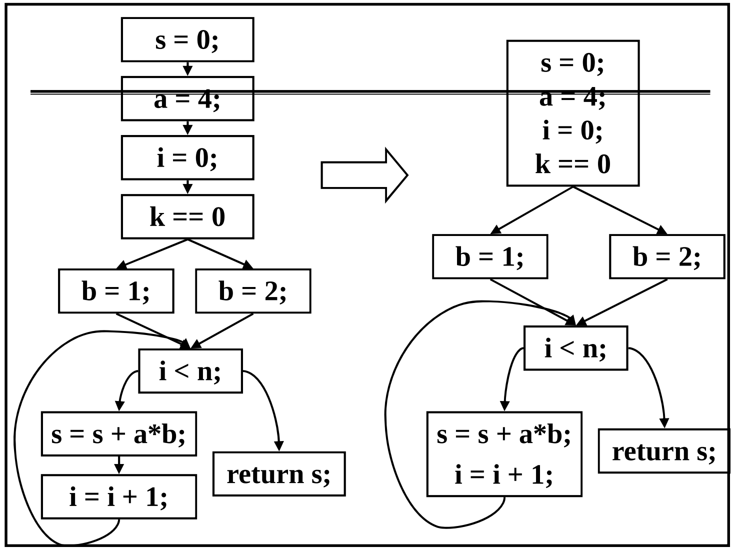

s = 0;
a = 4;
i = 0;
k == 0
b = 1;
b = 2;
i < n;
s = s + a*b;
i = i + 1;
return s;
s = 0;
a = 4;
i = 0;
k == 0
b = 1;
i < n;
s = s + a*b;
i = i + 1;


s = 0;
a = 4;
i = 0;
k == 0
b = 1;
b = 2;
i < n;
s = s + a*b;
i = i + 1;
return s;
s = 0;
a = 4;
i = 0;
k == 0
b = 1;
i < n;
s = s + a*b;
i = i + 1;


s = 0;
a = 4;
i = 0;
k == 0
b = 1;
b = 2;
i < n;
s = s + a*b;
i = i + 1;
return s;
s = 0;
a = 4;
i = 0;
k == 0
b = 1;
i < n;
s = s + a*b;
i = i + 1;
return s;


s = 0;
a = 4;
i = 0;
k == 0
b = 1;
b = 2;
i < n;
s = s + a*b;
i = i + 1;
return s;
s = 0;
a = 4;
i = 0;
k == 0
b = 1;
b = 2;
i < n;
s = s + a*b;
i = i + 1;
return s;





Optimization Overview

- **Optimization seeks to improve a program's utilization of some resource**优化试图改善程序对某资源的利用
  - **Execution time (most often)**
  - **Code size**
  - **Network messages sent**
  - **Battery power used, etc.**
- **Optimization should not alter what the program computes**优化不该改变程序功能
  - **The answer must still be the same**

优化的分类

- **For languages like C and Cool there are three granularities of optimizations**按粒度来分
  1. **Local optimizations**
     - **Apply to a basic block in isolation**
  2. **Global optimizations**
     - **Apply to a control-flow graph (method body) in isolation**
  3. **Inter-procedural optimizations**
     - **Apply across method boundaries**
- **Most compilers do (1), many do (2) and very few do (3)**

Cost of Optimizations

- 实际中， a conscious decision is made not to implement the fanciest optimization known
- Why?
  - Some optimizations are hard to implement
  - Some optimizations are costly in terms of compilation time
  - The fancy optimizations are both hard and costly
- The goal: maximum improvement with minimum of cost

Local Optimizations

- The simplest form of optimizations
- No need to analyze the whole procedure body
  - Just the basic block in question
- Example: algebraic simplification

Algebraic Simplification

- Some statements can be deleted

  x := x + 0

  x := x * 1
- Some statements can be simplified

  x := x * 0 ⇒ x := 0

  y := y ** 2 ⇒ y := y * y

  x := x * 8 ⇒ x := x << 3

  x := x * 15 ⇒ t := x << 4; x := t - x

  (on some machines << is faster than *; but not on all!)

Constant Folding常量折叠

- Operations on constants can be computed at compile time
- In general, if there is a statement

  x := y op z
  - And y and z are constants
  - Then y op z can be computed at compile time
- Example: x := 2 + 2 ⇒ x := 4
- Example: if 2 < 0 jump L can be deleted

控制流优化

- **Eliminating unreachable code:**
  - **Code that is unreachable in the control-flow graph**
  - **Basic blocks that are not the target of any jump or "fall through" from a conditional**
  - **Such basic blocks can be eliminated**
- **Why would such basic blocks occur?**
- **Removing unreachable code makes the program smaller**
  - **And sometimes also faster, due to memory cache effects (increased spatial locality)**

Single Assignment Form

- **Some optimizations are simplified if each assignment is to a temporary that has not appeared already in the basic block变量只定值一次**
- **Intermediate code can be rewritten to be in single assignment form**

$$
\begin{array}{lcl}
x := a + y & & x := a + y \\
a := x & \Rightarrow & a_1 := x \\
x := a * x & & x_1 := a_1 * x \\
b := x + a & & b := x_1 + a_1
\end{array}
$$

(x_1 and a_1 are fresh temporaries)

Common Subexpression Elimination

- **Assume**
  - **Basic block is in *single assignment form***
- **All assignments with same rhs compute the same value**
- **Example:**

$$
\begin{array}{lcl}
x := y + z & & x := y + z \\
\ldots & \Rightarrow & \ldots \\
w := y + z & & w := x
\end{array}
$$

- **Why is single assignment important here?**

Copy Propagation

- **If w := x appears in a block, all subsequent uses of w can be replaced with uses of x**
- **例:**

$$
\begin{array}{lcl}
b := z + y & & b := z + y \\
a := b & \Rightarrow & a := b \\
x := 2 * a & & x := 2 * b
\end{array}
$$

- **This does not make the program smaller or faster but might enable other optimizations**
  - **Constant folding**
  - **Dead code elimination**
- **Again, single assignment is important here.**

Copy Propagation and Constant Folding

- Example:

a := 5		a := 5
x := 2 * a	⇒	x := 10
y := x + 6		y := 16
t := x * y		t := x << 4

Dead Code Elimination

- If
  - w := rhs appears in a basic block
  - w does not appear anywhere else in the program
- Then
  - the statement w := rhs is dead and can be eliminated
- Dead = does not contribute to the program's result

Example: (a is not used anywhere else)

x := z + y		b := z + y		b := z + y
a := x	⇒	a := b	⇒	x := 2 * b
x := 2 * a		x := 2 * b		

Applying Local Optimizations

- Each local optimization does very little by itself
- Typically optimizations interact
  - Performing one optimizations enables other opt.
- Typical optimizing compilers repeatedly perform optimizations until no improvement is possible
  - The optimizer can also be stopped at any time to limit the compilation time

An Example

- Initial code:

```
a := x ** 2
b := 3
c := x
d := c * c
e := b * 2
f := a + d
g := e * f
```

- **Algebraic optimization:**

  a := x ** 2
  b := 3
  c := x
  d := c * c
  e := b * 2
  f := a + d
  g := e * f

- **Algebraic optimization:**

  a := x * x
  b := 3
  c := x
  d := c * c
  e := b + b
  f := a + d
  g := e * f

- **Copy propagation:**

  a := x * x
  b := 3
  c := x
  d := c * c
  e := b + b
  f := a + d
  g := e * f

- **Copy propagation:**

  a := x * x
  b := 3
  c := x
  d := x * x
  e := 3 + 3
  f := a + d
  g := e * f

- **Constant folding:**

a := x * x
b := 3
c := x
d := x * x
e := 3 + 3
f := a + d
g := e * f

- **Constant folding:**

a := x * x
b := 3
c := x
d := x * x
e := 6
f := a + d
g := e * f

- **Common subexpression elimination:**

a := x * x
b := 3
c := x
d := x * x
e := 6
f := a + d
g := e * f

- **Common subexpression elimination:**

a := x * x
b := 3
c := x
d := a
e := 6
f := a + d
g := e * f

- **Copy propagation:**

a := x * x
b := 3
c := x
d := a
e := 6
f := a + d
g := e * f

- **Copy propagation:**

a := x * x
b := 3
c := x
d := a
e := 6
f := a + a
g := 6 * f

- **Dead code elimination:**

a := x * x
b := 3
c := x
d := a
e := 6
f := a + a
g := 6 * f

- **Dead code elimination:**

a := x * x

f := a + a
g := 6 * f

- **This is the final form**

Peephole Optimizations on Assembly Code

- The optimizations presented before work on intermediate code
  - They are target independent
  - But they can be applied on assembly language also
- *Peephole optimization* is an effective technique for improving assembly code窥孔优化
  - The "peephole" is a short sequence of (usually contiguous) instructions
  - The optimizer replaces the sequence with another equivalent (but faster) one

Peephole Optimizations (Cont.)

- Write peephole optimizations as replacement rules

  $i_1, \ldots, i_n \to j_1, \ldots, j_m$

  where the rhs is the improved version of the lhs
- Examples:

  move $a $b, move $b $a → move $a $b
  - Works if move $b $a is not the target of a jump

  addiu $a $b k, lw $c ($a) → lw $c k($b)
  - Works if $a not used later (is "dead")

MIPS指令

- `addiu d,s,const`
- # $d <-- s + const.
- # Const is 16-bit two's comp. sign-extended to 32 bits
- # when the addition is done. No overflow trap.

- `lw register_destination, RAM_source`
- #copy word (4 bytes) at source RAM location to destination register.

Peephole Optimizations (Cont.)

- Many (but not all) of the basic block optimizations can be cast as peephole optimizations
  - Example: addiu $a $b 0 → move $a $b
  - Example: move $a $a →
  - These two together eliminate addiu $a $a 0
- Just like for local optimizations, peephole optimizations need to be applied repeatedly to get maximum effect

Local Optimizations. Notes.

- **Intermediate code is helpful for many optimizations**
- **Many simple optimizations can still be applied on assembly language**

Local Optimizations. Notes (II).

- **Serious problem: what to do with pointers?**
  - ***t may change even if local variable t does not: *Aliasing***
  - **Arrays are a special case (address calculation)**
- **What to do about globals?**
- **What to do about calls?**
  - **Not exactly jumps, because they (almost) always return.**
  - **Can modify variables used by caller**
- **Next: global optimizations**

10.3 寄存器分配与指派

- 给目标程序中的具体值分配某些寄存器
  - 如：基地址分配一组；算数运算一组；栈指针分配一个固定寄存器等
- 全局寄存器分配
  - 将寄存器分配给频繁使用的基本块间的活跃变量
  - 将循环中经常使用的值保存在固定的寄存器中
  - 语言中的寄存器变量让程序员直接执行寄存器分配操作

图染色法寄存器分配

- Outline
- What is register allocation
- Webs
- 干涉图Interference Graphs
- 图着色Graph coloring
- 溢出Spilling
- 分裂Splitting
- More optimizations (略)

- 本节内容来自 **6.035** ©MIT Fall 1999

Storing values between def and use

- Program computes with values
  - value definitions (where computed)
  - value uses (where read to compute new values)
- Values must be stored between def and use
- First Option
  - store each value in memory at definition
  - retrieve from memory at each use
- Second Option
  - store each value in register at definition
  - retrieve value from register at each use

Issues

- On a typical RISC architecture
  - All computation takes place in registers
  - Load instructions and store instructions transfer values between memory and registers
- Add two numbers, values in memory
  - load r1, 4(sp)
  - load r2, 8(sp)
  - add r3,r1,r2
  - store r3, 12(sp)

Issues

- Fewer instructions when using registers
  - Most instructions are register-to-register
  - Additional instructions for memory accesses
- Registers are faster than memory
  - wider gap in faster, newer processors
  - Factor of about 4 bandwidth, factor of about 3 latency
  - Could be bigger if program characteristics were different
- But only a small number of registers available
  - Usually 32 integer and 32 floating-point registers
  - Some of those registers have fixed users (r0, ra, sp, fp)

Register Allocation

- Deciding which values to store in limited number of registers
- Register allocation has a direct impact on performance
  - Affects almost every statement of the program
  - Eliminates expensive memory instructions
  - # of instructions goes down due to direct manipulation of registers (no need for load and store instructions)
  - Probably is the optimization with the most impact!

What can be put in a register?

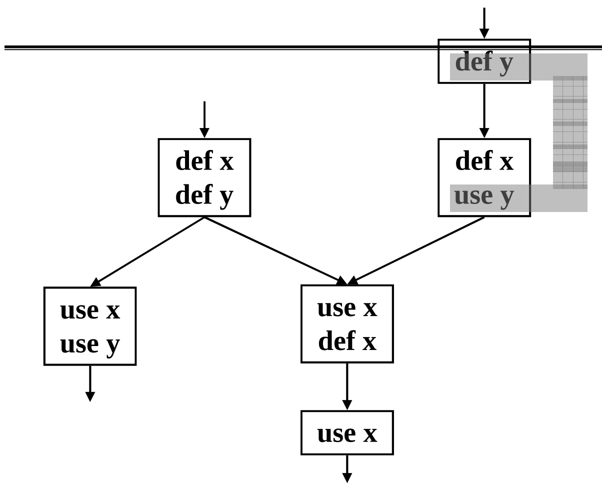
- Values stored in compiler-generated temps
- Language-level values
  - Values stored in local scalar variables
  - Big constants
  - Values stored in array elements and object fields
- Issue: alias analysis
- Register set depends on the data-type
  - floating-point values in floating point registers
  - integer and pointer values in integer registers

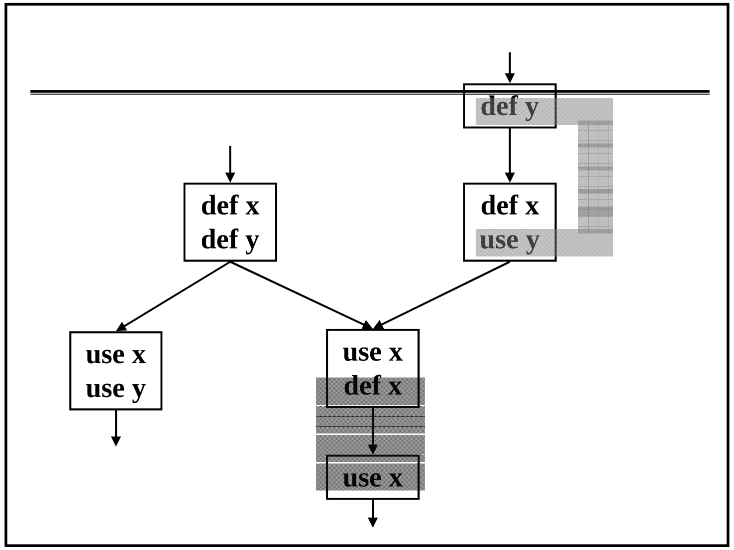
Web-Based Register Allocation

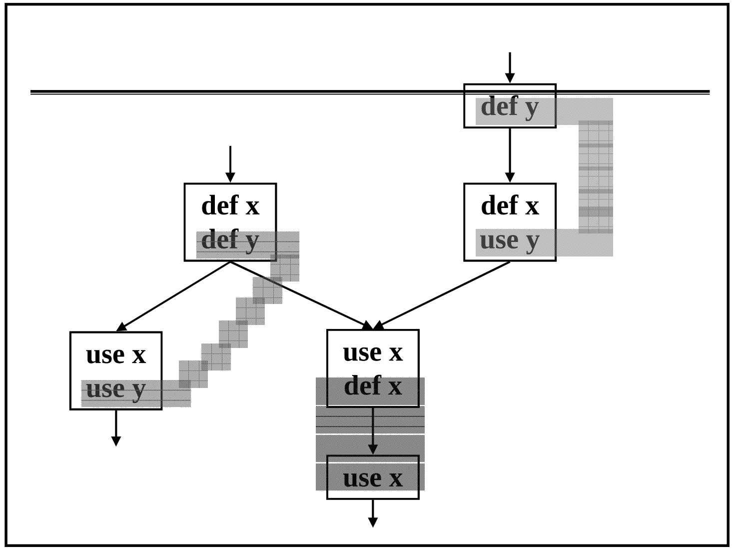
- Determine live ranges for each value (*web*)
- Determine overlapping ranges (interference)
- Compute the benefit of keeping each web in a register (spill cost)
- Decide which webs get a register (allocation)
- Split webs if needed (spilling and splitting) 溢出和分裂
- Assign hard registers to webs (assignment)
- Generate code including spills (code gen)

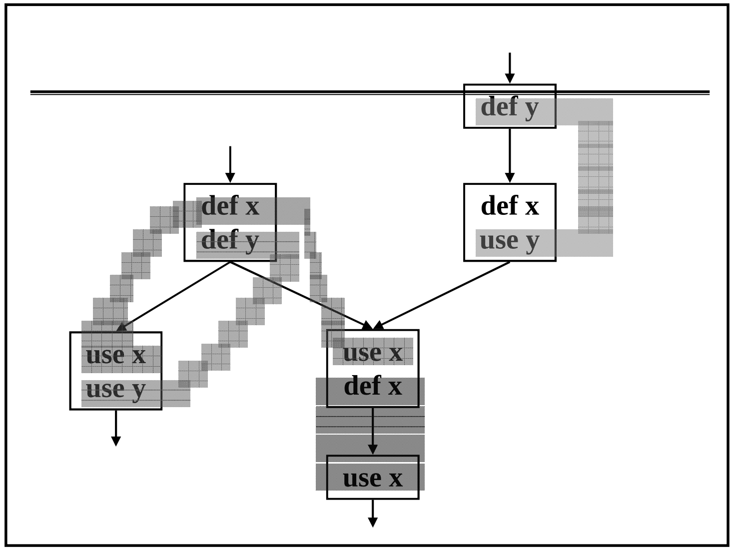
Webs

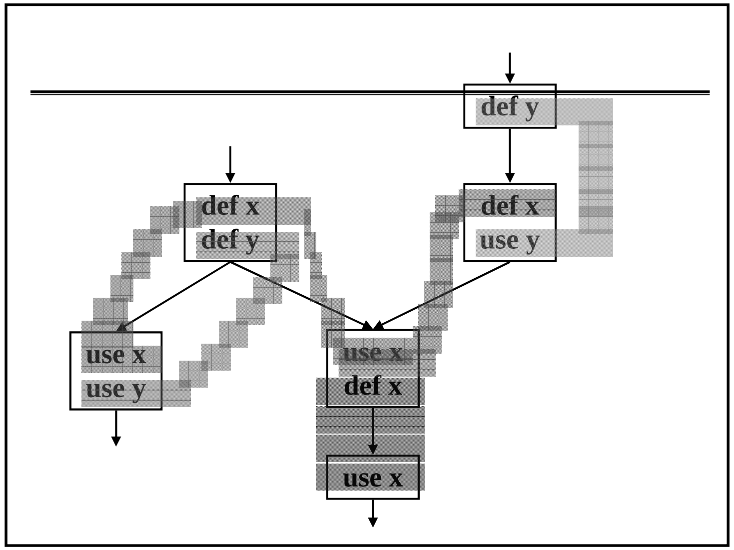
- Starting Point: def-use chains (DU chains)
  - Connects definition to all reachable uses
- Conditions for putting defs and uses into same web
  - Def and all reachable uses must be in same web
  - All defs that reach same use must be in same web
- Use a union-find algorithm

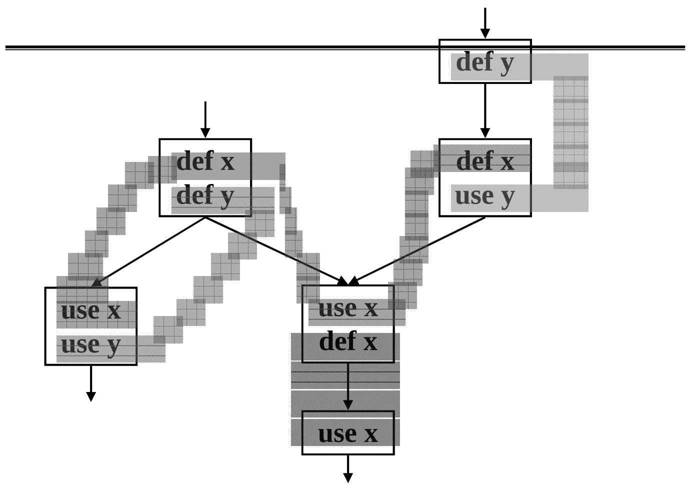


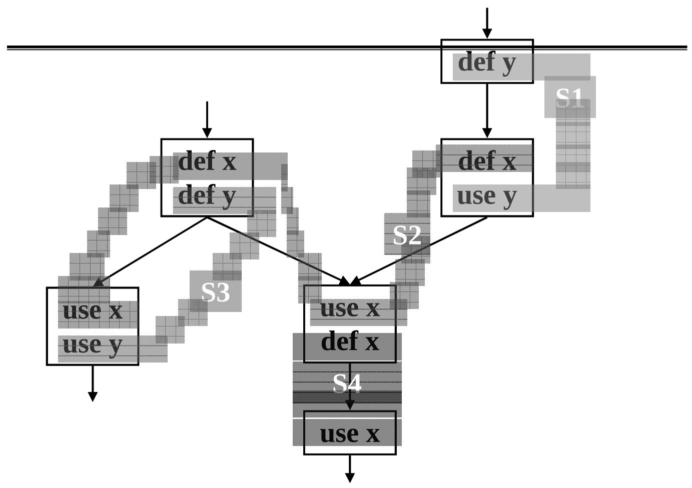

def y
def x
def y
def x
use y
use x
use y
use x
def x
use x


def y
def x
def y
def x
use y
use x
use y
use x
def x
use x


def y
def x
def y
def x
use y
use x
use y
use x
def x
use x


def y
def x
def y
def x
use y
use x
use y
use x
def x
use x





Webs

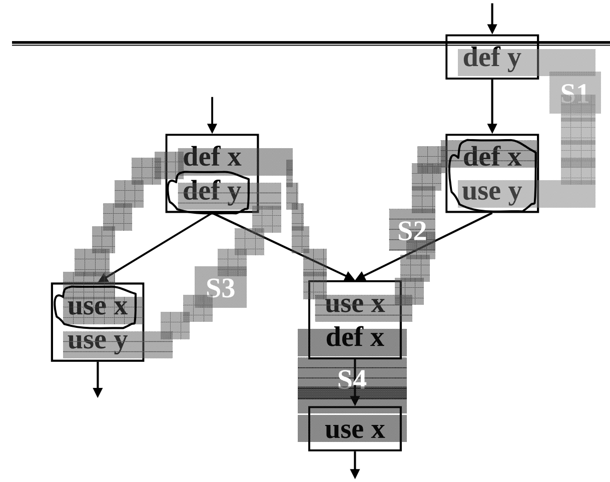
- Web is unit of register allocation
- If web allocated to a given register R
  - All definitions computed into R
  - All uses read from R
- If web allocated to a memory location M
  - All definitions computed into M
  - All uses read from M
- Issue: instructions compute only from registers
- Reserve some registers to hold memory values

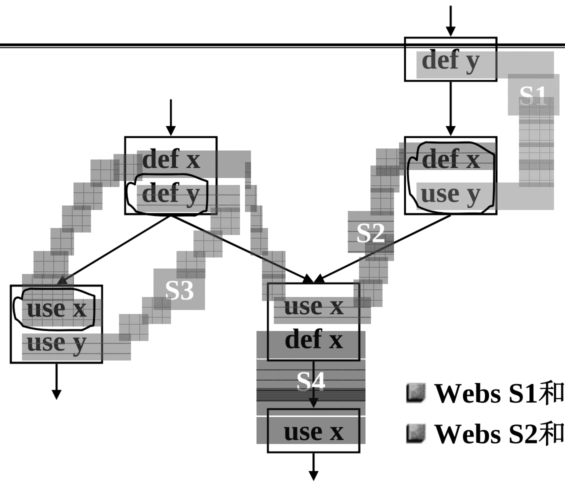
Convex Sets and Live Ranges

- Concept of convex set 凸集
- A set S is convex if
  - A, B in S and C is on a path from A to B implies
  - C is in S
- Concept of live range of a web
  - Minimal convex set of instructions that includes all defs and uses in web
  - Intuitively, region in which web's value is live

Interference

- Two webs interfere if their live ranges overlap (have a nonemtpy intersection)
- If two webs interfere, values must be stored in different registers or memory locations
- If two webs do not interfere, can store values in same register or memory location

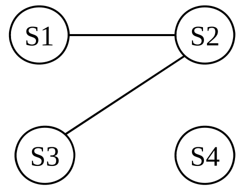




- **Webs S1和S2干涉**
- **Webs S2和S3干涉**

Interference Graph

- Representation of webs and their interference
  - Nodes are the webs
  - An edge exists between two nodes if they interfere



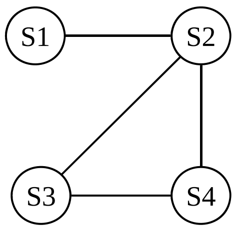
Register Allocation Using Graph Coloring

- Each web is allocated a register
  - each node gets a register (color)
- If two webs interfere they cannot use the same register
  - if two nodes have an edge between them, they cannot have the same color

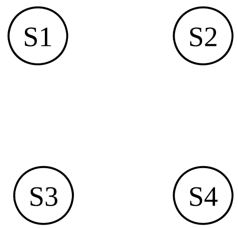
Graph Coloring

- Assign a color to each node in graph
- Two nodes connected to same edge must have different colors
- Classic problem in graph theory
- NP complete
  - But good heuristics exist for register allocation

例. 图着色

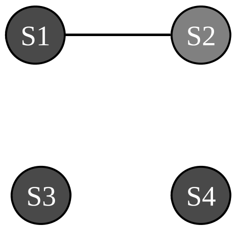


例. 图着色



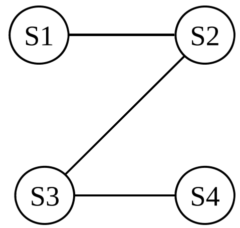
- **1**个颜色

例. 图着色



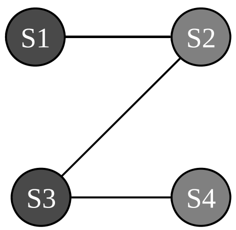
- 2个颜色

例. 图着色



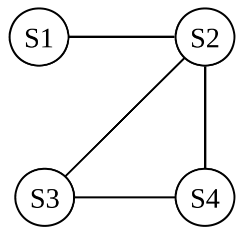
- 2个颜色

例. 图着色



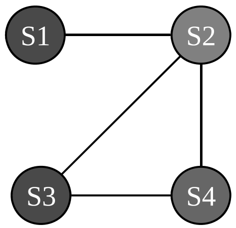
- 2个颜色

例. 图着色



- 3个颜色

例. 图着色



- **3个颜色**

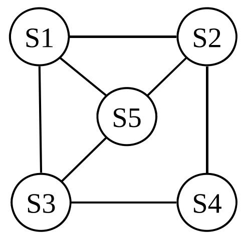
寄存器着色的启发式方法

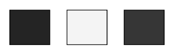
- 用N种颜色进行图着色
- 如果度<N（网点的度＝弧的数目）
  - 则网点总是可以着色的
  - 在对剩余的点着色完毕后，必须至少还有一种颜色来对当前点进行着色
- 如果度>=N
  - 仍然有N种颜色来进行着色

寄存器着色的启发式方法cont.

- 删去度<N的网点
  - 将其放入栈中
- 当所有的网点都>=N时
  - 找出可溢出的网点（不对该网点着色）
  - 删除这个网点
- 当为空时，开始着色
  - 从栈中弹出一个网点
  - 给该网点分配一个不同于它连接网点的颜色（因为度<N，仍然存在这这样一种颜色）

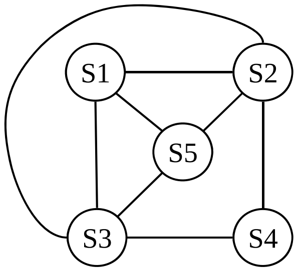
例. 图着色例子



- **3个颜色** 

S4;S2;S1;S3

例. 图着色例子



- **3**个颜色

S4;S2;S1;S3

溢出

- Option 1
  - Pick a web and allocate value in memory
  - All defs go to memory, all uses come from memory
- Option 2
  - Split the web into multiple webs
- In either case, will retry the coloring

Which web to pick?

- One with interference degree >= N
- One with minimal spill cost (cost of placing value in memory rather than in register)
- What is spill cost?
  - Cost of extra load and store instructions

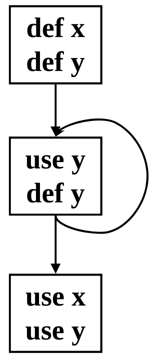
Ideal and Useful Spill Costs

- Ideal spill cost -dynamic cost of extra load and store instructions. Can’t expect to compute this.
  - Don’t know which way branches resolve
  - Don’t know how many times loops execute
  - Actual cost may be different for different executions
- Solution: Use a static approximation
  - profiling can give instruction execution frequencies or use heuristics based on structure of control flow graph

One Way to Compute Spill Cost

- Goal: give priority to values used in loops
- So assume loops execute 10 or 8 times
- Spill cost =
  - 所有def点存储指令的代价乘以10为底循环嵌套
  - 深度为幂的结果加上
  - sum over all use sites of cost of a load instruction times 10 to the loop nesting depth power
- Choose the web with the lowest spill cost

Spill Cost Example

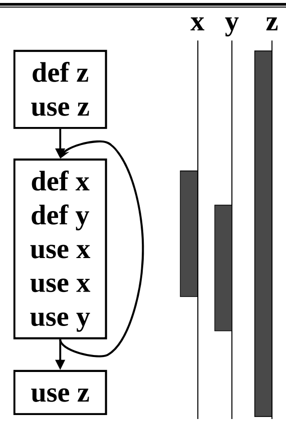


- **Spill Cost For x**
  - **storeCost +loadCost**
- **Spill Cost For y**
  - **9*storeCost +9*loadCost**
- **With 1 Register, Which Variable Gets Spilled?**

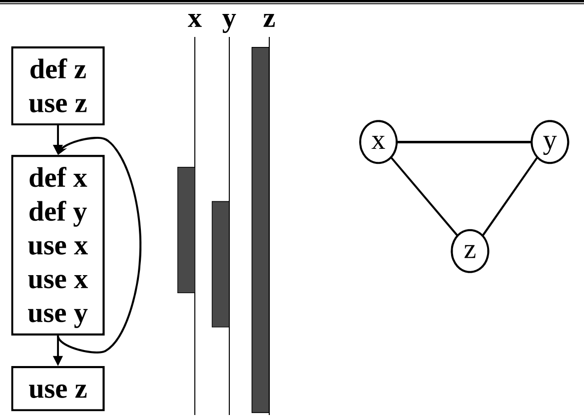
Splitting Rather Than Spilling

- Split the web
  - Split a web into multiple webs so that there will be less interference in the interference graph making it N-colorable
  - Spill the value to memory and load it back at the points where the web is split

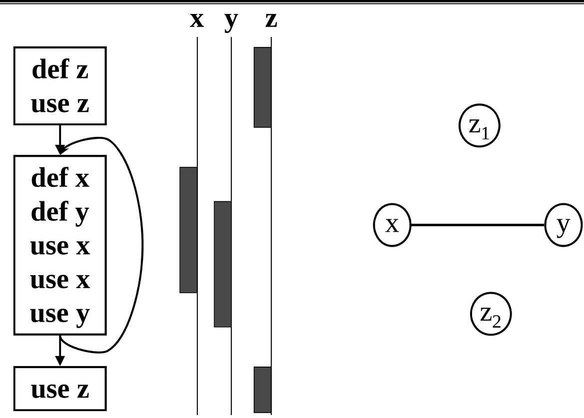
Splitting Example



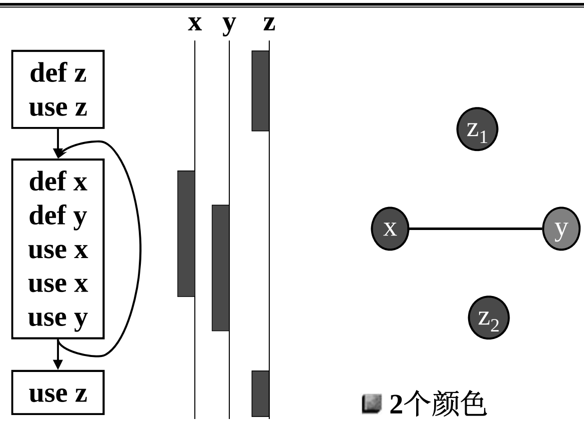
Splitting Example



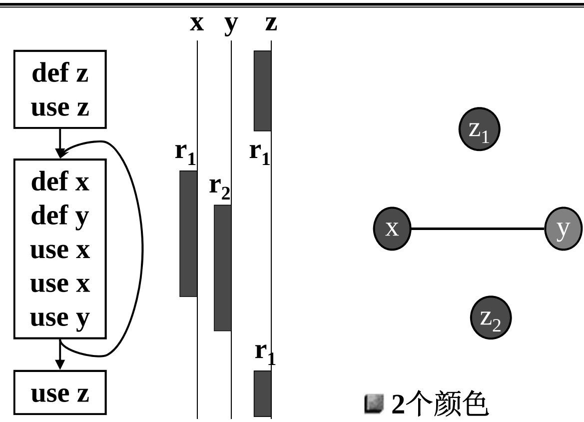
Splitting Example



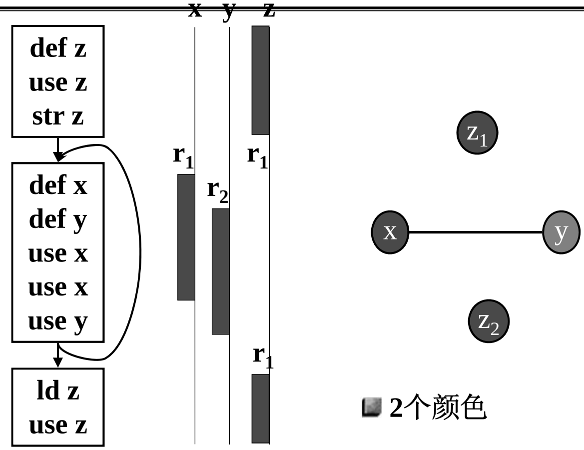
Splitting Example



Splitting Example



Splitting Example



- 2个颜色

分割启发式算法

- 确认一个没有R种图着色可能的程序点
  - 选择一个未在程序点最大嵌套范围内使用的web
  - 在相应弧处断开该web
  - 重做冲突图
  - 重新对图着色

Cost and benefit of splitting

- Cost of splitting a node
  - Proportion to number of times splitted edge has to be crossed dynamically
  - Estimate by its loop nesting
- Benefit
  - Increase colorability of the nodes the splitted web interferes with
  - Can approximate by its degree in the interference graph
- Greedy heuristic
  - pick the live-range with the highest benefit-to-cost ration to spill

寄存器目标（预着色）

- 在一定时间，一些变量需要存储在特殊的寄存器中
  - 一个函数首先六个实参
  - 返回值
- 对这些web进行预着色并且将他们固定在正确的寄存器中
- 可以消除不必要的复制指令

总结

- 寄存器分配
  - 在定义和使用间把值存储在寄存器中
  - 可以充分提高性能
- 关键概念
  - Web
  - 冲突图
  - 可着色
  - 分割

Control Flow Analysis

- CFG，有向图，结点表示基本块，弧表示控制流
- 每个结点有一个直接前驱结点集，有一个直接后继结点集

Data Flow Analysis

- 令
  - $in(b_i)$ 为到达基本块b_i时的活跃变量集
  - $out(b_i)$为离开基本块b_i时的活跃变量集
  - $def(b_i)$为在基本块b_i中定值（赋值）的变量集
  - $use(b_i)$为在基本块b_i中引用的变量集
- $def(b_i)$ 和 $use(b_i)$ 与 $in(b_i)$ 和 $out(b_i)$ 的关系如下:

$$in(b_i) = use(b_i) \cup (out(b_i) \setminus def(b_i))$$
$$out(b_i) = \bigcup_{x \in succ(b_i)} in(x)$$