

## 6.035 2005年秋

### 讲座16: 寄存器分配

## 在def和使用之间存储数值

- 程序使用这些值进行计算
  - 值定义(在何处计算)
  - 值使用(在何处被读出以计算新的值)
- 值必须在def和使用之间存储
  - 第一选项
    - 定义时在存储器中存储每个值
    - 在每次使用时从存储器中取回
  - 第二选项
    - 定义时在寄存器中存储每个值
    - 在每次使用时从寄存器中取回

## 寄存器分配

- 决定在数量有限的寄存器中存储哪些值
- 寄存器分配对性能有直接影响
  - 对每一句程序几乎都有影响
  - 清除很消耗存储空间的指令
  - 由于直接操纵寄存器,#指令直接传递(不需要导出和存储指令)
  - 可能为最有效的优化

## 在寄存器中可存储的是?

- 存储在编译器产生的临时数据值
- 语言级的值
  - 存储在本地标量变量的值
  - 大常数
  - 存储在数组元素和目标领域中的值
    - 事件: 别名分析
- 数据类型不同,寄存器类型不同
  - 浮点数存储在浮点数寄存器中
  - 整数和指针存储在整数寄存器中

## issues

- 使用寄存器时只能执行少量指令
  - 使用存储器执行其他指令
- 寄存器比存储器速度更快
  - 在更快更新的处理器中,差距更大
  - 4倍带宽和3倍延迟的因素
  - 如果程序特点不同,则可能放大这个特点
- 但可用寄存器的数量很少
  - 通常有16个整数和16个浮点寄存器
  - 其中一些寄存器有特定的使用者(例如:rsp,rbp)

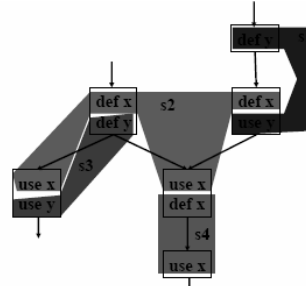
## 基于web的寄存器分配

- 确定每个值的活跃范围(网)
- 确定重叠范围(接口)
- 计算将每个web保存在一个寄存器中的收益(溢出代价)
- 确定哪个web分配一个寄存器(分配)
- 需要的话分割web(溢出和分割)
- 指派寄存器给web(指派)
- 生成包括溢出在内的代码(代码生成)

## webs

- 开始点：定义—使用链（DU链）
  - 将所用的定义连接到使用
- 把定值和使用放在同一网中的情况
  - 定义和所有可达的使用都必须在同一个web中
  - 所用达到同一个使用的定义必须在同一个web中
- 使用并查算法

## 示例



## Webs

- Web是寄存器分配单位
- 如果web对一个给定寄存器R进行了分配
  - 所有的定义都存储到R中
  - 所有的使用都从R中读取
- 如果web对一个存储器区域M进行了分配
  - 所有的定义都存储在M中
  - 素有的使用都从M中读取

## 凸集和活跃范围

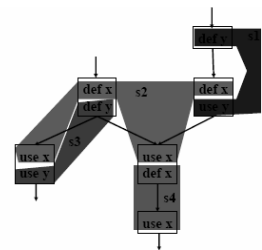
- 凸集的概念
- 在下面条件下，A使S为凸集
  - A,B包含在S中，并且C在B到B包含的路径上
  - C包含在S中
- Web的活跃范围概念
  - 包含web中的所有定义和使用指令的最小凸集
  - 直观上的，则是web实际值的区域

## 冲突

- 两个web，如果其活跃范围重叠的话，就会冲突（有一个非空的交集）
- 如果两个web冲突的话，则值必须存储在不同的寄存器和存储位置中
- 如果两个web不冲突，则值可以存储在相同的寄存器和存储位置中

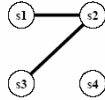
## 示例

web S1和S2互相冲突  
web S2和S3互相冲突



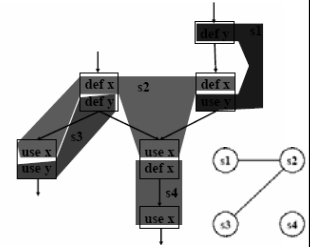
## 冲突图

- 是web的它们之间冲突的直观表现
  - 网点代表web
  - 如果两个web冲突的话, 则两者之间没有连线



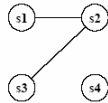
## 示例

- web S1和S2互相冲突
- web S2和S3互相冲突



## 通过图着色来分配寄存器

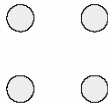
- 每个web分配一个寄存器
  - 每个网点分配一个寄存器 (一个颜色)
- 如果两个web冲突的话, 则他们不能使用同一个寄存器
  - 如果两个网点有连线的话, 则他们不可能使用同一种颜色



## 图着色

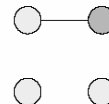
- 对图中的每个网点分配一种颜色
- 连接到同一边界的两个网点必须着不同的颜色
- 图理论中的经典问题
- NP完全问题
  - 但在寄存器分配中可用乐观启发式方法

## 图着色示例



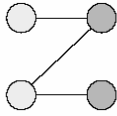
- 一种颜色

## 图着色示例



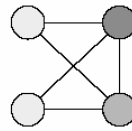
- 两种颜色

## 图着色示例



- 仍为两种颜色

## 图着色示例



- 3种颜色

## 寄存器着色的启发式方法

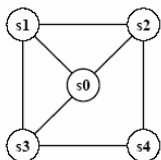
- 用 $N$ 种颜色进行图着色
- 如果度 $<N$  (网点的度=弧的数目)
  - 则网点总是可以着色的
  - 在对剩余的点着色完毕后, 必须至少还有一种颜色来对当前点进行着色
- 如果度 $\geq N$ 
  - 仍然有 $N$ 种颜色来进行着色

## 寄存器着色的启发式方法

- 删去度 $<N$ 的网点
  - 将其放入栈中
- 当所有的网点都 $\geq N$ 时
  - 找出可溢出的网点 (不对该网点着色)
  - 删除这个网点
- 当穷尽后, 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 $<N$ , 仍然存在这样一种颜色)

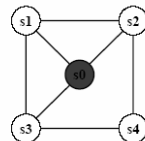
## 着色示例

$N = 3$  ■■■



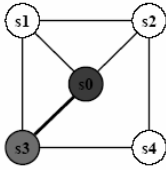
## 着色示例

$N = 3$  ■■■



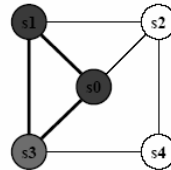
### 着色示例

N = 3 ■ ■ ■



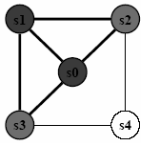
### 着色示例

N = 3 ■ ■ ■



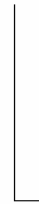
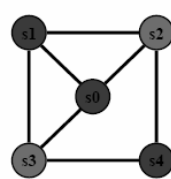
### 着色示例

N = 3 ■ ■ ■



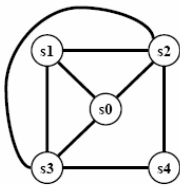
### 着色示例

N = 3 ■ ■ ■



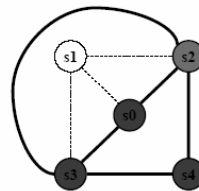
### 另一个着色示例

N = 3



### 另一个着色示例

N = 3 ■ ■ ■



## 现在应该做的

- 选项1
  - 选择一个web, 并且在存储器中指派值
  - 所有的定义都存储在存储器中, 所有的使用都从存储器中提取
- 选项2
  - 将一个web分割为多个web
- 在任一种选项中, 都要重新着色

## 如果选择web

- 冲突的度 $\geq N$
- 有最小的溢出代价 (存储在存储器而不是在寄存器中的代价)
- 什么是溢出代价?
  - 额外的载入和存储指令的代价

## 理想溢出代价和有用溢出代价

- 理想溢出代价-额外载入和存储指令的动态代价。但是不要期望达到这样的状态。
  - 不知道需要解决哪一个分支
  - 不知道循环需要执行多少次
  - 实际的代价因不同的执行方法而不同
- 解决方法: 使用一个静态的近似值
  - 断面图可以给出指令执行频率
  - 或者使用基于控制流图结构的启发式

## 计算溢出代价的方法

- 目标: 在循环中的值更有优先权
- 所以假定循环执行10到100次
- 溢出代价=
  - 所有def点存储指令的消耗乘以10为底循环嵌套深度为幂的结果加上
  - 所有def点载入指令的消耗乘以10为底循环嵌套深度为幂的结果加上
- 选择最小溢出代价的web

## 溢出代价示例



x的溢出代价  
存储代价+载入代价

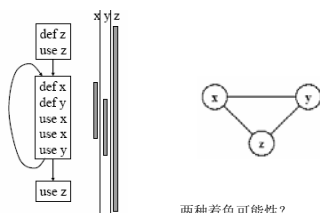
y的溢出代价  
 $9 \times$  存储代价 +  $9 \times$  载入代价

在只有一个寄存器的前提下,  
溢出哪个变量?

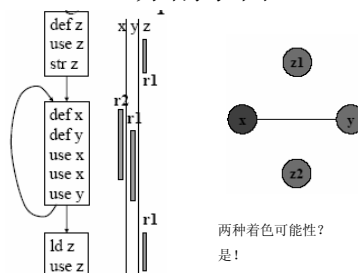
## 分割rather than溢出

- 分割web
  - 将一个web分割为多个web, 这样在冲突图中就会有较少的冲突, 就有N种着色可能
  - 将溢出的值存储在存储器中, 然后在web分割点上载回

## 分割示例



## 分割示例



## 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

## 分割的代价和益处

- 分割一个网点的代价
  - 分割边缘的次数比例必须动态跨越
  - 通过循环嵌套进行评估
- 益处
  - 增加与分割web冲突的网点的着色可能性
  - 可以通过冲突图中的度进行近似
- 贪婪启发式算法
  - 选出有最高获利/消耗溢出比率的活跃范围

## 进一步优化

- 寄存器合并
- 寄存器目标
- Web预分割
- 程序间寄存器分配

## 寄存器分配

- 找出寄存器复制指令如  $s_j = s_i$
- 如果  $s_i$  和  $s_j$  不相互冲突，则合并他们的web
- 支持意见:
  - 和复制传播相仿
  - 降低了指令数量
- 反对意见:
  - 可能增大合并网点的度
  - 可以着色的图可能变得不能着色

## 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

## Web的预分割

- 一些活跃范围有很大的“死”区域
  - 变量未被使用的大区域
- 分割活跃区域
  - 需要付出一些溢出代价
  - 但图会变得易于着色
- 可以找到分割的合适位置
  - 在调用点（无论如何都需要溢出）
  - 在一个大循环附近（为在循环内部使用的值预留寄存器）

## 程序间寄存器分配

- 越过程序边界来保存寄存器代价是昂贵的
  - 特别是对于有很多小函数的程序
- 调用惯例太过常规，同时没有效率
- 通过进行程序间寄存器分配来对每个函数定制个性化的调用惯例

## 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