五、目标代码生成 (17. 寄存器分配)

魏恒峰

hfwei@nju.edu.cn

2024 年 06 月 14 日

Register Allocation

The ***Register Allocation*** *problem* consists in mapping a program P_v, that can use an unbounded number of virtual registers, to a program P_p that contains a finite (possibly small) number of physical registers. Each target architecture has a different number of physical registers. If the number of physical registers is not enough to accommodate all the virtual registers, some of them will have to be mapped into memory. These virtuals are called *spilled virtuals*.

这也是一个组合优化问题

在同一个程序点上活跃的变量是有冲突的，不能分配到同一个寄存器。

- Target-independent code generation algorithms
  - Instruction Selection
    - Introduction to SelectionDAGs
    - SelectionDAG Instruction Selection Process
    - Initial SelectionDAG Construction
    - SelectionDAG LegalizeTypes Phase
    - SelectionDAG Legalize Phase
    - SelectionDAG Optimization Phase: the DAG Combiner
    - SelectionDAG Select Phase
    - SelectionDAG Scheduling and Formation Phase
    - Future directions for the SelectionDAG
  - SSA-based Machine Code Optimizations
  - Live Intervals
    - Live Variable Analysis
    - Live Intervals Analysis
  - Register Allocation
    - How registers are represented in LLVM
    - Mapping virtual registers to physical registers
    - Handling two address instructions
    - The SSA deconstruction phase
    - Instruction folding
    - Built in register allocators
  - Prolog/Epilog Code Insertion
  - Compact Unwind
  - Late Machine Code Optimizations
  - Code Emission
    - Emitting function stack size information
  - VLIW Packetizer
    - Mapping from instructions to functional units
    - How the packetization tables are generated and used

https://www.llvm.org/docs/CodeGenerator.html#register-allocator

Built in register allocators

The LLVM infrastructure provides the application developer with three different register allocators:

- *Fast* — This register allocator is the default for debug builds. It allocates registers on a basic block level, attempting to keep values in registers and reusing registers as appropriate.
- *Basic* — This is an incremental approach to register allocation. Live ranges are assigned to registers one at a time in an order that is driven by heuristics. Since code can be rewritten on-the-fly during allocation, this framework allows interesting allocators to be developed as extensions. It is not itself a production register allocator but is a potentially useful stand-alone mode for triaging bugs and as a performance baseline.
- *Greedy — The default allocator.* This is a highly tuned implementation of the *Basic* allocator that incorporates global live range splitting. This allocator works hard to minimize the cost of spill code.
- *PBQP* — A Partitioned Boolean Quadratic Programming (PBQP) based register allocator. This allocator works by constructing a PBQP problem representing the register allocation problem under consideration, solving this using a PBQP solver, and mapping the solution back to a register assignment.

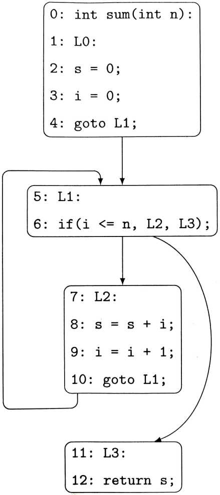
```
int sum(int n){
  int s = 0, i = 0;
  while(i <= n){
    s += i;
    i += 1;
  }
  return s;
}
```

```
int sum(int n):
  L0:
    s = 0;
    i = 0;
    goto L1;
  L1:
    if(i<=n, L2, L3);
  L2:
    s = s + i;
    i = i + 1;
    goto L1;
  L3:
    return s;
}
```

以非 SSA 形式的中间代码为例

问题 1: 变量 n, s, i 的活跃区间 (live interval) 分别是什么?

```
1  int sum(int n):
2    L0:
3      s = 0;
4      i = 0;
5      goto L1;
6    L1:
7      if(i<=n, L2, L3);
8    L2:
9      s = s + i;
10     i = i + 1;
11     goto L1;
12   L3:
13     return s;
14 }
```



问题 2: 在第 3 行后, 有哪些变量是活跃的?

Definition (活跃 (Live))

对于给定的变量 x, 考虑从其一个定义点 p 到使用点 q 的路径 l。

对于该路径 l 上的任意点 r, 如果 r 和 q 之间没有对变量 x 的其它定义,

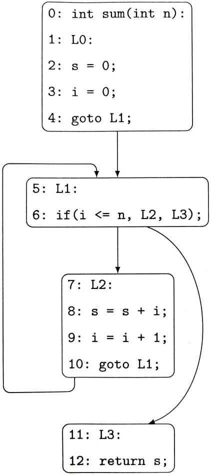
则称 x 在程序点 r 上是活跃的。

在同一个程序点上活跃的变量是有冲突的, 不能分配到同一个寄存器。

Definition (活跃分析 (Liveness Analysis))

分析变量的活跃点的程序分析被称为 活跃分析。

$\textsc{LiveIn}(s) : s$ 执行前的活跃变量集合

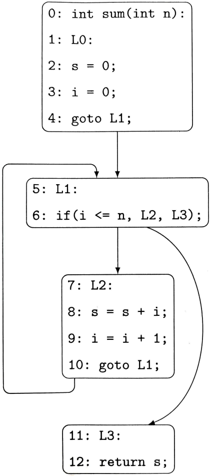


$$\textsc{LiveOut}(s) = \bigcup_{p \in succ(s)} \textsc{LiveIn}(p)$$

$$\textsc{LiveIn}(s) = (\textsc{LiveOut}(s) \setminus \mathsf{def}(s)) \cup \mathsf{use}(s)$$

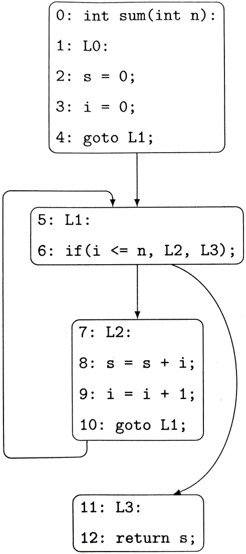
如何求解这个“数据流”方程组?

$\textsc{LiveOut}(s) : s$ 执行后的活跃变量集合

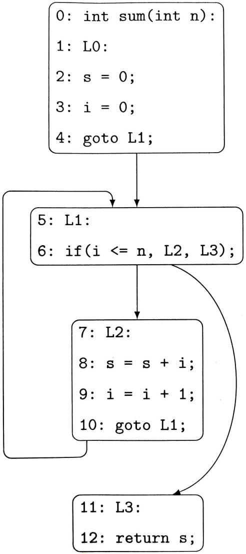


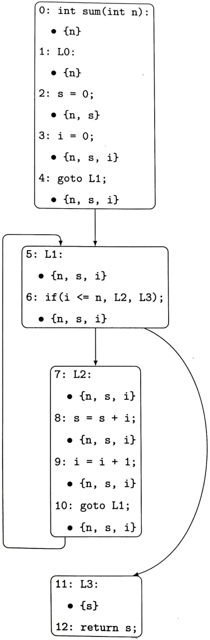

```
void liveness(program p){
  for(each statement s in p){
    liveIn[s] = ϕ;
    liveOut[s] = ϕ;
  }
  while(liveIn or liveOut set still changing){
    for(each statement s in p){
      liveOut[s] = ⋃_i liveIn(p_i); // p_i are successor of s
      liveIn[s] = (liveOut[s] - def(s)) ⋃ use(s);
    }
  }
}
```

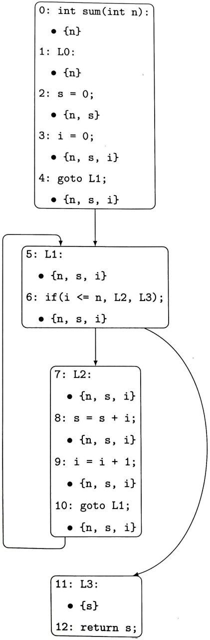
不动点算法



语句	use	def	out/in 初始值	out	in	out	in
2	$\varnothing$	s	$\varnothing$	n,s	n	…	…
3	$\varnothing$	i	$\varnothing$	i,n,s	n,s	…	…
6	i,n	$\varnothing$	$\varnothing$	i,s	i,n,s	…	…
8	i,s	s	$\varnothing$	i	i,s	…	…
9	i	i	$\varnothing$	$\varnothing$	i	…	…
12	s	$\varnothing$	$\varnothing$	$\varnothing$	s	…	…


0: int sum(int n):
1: L0:
2: s = 0;
3: i = 0;
4: goto L1;
5: L1:
6: if(i <= n, L2, L3);
7: L2:
8: s = s + i;
9: i = i + 1;
10: goto L1;
11: L3:
12: return s;


0: int sum(int n):
• {n}
1: L0:
• {n}
2: s = 0;
• {n, s}
3: i = 0;
• {n, s, i}
4: goto L1;
• {n, s, i}
5: L1:
• {n, s, i}
6: if(i <= n, L2, L3);
• {n, s, i}
7: L2:
• {n, s, i}
8: s = s + i;
• {n, s, i}
9: i = i + 1;
• {n, s, i}
10: goto L1;
• {n, s, i}
11: L3:
• {s}
12: return s;

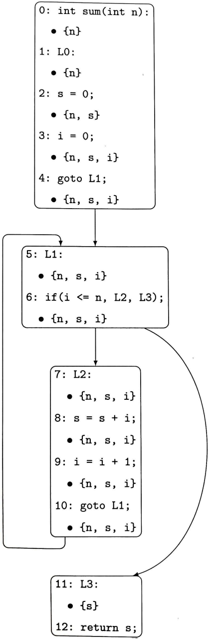


s.index : 语句 s 的行号

```
// Input: a list of basic blocks, which have been linearized
// for any variable x in the program, update the lower and upper bound
// of variable x in the map "interval"
void calculate_intervals(block blocks[]){
  for(each variable x in this program)
    for(each statement s in blocks)
      if(x ∈ liveOut(s)){ // x is live at statement s
        if(s.index < interval[x].l)
          interval[x].l = s.index;
        if(s.index > interval[x].h)
          interval[x].h = s.index;
      }
}
```

不计最后一次“使用” (use)

$n : [0, 10] \quad s : [2, 10] \quad i : [3, 11]$



$$n : [0, 10] \quad s : [2, 10] \quad i : [3, 11]$$

线性扫描分配算法 @ TOPLAS1999

Linear Scan Register Allocation

MASSIMILIANO POLETTO
Laboratory for Computer Science, MIT
and
VIVEK SARKAR
IBM Thomas J. Watson Research Center

三大关键操作: **占用、释放、溢出**

$$|R| = 3 \quad (R_1, R_2, R_3)$$

$x_1 : [2, 16]$

$x_2 : [2, 20]$

$x_3 : [7, 8]$

$x_4 : [9, 10]$

$x_5 : [11, 12]$

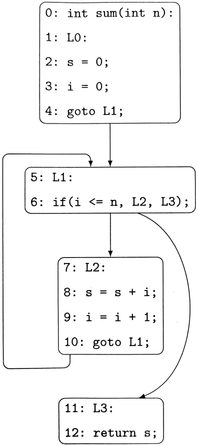
$x_6 : [15, 19]$

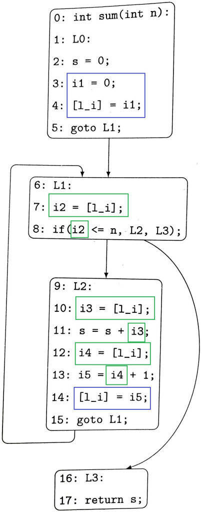
$x_7 : [17, 19]$

$$|R| = 2 \quad (R_1, R_2)$$

缺点: 引入了新的临时变量，需要进行迭代

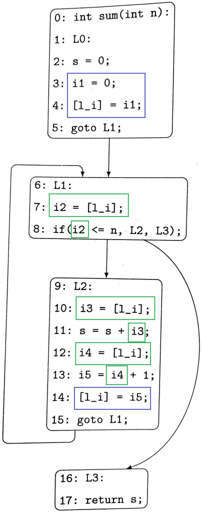
(稍感欣慰的是，新临时变量的活跃区间都很短)





溢出变量 i: 使用 store/load 存/取内存

解决方案一：生成代码时，使用临时物理寄存器实现临时变量



但是，物理寄存器本来就不够用

使用某寄存器前将其保存到内存中

```
addi sp, sp, -4
store x2, sp, 0
```

使用 x2 作为临时寄存器

```
load x2, sp, 0
addi sp, sp, 4
```

缺点：可能带来大量内存操作

解决方案二: 预留若干物理寄存器, 作为溢出处理时所需的临时寄存器

$$\cdots = x$$

$$\cdots$$

$$x = \cdots$$

$$r_i = \cdots$$

$$[l_x] = r_i$$

$$\cdots$$

$$r_i = [l_x]$$

$$\cdots = r_i$$

预留多少 (K) 个物理寄存器?

K 为程序中所有语句 def 集合或 use 集合的最大元素个数

(对于 RISC-V 典型程序, $K = 2$)

解决方案二: 预留若干物理寄存器, 作为溢出处理时所需的临时寄存器

$$y = \tau(x_1, x_2, \cdots, x_K)$$

假设 $x_1, \cdots, x_K, y$ 均发生溢出

$$r_1 = [l_{x_1}]$$

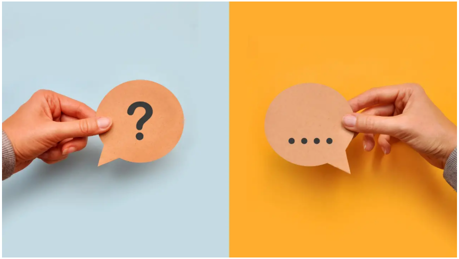
$$\ldots$$

$$r_K = [l_{x_K}]$$

$$r_1 = \tau(r_1, \cdots, r_K)$$

$$[l_y] = r_1$$

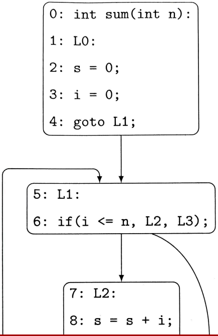
还有两个重要问题要解决

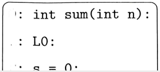


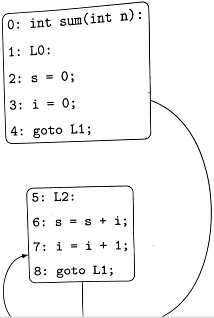
代码编号问题 (index)　　ϕ-指令消除问题

影响“活跃变量”分析结果吗？ (不影响)

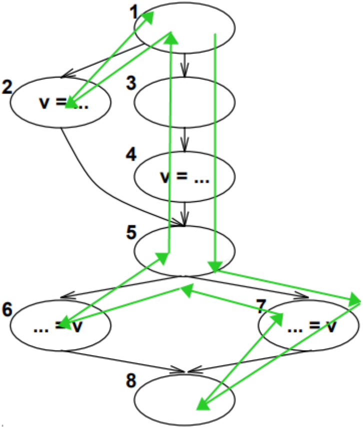
```
int sum(int n):
  L0:
    s = 0;
    i = 0;
    goto L1;
  L1:
    if(i<=n, L2, L3);
  L2:
    s = s + i;
    i = i + 1;
    goto L1;
  L3:
    return s;
}
```







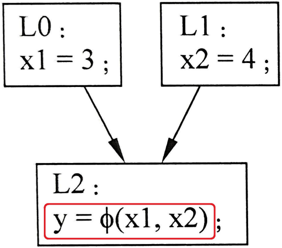
常用编号策略: 深度优先搜索顺序

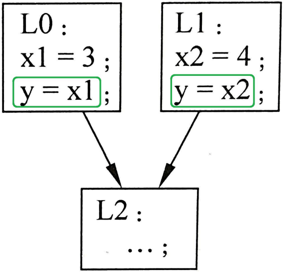


逆后序遍历序 (reverse post-order traversal ordering)

$$\phi$$

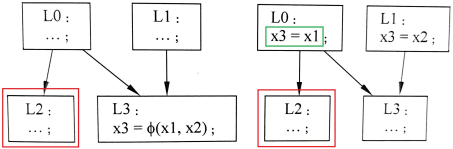
基本思想: 将“拷贝”操作上推到前驱基本块的末尾



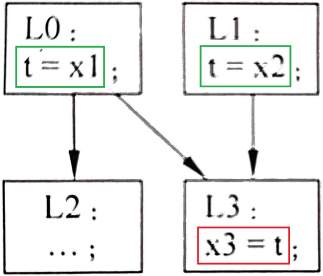


注意“循环”情况

问题一：备份丢失 (lost-copy) 问题

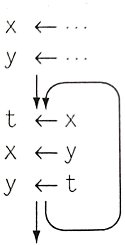


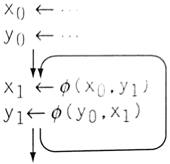
"关键路径"导致的备份丢失问题



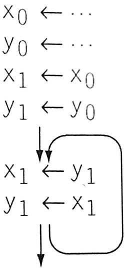
引入临时变量消除备份丢失

问题二: 交换 (swap) 问题





ϕ 指令的并行语义



交换问题

$t_x \leftarrow x_0$

$t_y \leftarrow y_0$

引入临时变量

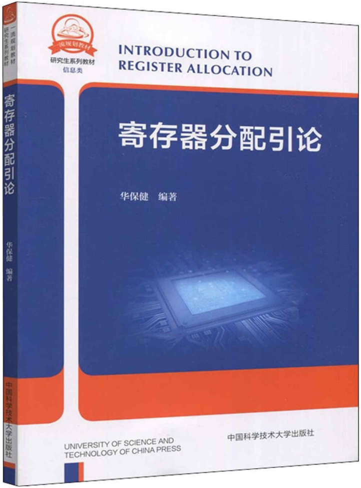
$x_1 \leftarrow t_x$

$y_1 \leftarrow t_y$

$\cdots$

$t_x \leftarrow y_1$

$t_y \leftarrow x_1$


研究生系列教材
信息类
INTRODUCTION TO
REGISTER ALLOCATION
寄存器分配引论
华保健 编著
UNIVERSITY OF SCIENCE AND
TECHNOLOGY OF CHINA PRESS
中国科学技术大学出版社

目 录

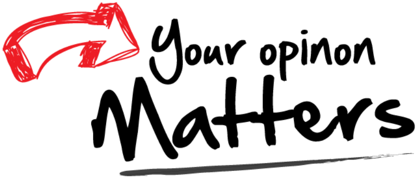
序一 ……………………………………………… i
序二 ……………………………………………… iii
前言 ……………………………………………… V
第 1 章 基础知识 ……………………………………………… 1
1.1 控制流图 ……………………………………………… 1
1.1.1 流图的定义 ……………………………………………… 1
1.1.2 流图的数据结构 ……………………………………………… 5
1.1.3 流图的构造 ……………………………………………… 9
1.2 活跃分析 ……………………………………………… 11
1.2.1 数据流方程 ……………………………………………… 12
1.2.2 不动点算法 ……………………………………………… 14
1.3 干涉图 ……………………………………………… 15
1.4 寄存器分配 ……………………………………………… 17
1.4.1 栈分配策略 ……………………………………………… 17
1.4.2 寄存器分配策略 ……………………………………………… 19
1.5 深入阅读 ……………………………………………… 22
第 2 章 图着色分配 ……………………………………………… 24
2.1 基本思想 ……………………………………………… 24
2.2 Kempe 算法 ……………………………………………… 26
2.2.1 Kempe 定理及其应用 ……………………………………………… 27
2.2.2 乐观着色 ……………………………………………… 29

2.3 溢出 ……………………………………………… 31
2.3.1 溢出着色 ……………………………………………… 33
2.3.2 溢出策略 ……………………………………………… 37
2.4 接合 ……………………………………………… 39
2.4.1 激进接合 ……………………………………………… 40
2.4.2 保守接合 ……………………………………………… 43
2.4.3 迭代接合 ……………………………………………… 48
2.5 干涉的保守性 ……………………………………………… 53
2.6 深入阅读 ……………………………………………… 54
第 3 章 线性扫描分配 ……………………………………………… 55
3.1 基本思想 ……………………………………………… 55
3.2 活跃区间分析 ……………………………………………… 57
3.2.1 线性序 ……………………………………………… 57
3.2.2 活跃区间算法 ……………………………………………… 61
3.3 线性扫描分配 ……………………………………………… 64
3.3.1 分配算法 ……………………………………………… 64
3.3.2 溢出和接合 ……………………………………………… 68
3.3.3 时间复杂度 ……………………………………………… 70
3.4 深入阅读 ……………………………………………… 71
第 4 章 弦图分配 ……………………………………………… 72
4.1 弦图 ……………………………………………… 72
4.2 弦图基本性质 ……………………………………………… 74
4.2.1 完美消去序列 ……………………………………………… 75
4.2.2 最大势算法 ……………………………………………… 78
4.3 弦图分配算法 ……………………………………………… 81
4.3.1 分配算法 ……………………………………………… 81
4.3.2 溢出 ……………………………………………… 83
4.3.3 接合 ……………………………………………… 84
4.3.4 时间复杂度 ……………………………………………… 86
4.4 深入阅读 ……………………………………………… 87

第 5 章 SSA 分配 …… 89
5.1 SSA 及其基本性质 …… 89
5.1.1 SSA 的性质与构造 …… 89
5.1.2 ϕ 语义 …… 92
5.1.3 SSA 消去 …… 93
5.2 SSA 上的活跃分析和干涉图 …… 97
5.2.1 活跃分析算法 …… 97
5.2.2 SSA 与弦图 …… 100
5.3 SSA 寄存器分配算法 …… 104
5.3.1 整体架构 …… 104
5.3.2 溢出 …… 105
5.3.3 着色 …… 109
5.3.4 ϕ 消去 …… 114
5.3.5 接合 …… 115
5.3.6 时间复杂度 …… 116
5.4 深入阅读 …… 116
第 6 章 线性规划分配 …… 118
6.1 整数线性规划基础 …… 118
6.1.1 线性规划的定义 …… 118
6.1.2 线性规划的求解 …… 120
6.1.3 问题求解的模型 …… 120
6.2 寄存器分配 …… 123
6.2.1 约束生成 …… 125
6.2.2 目标函数 …… 133
6.2.3 约束求解 …… 134
6.3 寄存器指派 …… 134
6.3.1 活跃区间切分 …… 134
6.3.2 指派算法 …… 137
6.3.3 时间复杂度 …… 138
6.4 深入阅读 …… 139

第 7 章 PBQP 分配 …… 14
7.1 二次分配问题基础 …… 14
7.1.1 二次分配问题 …… 140
7.1.2 PBQP …… 142
7.2 PBQP 寄存器分配模型 …… 144
7.2.1 问题求解的一般步骤 …… 144
7.2.2 寄存器分配的 PBQP 模型 …… 145
7.2.3 PBQP 图 …… 147
7.3 PBQP 寄存器分配算法 …… 149
7.3.1 PBQP 寄存器分配算法 …… 149
7.3.2 PBQP 求解算法 …… 150
7.3.3 时间复杂度 …… 163
7.4 深入阅读 …… 163
参考文献 …… 164


Thank
You!



Office 926

hfwei@nju.edu.cn