

## Register allocation

---

### Motivation

- registers much faster than memory
- limited number of physical registers
- keep values in registers as long as possible (minimize number of load/stores executed)

### Register allocation

1. for simplicity, assume an infinite set of virtual registers during optimization & code gen.
2. map virtual registers onto finite # of registers
3. assign virtual registers to physical registers

### Approaches

- local allocation      top-down — assign registers by frequency  
                                 bottom-up — spill registers by reuse distance
- global allocation      top-down — color interference graph  
                                 bottom-up — split live ranges

## Global register allocation (top-down)

---

### Register coloring

- map register allocation to graph coloring
- major steps
  1. global data-flow analysis to find *live ranges*
  2. build and color interference graph
  3. if unable to color, spill registers and repeat

## Local register allocation

---

### Top-down

- rank virtual registers by # uses in block
- reserve sufficient registers for memory operations
- assign remaining physical registers by rank
- problem – assignment fixed for entire block

### Bottom-up

- put physical registers on free list
- for each instruction in order
  - if operand not in register, assign free register
  - if free list empty, reclaim register holding value whose use is furthest in future

### Example

Top-down      Bottom-up

- 1)  $a = b + c$
- 2)  $c = a + d$
- 3)  $a = a + c$

## Live ranges

---

### Definition

- all definitions which reach a use, plus all uses reached by these definitions

```
      a =
b =
  = a      b =
           = b
           = b
```

- a single virtual register name may comprise several live ranges

```
  a =          b =
    = a
  a =          b =          = b
    = a          = b
```

Live ranges delineate when variables need to be stored in the same physical register to avoid extra code

## Interference

Using live ranges, an *interference graph* is constructed where

- vertices represent live ranges
- edges represent *interferences* between two live ranges, *i.e.*, both ranges are live at some point and cannot occupy the same register
- a coloring represents a register assignment (one color per register)

Note that the abstraction subtly changes our goals. However, the separation of optimization and allocation justifies the goal of minimal coloring.

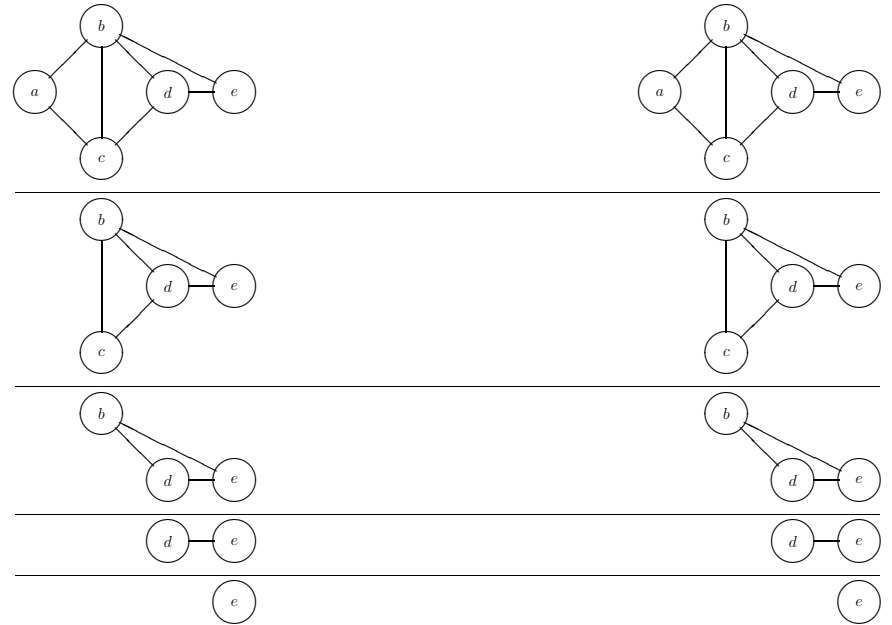
### Building the interference graph

- at each point  $p$  in the program, add edge  $(x, y)$  for all pairs of live ranges  $x, y$  live at  $p$

Example

a =	a =
= a	b =
b =	= a
= b	= b

## Simplification Example



## Coloring

### Graph coloring

- given graph, find assignment of colors to each node such that no neighbors have same color
- determining whether a graph may be colored with  $k$  colors is NP-hard for  $k > 2$

### Register coloring

- find a legal coloring given  $k$  colors,
- $k$  is the number of available registers

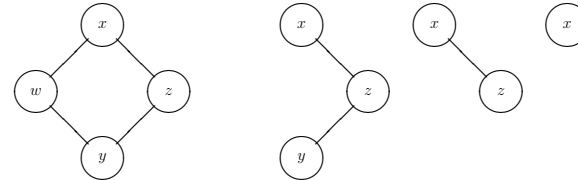
### Coloring algorithm

[Chaitin et al., 1981]

1. Repeatedly remove nodes with degree  $< k$  from the graph and push them on a stack.
2. If every remaining node is degree  $\geq k$ , spill node with lowest spill cost and remove it from the graph.
3. Reassemble the graph with nodes popped from the stack. As each node is added to the graph, choose a color differing from neighbors in the graph.

## Optimistic coloring

What if we have two registers? Coloring algorithm would spill a variable



### Optimistic coloring algorithm

[Briggs et al., 1989]

1. Repeatedly remove nodes with degree  $< k$  from the graph and push them on a stack.
2. If every remaining node is degree  $\geq k$ , select node with lowest spill cost and remove it from the graph.
3. Reassemble the graph with nodes popped from the stack. If node cannot be colored, spill it.

Deferring spill decisions helps when

- neighbors of a node are the same color
- a neighboring node has already been spilled

## Spilling

### Spill code

- when too few registers, *spill* register to memory
- insert load (before use) and store (after def)

```

a =      r1 =      r1 =
b =      r2 =      store r1, a
  = b      = r2      r1 =
  = a      = r1      = r1
                    load r1, a
                    = r1

```

### Effects

- breaks live range into many small live ranges
- reduces chance of interference
- very expensive, introduces load/store for each use/def over entire live range

## Allocation with Spilling

### Approach

- apply different cost estimate heuristics, pick best result
- most expense is in building interference graph
- spill cost estimates can be calculated efficiently

### Reducing spill code

- value modified – store register to memory (dirty)
- read-only value – reload from memory (clean)
- constant value – recompute value (rematerialize)

*Recognizing special cases can reduce need to spill*

## Spill Costs

### Which live ranges to spill? Two goals

- try to minimize cost of spill
- try to maximize decrease in interference (reduce need for more spills)

### Cost functions

$degree(v)$	# of edges for $v$ in interference graph
$depth(I)$	loop nesting depth of instruction $I$
$cost(v)$	$\sum_{v \text{ live at instr } I} 10^{\text{depth}(I)}$

### Cost estimate heuristics

- $cost(v) / degree(v)$
  - $cost(v) / degree(v)^2$
  - etc...
- [Chaitin et al.]

## Global register allocation (bottom-up)

### Live range splitting

- insert copies to split up live ranges
- hope to reduce spilling
- also controls spill code placement

```

a =      a =
        a = a
        = a      = a

```

### Coalescing (Subsumption)

- allocate source and destination of copy to same register to eliminate register-to-register copies
- combines live ranges
- can clean up unnecessary splits

```

a =      r1 =
b = a
  = b      = r1

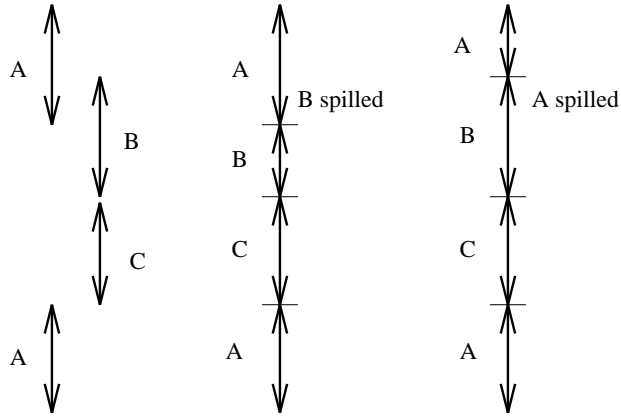
```

## Live range splitting

### One approach

[Chow & Hennessy 90]

1. locally allocate registers for each basic block
2. prioritize live ranges by estimated spill cost
3. allocate registers to live ranges
4. split live range if no colors available



## Combining Scheduling and Allocation

### Allocation before scheduling

- register assignment introduces dependences
- reduces freedom of scheduler
- example

```

load vr1,a      load r1,a      load r1,a
vr3 = vr1       r3 = r1        load r2,b
load vr2,b      load r1,b      r3 = r1
vr4 = vr2       r4 = r1        r4 = r2
    
```

## Enhancement examples

Original	Chaitin	Splitting	Rematerialized
$p \leftarrow f()$	$p \leftarrow f()$ <i>spill p</i>	$p \leftarrow f()$ <i>spill p</i>	
$y \leftarrow y + [p]$ ...regs...	<i>reload p</i> $y \leftarrow y + [p]$ ...regs...	<i>reload p</i> $y \leftarrow y + [p]$ ...regs...	$p \leftarrow f()$ $y \leftarrow y + [p]$ ...regs...
		<i>reload p</i>	$p \leftarrow f()$
$p \leftarrow p + 1$	<i>reload p</i> $p \leftarrow p + 1$ <i>spill p</i>	$p \leftarrow p + 1$	$p \leftarrow p + 1$

## Combining Scheduling and Allocation

### Scheduling before allocation

- lengthens live range of virtual registers
- increases register pressure, causes spills
- need to schedule spill code after allocation
- example

```

load vr1,a      load vr1,a
vr4 = vr1       load vr2,b
load vr2,b      load vr3,c
vr5 = vr2       vr4=vr1
load vr3,c      vr5=vr2
vr6 = vr3       vr6=vr3
    
```

## Combining Scheduling and Allocation

---

*We see that instruction scheduling and register allocation are interdependent. Some possible solutions to problem are:*

### Assigning registers

- first-fit — lowest number available register (reduces total number of registers assigned)
- round-robin — cycle through all registers (reduces memory-related dependences)

### Change ordering

- postpass – allocate then schedule
- prepass – schedule then allocate
- multipass – schedule, allocate, then schedule

### Integrated prepass scheduling

- schedule instructions first as prepass
- bias schedule to reduce local register pressure
- allocate registers after scheduling