Register Allocation

Michael O'Boyle

February, 2014

School of informatics

Course Structure

- L1 Introduction and Recap
- L2 Course Work
- L3+4 Scalar optimisation and dataflow
- L5 Code generation
- L6 Instruction scheduling
- **L7 Register allocation**
- Then high level approaches followed by adaptive compilation

Overview

- Local Allocation - spill code
- Clean and dirty spills
- Liveness analysis
- Global Allocation based on graph colouring
- Coalescing

Problem

- Registers are a finite resource. Sources and targets for many instructions on modern RISC like architectures.

- Code generation assumes an unbounded number of registers to simplify matters. Map unbounded number to the finite set.

- Key to this is knowing whether a value within a register is still needed. If not reuse it. If all values cannot be mapped to k registers - have to spill to memory - increasingly expensive

- In simplest case a NP-complete problem. Solutions characterised by scope and heuristics to reduce complexity. Assume code generation and scheduling unchangeable. Clearly a trade off between reg use and ILP - space and time.

Local allocation

- Focuses on basic block and maps virtual registers to physical registers
- Top-down allocation computes a priority with most important ones allocated a reg the others are spilled.
- Poor as virtual registers allocated a physical reg for the entire scope
- Bottom-up - iterates over block allocation on demand. Frees a register if it “knows” that no longer needed. Uses distance to next use as as spill metric.
- Spill clean values rather dirty as a way of minimising spill code

Spill code

1 registers: 2 values to manage

```

x =
y =

  = y
  = x
```

```
x =
Mem[spill] = x
y =

  = y
x = Mem[spill]
  = x
```

```
R1 =
store R1 -> R0   % Mem[R0]=R
R1 =

  = R1
load R0 -> R1    % R1 = Mem[
  = R1
```

Write spilled value to memory

Note still need R0 register for storage address.

Local allocation - spill code

2 registers: x1 clean in r1, x2 dirty in r2. Refer *x3*,x1,x2- must spill one:

```
load  x1 -> r1
load  x2 -> r2
add r2, 1 -> r2
      = x3
      = x1

      = x2
```

```
store r2 -> x2
load  x3 -> r2
      = r2 (use x3)
      = r1 (use x1)
load x2 -> r2
      = r2 (use x2)
Spill dirty
```

```
load x3 -> r1
      = r1 (use x3)
load x1 -> r1
      = r1 (use x1)
      = r2 (use x2)

Spill clean
```

Not always best sequence x3,x1,x3,x1,x2 - better to spill dirty values

Taking into account clean/dirty data makes it NP-complete

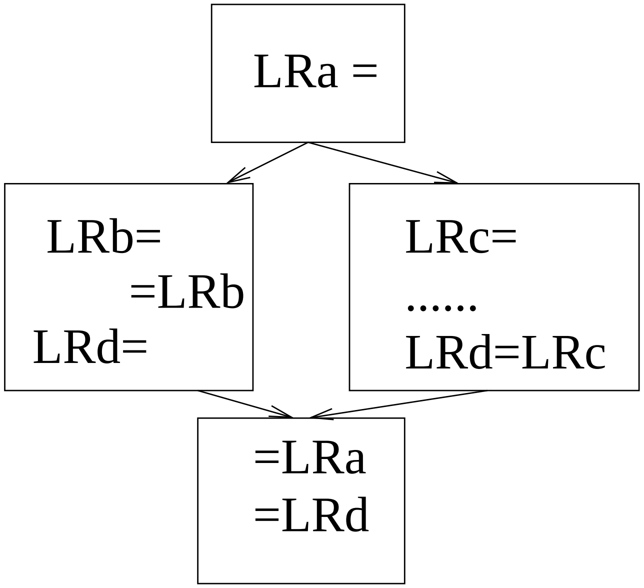
Beyond basic blocks - Liveness

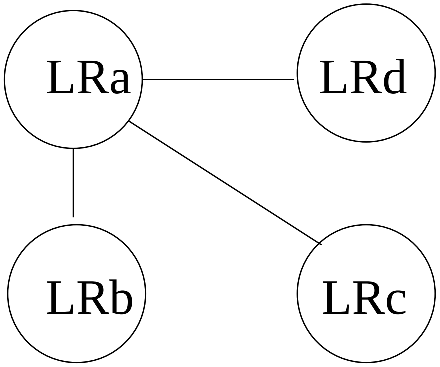
- Local allocation does not capture reuse of values across multiple blocks
- Must handle values defined in prev blocks and preserve values for later use
- Use live ranges allocate live range to register rather than variables or values A live range is from the definition to last use
- Perform live variable dataflow analysis to track live variables across blocks. $\text{LiveIn}(b) = \text{UEVar}(b) \cup (\text{LiveOut}(b) \cap \text{NotVarKill}(b))$
- Only values alive at a particular point need be allocated a register - used by local allocators too. Local approaches fail when tracking location of values and deciding on spill location

Global reg allocation

- Makes no distinction between local and global
- From live ranges construct an interference graph
- Colour interference graph so that no two nodes have same colour
- If graph needs more than k colours - decide on where to place spill code
- Colouring is NP-complete so we will need heuristics
- Map colours onto physical processors

Interference graph



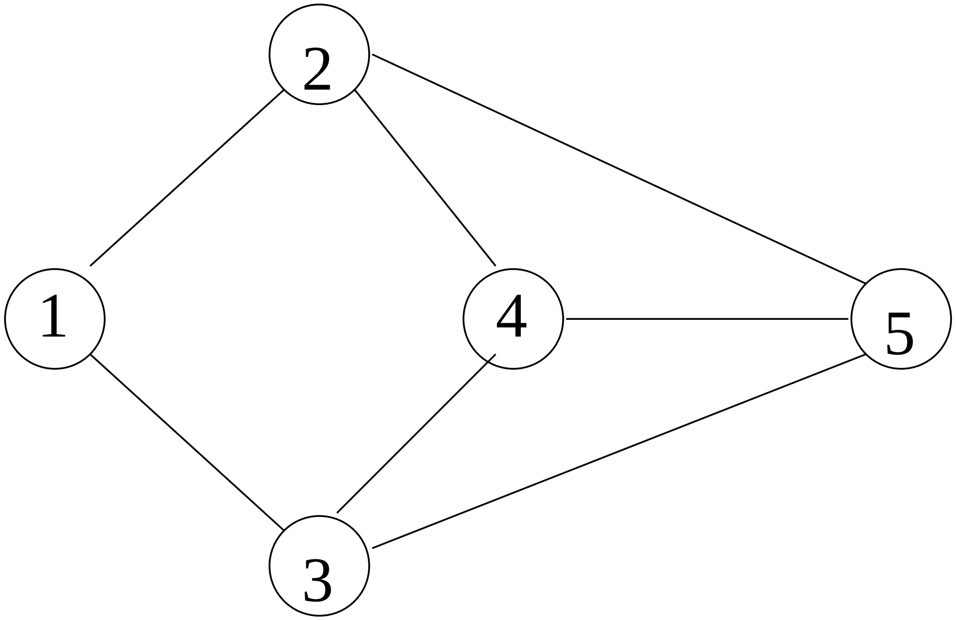


Live ranges interfere if one is live at the definition of another and have different values

Graph colouring

- Colour graph with k colours/registers
- Important observation any node n that has less than k neighbours $\mid$ n $\mid < k$ can always be coloured
- Pick any node $|n| < k$ and put on stack
- Remove that node and its edges - this reduces degree of neighbours
- Any remaining nodes - spill one and continue
- Pop nodes of stack and colour

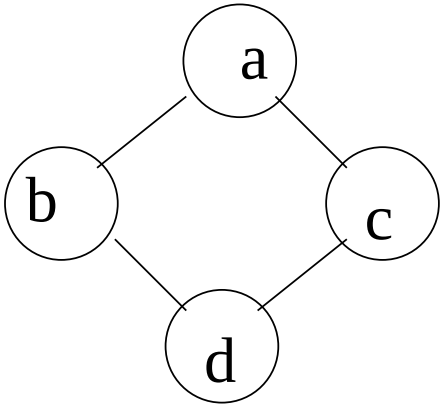
Graph colouring



5	g
4	r
3	b
2	b
1	g

3 colours. Remove 1 first as it has a degree less than 3. Colour as we pop

Graph colouring



d	1
c	2
a	1
b	maybe spill 2

2 colours - all have degree two. Default choose one and spill

If delay spilling can sometimes avoid it. This graph is 2 colourable

Spill candidates

- Minimise spill cost/ degree
- Spill cost is the loads and stores needed. Weighted by scope - ie avoid inner loops
- The higher the degree of a node to spill the greater the chance that it will help colouring
- Negative spill cost load and store to same mem location with no other uses
- Infinite cost - definition immediately followed by use. Spilling does not decrease live range

Alternative spilling

- Rather than spilling entire live ranges, spill only in high demand area -partial live ranges
- Splitting live ranges. Can reduce degree of interference graph. Smart splitting allows spilling to occur in “cheap” regions
- Coalesce - if two ranges don’t interfere and are connected by a copy - coalesce into one. Reduces degree of nodes that interfered with both

Coalescing

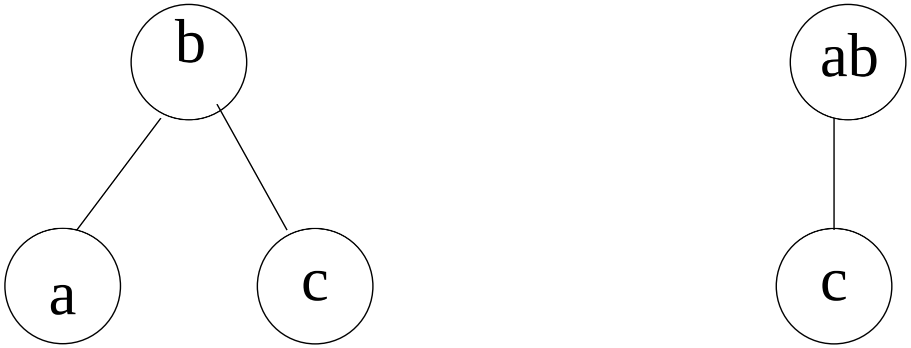
```
1:add LRt, LRu -> LRa
...
2:addI LRa, 0 ->LRb
3:xor  LRa, 0 -> LRc
...
4:add LRb, LRw -> LRx
5:add LRc, LRy -> LRz
```

```
1:add LRt, LRu -> LRab
...
...
3:xor LRab, 0 -> LRc
4:add LRab, LRw -> LRx
5:add LRc, LRy -> LRz
```

Live range of a [1..3], b[2...4], c [3..5] connected by 2 copies in 2,3.

Remove one copy here. Can also remove the other

Coalescing Reduces Degree

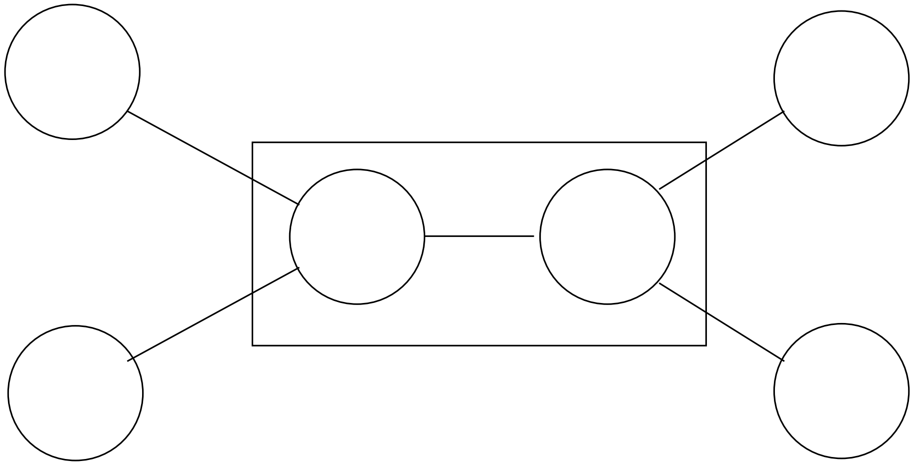


Guaranteed to not increase degree of interference on neighbours.

If a node interfered with both both before, coalescing helps

As it reduces degree, often applied before colouring takes place

Conservative Coalescing



Sometimes coalescing can increase the degree of the coalesced node and hence make colouring even more difficult

Conservative Coalescing $|LR_{ij}| < max(|LR_i|, |LR_j|)$

Iterative Coalescing: Conservative, Colour, Coalesce again...

Other approaches

- Top-down uses high level priorities to decide on colouring
- Hierarchical approaches - use control flow structure to guide allocation
- Exhaustive allocation - go through combinatorial options - very expensive but occasional improvement
- Rematerialisation - if easy to recreate a value do so rather than spill
- Passive splitting using a containment graph to make spills effective

Ongoing work

- Register allocation is a well studied topic. Linear scan for JITs
- Eisenbeis et al examining optimality of combined reg alloc and scheduling. Difficulty with general control-flow
- Partitioned register sets complicate matters. Allocation can require insertion of code which in turn affects allocation. Leupers investigated use of genetic algs for TM series partitioned reg sets.
- New work by Fabrice Rastello and others. Chordal graphs reduce complexity
- As latency increases see work in combined code generation, instruction scheduling and register allocation

Summary

- Local Allocation - spill code
- Liveness analysis
- Global Allocation based on graph colouring
- Bottom-up approaches
- Techniques to reduce spill code