

SWEN430 - Compilers

Lecture 19 - Register Allocation

David J. Pearce & Alex Potanin & Roma Klapaukh

*School of Engineering and Computer Science
Victoria University of Wellington*

What is Register Allocation?

```
r0 := r1 + r2
r4 := r0 + 1
r4 := r4 * 2
syscall
```

- Micro-processors have finite number of *registers*
- If we cannot allocate all program variables to registers, then must *spill* one or more to main memory.
- Fewer clock cycles required to fetch from register than cache, or main memory
- Reducing number of registers used increases performance

Register Spilling

- Typically, stack used to store temporary values which are accessed via *stack pointer*.
- Some registers better for spilling than others

```
r1 := ...           // array pointer
r2 := ...           // end pointer
r3 := ...           // sum
r4 := ...

.head
if r1 == r2 goto exit
r3 := r3 + *(r1)     // add value referred to by r1
r1 := r1 + 4         // 4 bytes per int
goto head
.exit
r4 := r4 + r3
```

- Which better to spill: r1,r2,r3 or r4?

Live Variables

Def-Use Path

A *def-use path* for a variable x is a path $v_1 \rightsquigarrow v_n$ through the Control-Flow Graph (CFG), where x is defined at v_1 and used at v_n , and where x is not defined on any node internal to that path (i.e. any node except v_1 and v_n).

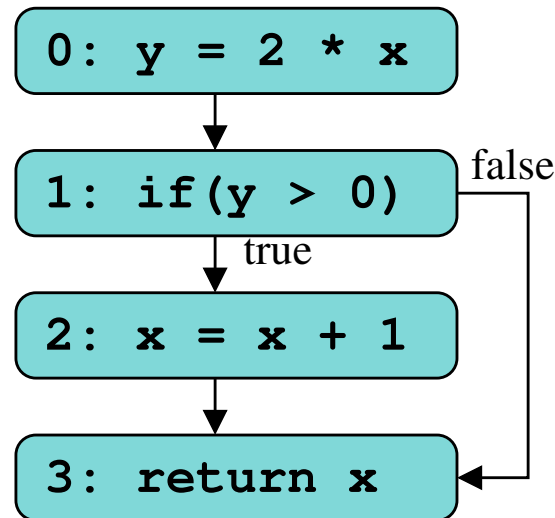
Live Variable

A variable x is *live* on an edge $v_1 \rightarrow v_2$ in a CFG iff $v_1 \rightarrow v_2$ is part of a def-use path for x .

- For simplicity, we assume parameters are “defined” at the first node.
- For a path $v_1 \rightsquigarrow v_n$, v_1 is *outgoing* and v_n is *incoming*; all internal nodes are both.

Live Variables (Cont'd)

- Example



Here, variable y is live on edge $0 \rightarrow 1$, but not anywhere else.
Variable x is live on all edges.

Def-use paths for x are: $\$ \rightarrow 0 \rightarrow 1 \rightarrow 2$, $\$ \rightarrow 0 \rightarrow 1 \rightarrow 3$ and $2 \rightarrow 3$.

(here, $\$$ represents virtual "entry" node)

Live Variables (cont'd)

Disjoint Def-Use Paths

Let x be a variable with def-use paths p_1, p_2 . Then, p_1 and p_2 are *disjoint* if they have no nodes of the same kind (i.e. incoming or outgoing) in common.

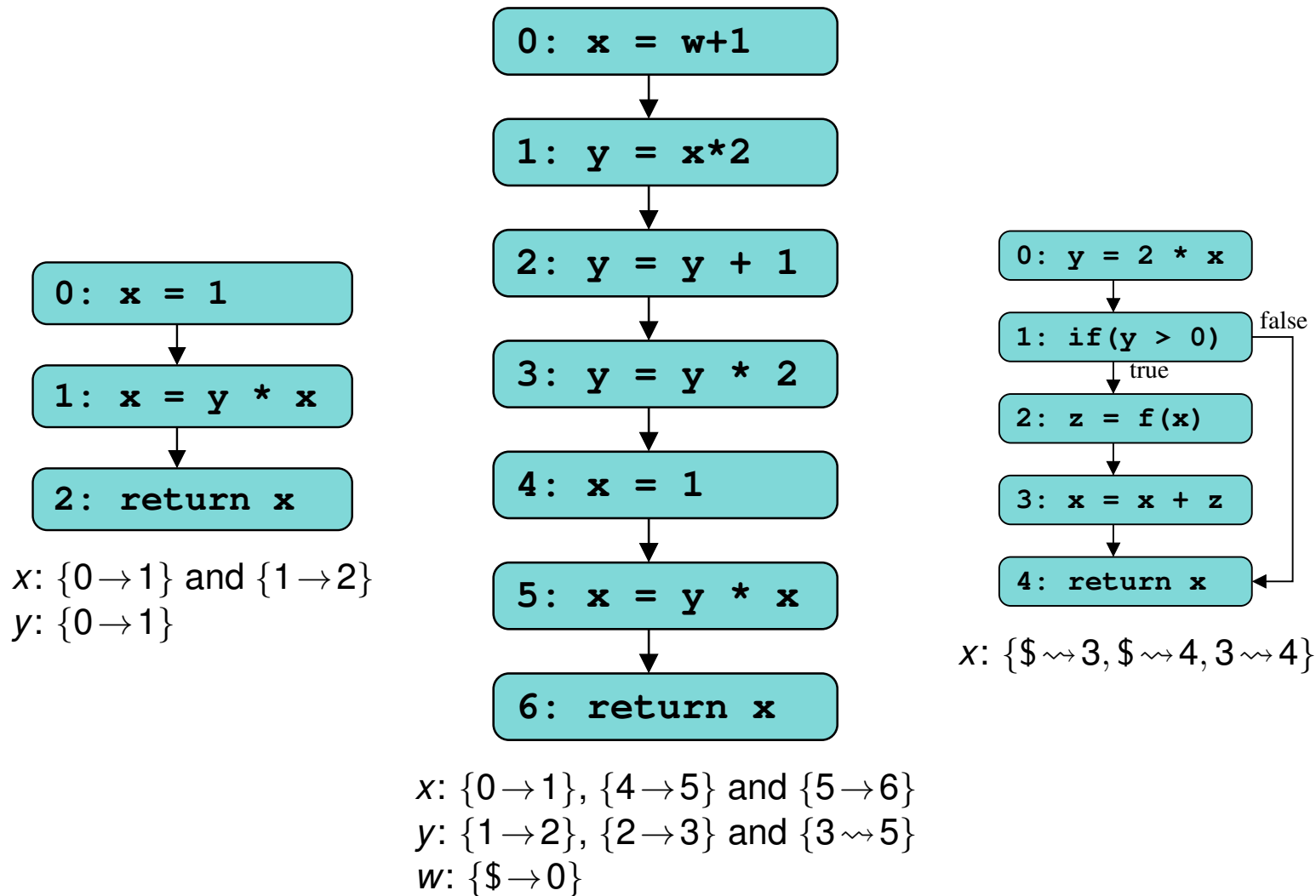
Live Range

A live range for a variable x is a maximal set, S , of def-use chains, such that S cannot be partitioned into two smaller sets S_1 and S_2 , where every element of S_1 is disjoint with every element of S_2 (and vice-versa).

- A variable may have more than one live range in a given method.
- This definition of live range is similar to that of a *def-use chain*.

Live Variables (cont'd)

- Examples



Interference

Live Range Interference

Live ranges R_1 and R_2 *interfere* if, for some def-use paths $p_1 \in R_1$ and $p_2 \in R_2$, it holds that p_1 and p_2 are not disjoint.

Interference Width

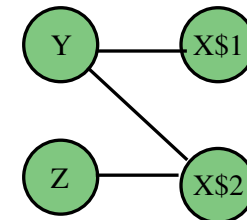
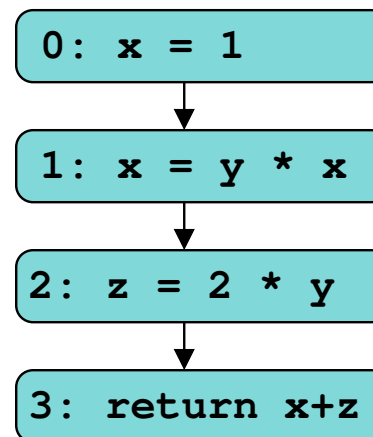
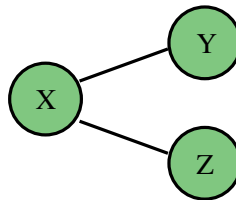
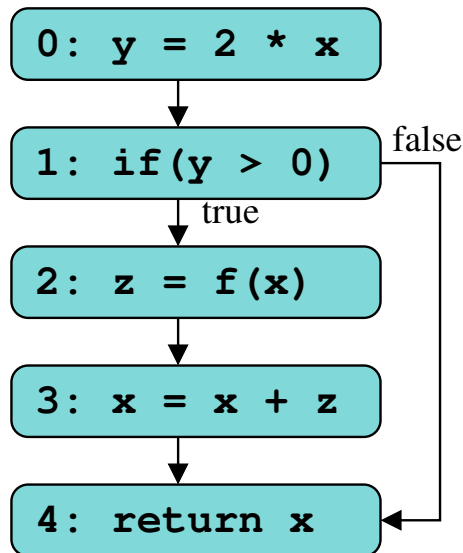
Consider any edge $v \rightarrow w$ in the CFG. The *interference width* for that edge is the number of live ranges (considering all variables) which have a def-use path containing it. The interference width for a method is the largest interference width of any edge.

- Let c be the interference width for a method M . Then, at most c registers are required for M .
- Can use fewer than c registers in some cases; in particular, if *instruction reordering* or *constant propagation* allowed.

Interference (cont'd)

Interference Graph

An *interference graph* is an undirected graph containing exactly one node per live range. An edge between two nodes exists iff their live ranges interfere.

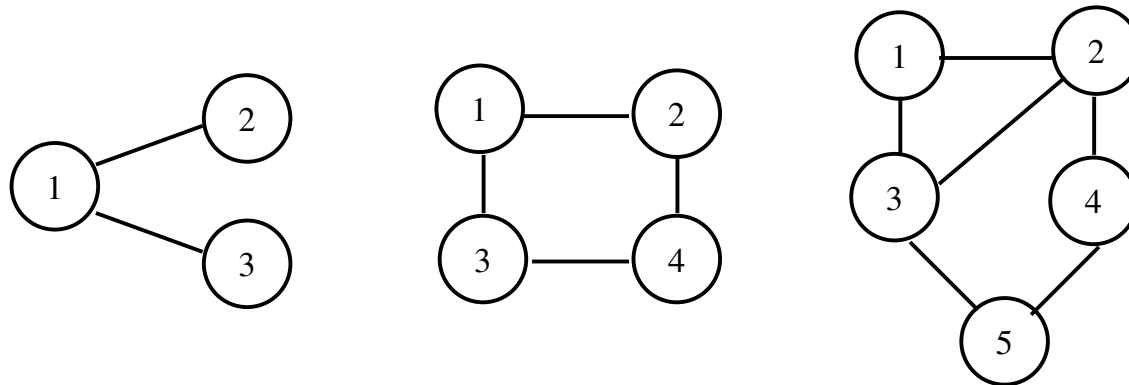


- Two nodes for x in second example, as two ranges

Graph Colouring

Colouring

An undirected graph is *coloured* iff every vertex can be given a colour such that, for each edge $x-y$, the colours of x and y differ.



- How many colours required for these graphs?

Graph Colouring (cont'd)

- Naive Algorithm for Graph Colouring:

```
{Var→Colour} allocate({Var} vars, {Colour} cols,
                      {Var→Color} assigned, Graph graph) {
    // base case
    if(vars.size() == 0 && isValid(assigned, graph) {
        return assigned;
    } else if(vars.size() == 0) {
        return null;
    }

    // recursive case
    Var var = RandomSelect(vars);
    {Var→Color} best = null;
    for(Color c : cols) {
        {Var→Color} r = allocate(vars-{var}, cols,
                                assigned + {var→c}, graph)
        if(isBetterThan(r, best)) { best = r; }
    }
    return best;
}
```

This could run for a long time! How can we do better?

Graph Colouring (cont'd)

- Assume Failure
 - Assume your algorithm will time-out and not complete search
 - Want to get good colourings quickly
 - Starting from highly-connected nodes helps
- Prune the Search Tree
 - Choose next node by traversing unvisited edges following BFS
 - Reduce range of colors for node by checking adjacent nodes which are already assigned
 - As soon as current assignment worse than best so far, stop it immediately
- Separate Components
 - Connected components can be explored separately
 - This reduces search space

Further Reading

- Books:

“Modern Compiler Design”, Andrew Appel (Chapters 10 + 11).

‘Engineering a Compiler”, Keith Cooper & Linda Torczon (pages 630–657).

‘Modern Compiler Design”, Steven Muchnick (Chapter 16)

- Papers:

“Linear scan register allocation”, M. Poletto and V.Sarkar. *ACM TOPLAS*, 21(5):895 - 913, 1999.

“The priority-based coloring approach to register allocation”, F.C. Chow and J.L. Hennessy. *ACM TOPLAS*, 12(4):501 - 536, 1990.

“Simple Register Spilling in a Retargetable Compiler”, C.W. Fraser and D.R. Hanson. *Software — Practice and Experience*, 22(1), 89–99, 1992.