COMPILERS
Liveness Analysis

hussein suleman

uct csc3003s 2009

Register Allocation

- IR trees are tiled to determine instructions, but registers are not assigned.
- Can we assign registers arbitrarily?
- What if:

```
mov d0,24
mov d1,36
add d0,d1
```

- was translated to:

```
mov eax,24
mov eax,36
add eax,eax
```

Allocation Issues

- Issues:
  - Registers already have previous values when they are used.
  - But there are a limited number of registers so we have to reuse!
  - Use of particular registers affects which instructions to choose.
  - Register vs. memory use affects the number and nature of LOAD/STORE instructions.
- Optimal allocation of registers is difficult.
  - NP-complete for k >1 registers

Liveness Analysis

- Problem:
  - IR contains an unbounded number of temporaries.
  - Actual machine has bounded number of registers.
- Approach:
  - Temporaries with disjoint live ranges (where their values are needed) can map to same register.
  - If not enough registers then spill some temporaries.
    - [i.e., keep them in memory]
- Liveness Analysis = determining when variables/registers hold values that may still be needed.

Control Flow Analysis

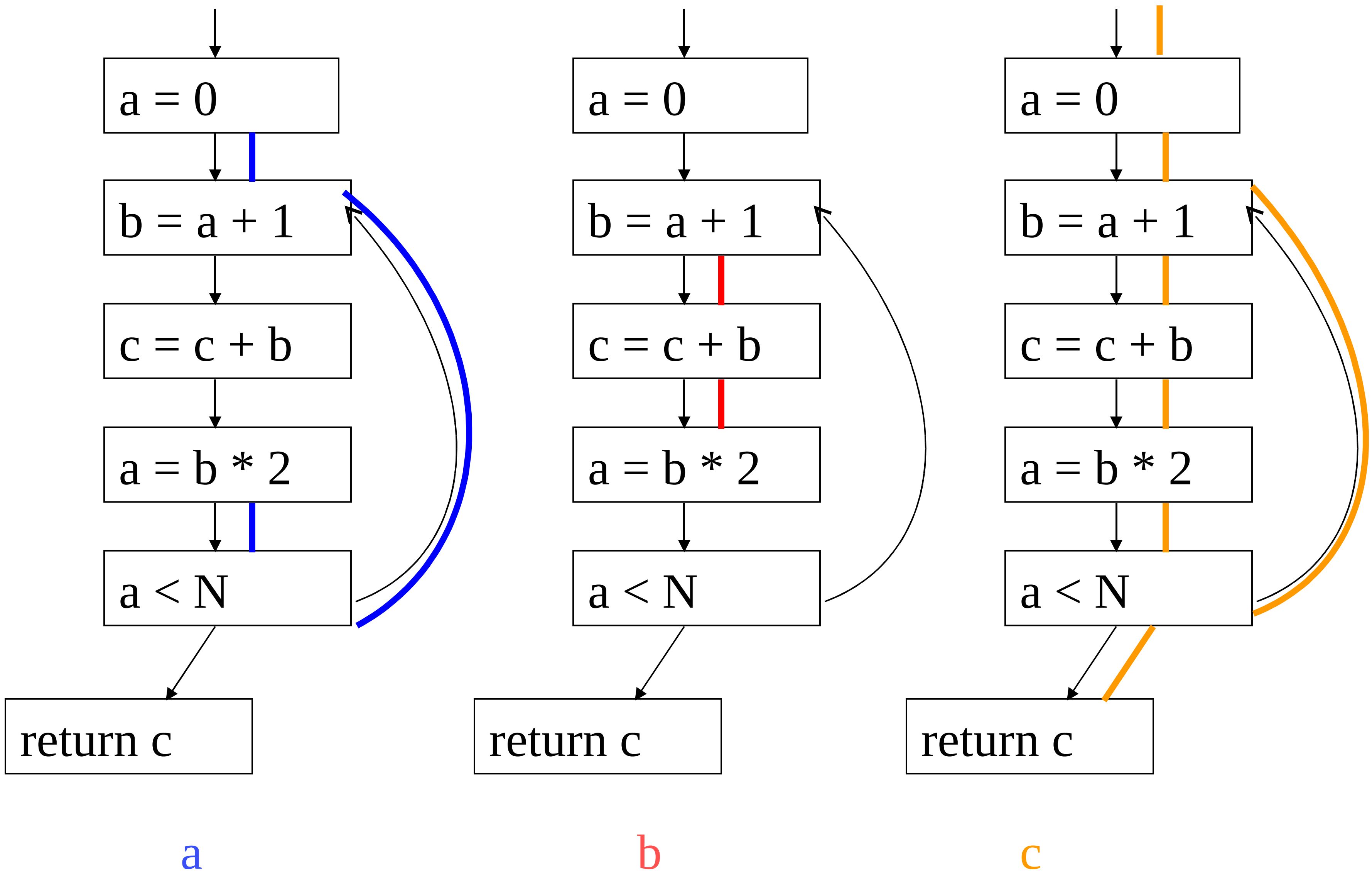
- Before performing liveness analysis, we need to understand the control flow by building a control flow graph [CFG]:
  - Nodes may be individual program statements or basic blocks.
  - Edges represent potential flow of control.
- Out-edges from node n lead to successor nodes, *succ[n]*.
- In-edges to node n come from predecessor nodes, *pred[n]*.

Control Flow Example 1/2

- Sample Program

```
            a = 0
L1:         b = a + 1
            c = c + b
            a = b * 2
            if a < N goto L1
            return c
```

Control Flow Example 2/2



def and use

- Gathering liveness information is a form of data flow analysis operating over the CFG.
- Liveness of variables "flows" around the edges of the graph.
  - assignments define a variable, v:
    - def[v] set of graph nodes that define v
    - def[n] set of variables defined by node n
  - occurrences of v in expressions use it:
    - use[v] = set of nodes that use v
    - use[n] = set of variables used in node n

Calculating Liveness 1/3

- v is *live* on edge e if there is a directed path from e to a use of v that does not pass through any def[v].
- v is *live-in* at node n if live on any of n's in-edges.
- v is *live-out* at n if live on any of n's out-edges.
- $v \in$ use[n] => v live-in at n
- v live-in at n => v live-out at all m $\in$ pred[n]
- v live-out at n, $v \notin$ def[n] => v live-in at n

Calculating Liveness 2/3

- Define:
  - in[n]: variables live-in at n
  - out[n]: variables live-out at n
- Then:
  - $\text{out}[n] = \bigcup_{s \in \text{succ}[n]} \text{in}[s]$
  - for a single node successor,
    - succ[n] = {} => out[n] = {}

Calculating Liveness 3/3

- Note:
  - $in[n] \supseteq use[n]$
  - $in[n] \supseteq out[n] - def[n]$
- use[n] and def[n] are constant [independent of control flow]
- Now,
- $v \in in[n]$ iff $v \in use[n]$ or $v \in out[n] - def[n]$
- Thus, $in[n] = use[n] \cup [out[n] - def[n]]$

Iterative Liveness Calculation

```
foreach n {in[n] = {}; out[n] = {}}
repeat
   foreach n
      in’[n]  = in[n];
      out’[n] = out[n];
      in[n]   = use[n] ∪ (out[n] - def[n])
      out[n]  =   ∪      in[s]
                s∈succ[n]
   until ∀n (in’[n] = in[n]) &
            (out’[n]  = out[n])
```

Liveness Algorithm Notes

- Should order computation of inner loop to follow the “flow”.
  - Liveness flows backward along control-flow arcs, from out to in.
- Nodes can just as easily be basic blocks to reduce CFG size.
- Could do one variable at a time, from uses back to defs, noting liveness along the way.

Liveness Algorithm Complexity 1/2

Complexity: for input program of size N

- ≤ N nodes in CFG
- => < N variables
- => N elements per in/out
- => O(N) time per set-union

for loop performs constant number of set operations per node

- => O(N^2) time for for loop

Liveness Algorithm Complexity 2/2

Each iteration of repeat loop can only add to each set.

Sets can contain at most every variable

=> sizes of all in and out sets sum to $2N^2$,

bounding the number of iterations of the repeat loop

=> worst-case complexity of $O(N^4)$

ordering can cut repeat loop down to 2-3 iterations

=> $O(N)$ or $O(N^2)$ in practice

Optimality 1/2

- Least fixed points
  - There is often more than one solution for a given dataflow problem (see example in text).
  - Any solution to dataflow equations is a conservative approximation:
    - v has some later use downstream from n,
      - => v $\in$ out(n)
      - but not the converse

- What is the implication of a non-least-fixed-point?

Optimality 2/2

- Conservatively assuming a variable is live does not break the program; just means more registers may be needed.
- Assuming a variable is dead when it is really live will break things.
- May be many possible solutions but want the “smallest”: the least fixed point.
- The iterative liveness computation computes this least fixed point.