CS3300 - Compiler Design

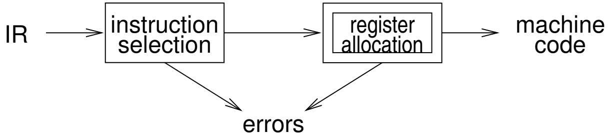
Liveness analysis and Register allocation

V. Krishna Nandivada

IIT Madras

Register allocation

Register allocation



Register allocation:

- have value in a register when used
- limited resources
- can effect the instruction choices
- can move loads and stores
- optimal allocation is difficult
  $\Rightarrow$ NP-complete for $k \geq 1$ registers

Liveness analysis

Problem:

- IR contains an unbounded number of temporaries
- machine has bounded number of registers

Approach:

- temporaries with disjoint live ranges can map to same register
- if not enough registers then spill some temporaries (i.e., keep them in memory)

The compiler must perform liveness analysis for each temporary:

It is live if it holds a value that may be needed in future

Example

$$
\begin{array}{rl}
 & a \leftarrow 0 \\
L_1: & b \leftarrow a+1 \\
 & c \leftarrow c+b \\
 & a \leftarrow b \times 2 \\
 & \text{if } a < N \text{ goto } L_1 \\
 & \text{return } c
\end{array}
$$

Liveness analysis

Gathering liveness information is a form of data flow analysis operating over the CFG:

- We will treat each statement as a different basic block.
- liveness of variables "flows" around the edges of the graph
- assignments define a variable, v:
  - $\mathit{def}(v) = \text{set of graph nodes that define } v$
  - $\mathit{def}[n] = \text{set of variables defined by } n$
- occurrences of v in expressions use it:
  - $\mathit{use}(v) = \text{set of nodes that use } v$
  - $\mathit{use}[n] = \text{set of variables used in } n$

Definitions

- v is <u>live</u> on edge e if there is a directed path from e to a <u>use</u> of v that does not pass through any *def*(v)
- v is <u>live-in</u> at node n if live on any of n's in-edges
- v is <u>live-out</u> at n if live on any of n's out-edges
- $v \in \mathit{use}[n] \Rightarrow v \text{ live-in at } n$
- (For programs with statically established no uninitialized variables) $v \text{ live-in at } n \Rightarrow v \text{ live-out at all } m \in \mathit{pred}[n]$
- $v \text{ live-out at } n, v \notin \mathit{def}[n] \Rightarrow v \text{ live-in at } n$

Liveness analysis

Define:

$$\begin{aligned} \mathit{in}[n] &= \text{variables live-in at } n \\ \mathit{out}[n] &= \text{variables live-out at } n \end{aligned}$$

Then:

$$\mathit{out}[n] = \bigcup_{s \in \mathit{succ}(n)} \mathit{in}[s]$$

$$\mathit{succ}[n] = \phi \Rightarrow \mathit{out}[n] = \phi$$

Note:

$$\begin{aligned} \mathit{in}[n] &\supseteq \mathit{use}[n] \\ \mathit{in}[n] &\supseteq \mathit{out}[n] - \mathit{def}[n] \end{aligned}$$

$\mathit{use}[n]$ and $\mathit{def}[n]$ are constant (independent of control flow)
Now, $v \in \mathit{in}[n]$ iff. $v \in \mathit{use}[n]$ or $v \in \mathit{out}[n] - \mathit{def}[n]$
Thus, $\mathit{in}[n] = \mathit{use}[n] \cup (\mathit{out}[n] - \mathit{def}[n])$

Iterative solution for liveness

N : Set of nodes of CFG;

foreach $\underline{n \in N}$ **do**

  $in[n] \leftarrow \phi$;

  $out[n] \leftarrow \phi$;

end

repeat

  foreach <u>n ∈ Nodes</u> **do**

    $in'[n] \leftarrow in[n]$;

    $out'[n] \leftarrow out[n]$;

    $in[n] \leftarrow use[n] \cup (out[n] - def[n])$;

    $out[n] \leftarrow \bigcup_{s \in succ[n]} in[s]$;

  end

until $\underline{\forall n, in'[n] = in[n] \wedge out'[n] = out[n]}$;

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

Iterative solution for liveness

Complexity: for input program of size N

- $\leq N$ nodes in CFG
  $\Rightarrow \leq N$ variables
  $\Rightarrow N$ elements per *in/out*
  $\Rightarrow \mathrm{O}(N)$ time per set-union
- **for** loop performs constant number of set operations per node
  $\Rightarrow \mathrm{O}(N^2)$ time for **for** loop
- each iteration of **repeat** loop can only add to each set
  sets can contain at most every variable
  $\Rightarrow$ sizes of all in and out sets sum to $2N^2$,
  bounding the number of iterations of the **repeat** loop

$\Rightarrow$ worst-case complexity of $\mathrm{O}(N^4)$

- ordering can cut **repeat** loop down to 2-3 iterations
  $\Rightarrow \mathrm{O}(N)$ or $\mathrm{O}(N^2)$ in practice

Least fixed points

There is often more than one solution for a given dataflow problem (see example).

Any solution to dataflow equations is a conservative approximation:

- v has some later use downstream from n
  $\Rightarrow v \in \mathit{out}(n)$
- but not the converse

Conservatively assuming a variable is live does not break the program; just means more registers may be needed.

Assuming a variable is dead when really live will break things.

Many possible solutions but we want the "smallest": the least fixpoint.

The iterative algorithm computes this least fixpoint.

Register allocation - by Graph coloring

- Step 1:
  - Select target machine instructions assuming infinite registers (temps).
  - If a instruction requires a special register – replace that temp with that register.
- Step 2:
  - Construct an interference graph.
  - Solve the register allocation problem by coloring the graph.
  - A graph is said to be <u>colored</u> if each each pair of neighboring nodes have different colors.

Graph coloring - a simplistic approach

Input: G - the interference graph, K - number of colors

repeat

    repeat

        Remove a node n and all its edges from G, such that degree of n is less than K;

        Push n onto a stack;

    until <u>G has no node with degree less than K</u>;

    `// G is either empty or all of its nodes have degree ≥ K`

    if <u>G is not empty</u> **then**

        Take one node m out of G, and mark it for spilling;

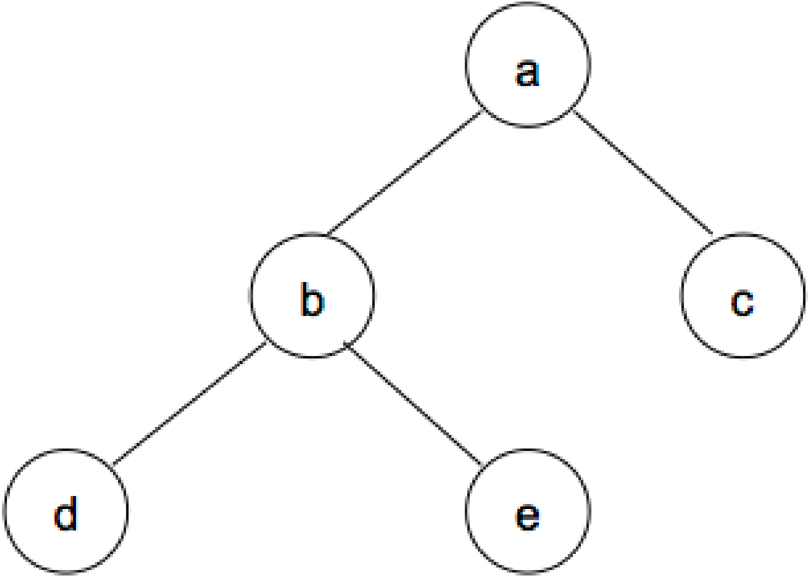
        Remove all the edges of m from G;

    end

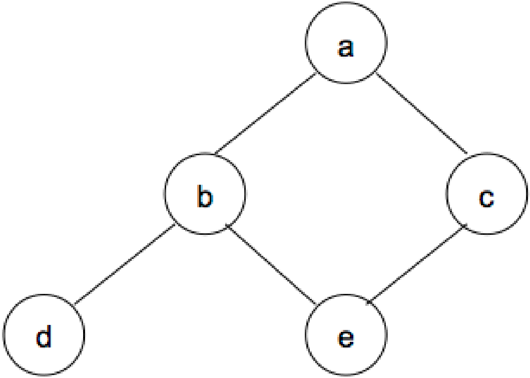
until <u>G is empty</u>;

Take one node at a time from the stack and assign a <u>non conflicting</u> color.

Example 1, available colors = 2



Example 2



We have to spill.

Graph coloring - Kempe's heuristic

- Algorithm dating back to 1879.

Input: G - the interference graph, K - number of colors

repeat

  repeat

    Remove a node n and all its edges from G, such that degree of n is less than K;

    Push n onto a stack;

  until G has no node with degree less than K;

  `// G is either empty or all of its nodes have degree ≥ K`

  if G is not empty **then**

    Take one node m out of G.;

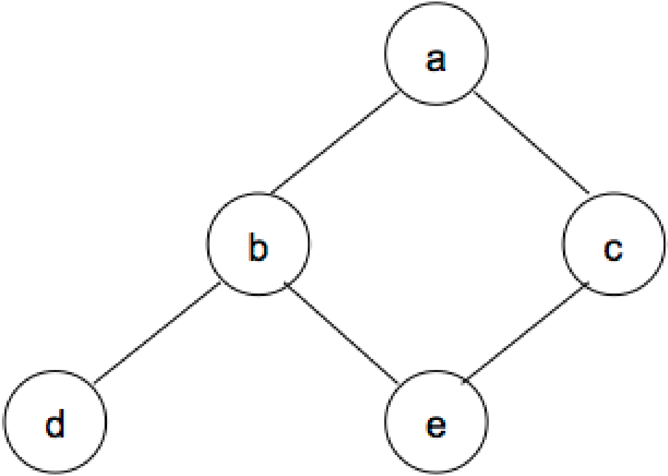
    push m onto the stack;

  end

until G is empty;

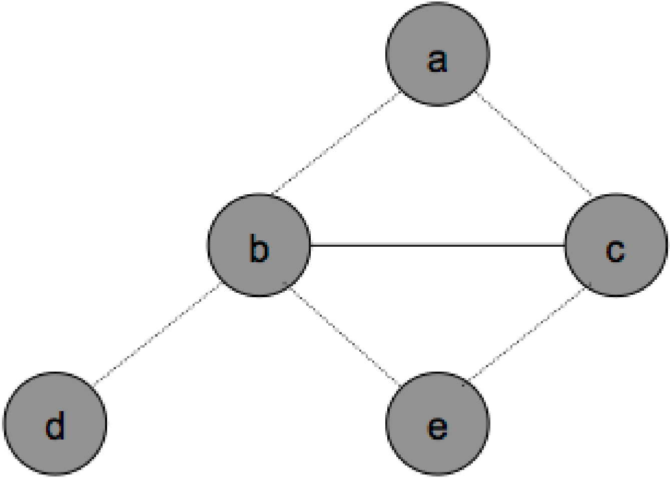
Take one node at a time from the stack and assign a non conflicting color (if possible, else spill).

Example 2 (revisited)



We don't have to spill.

Example 3



Don't have a choice. Have to spill.

Register allocation - Linear scan

Register allocation is **expensive**.

- Many algorithms use heuristics for graph coloring.
- Allocation may take time quadratic in the number of live intervals.

Not suitable

- Online compilers – need to generate code quickly. e.g. JIT compilers.
- Sacrifice efficient register allocation for compilation speed.

Linear scan register allocation - Massimiliano Poletto and Vivek Sarkar, ACM TOPLAS 1999

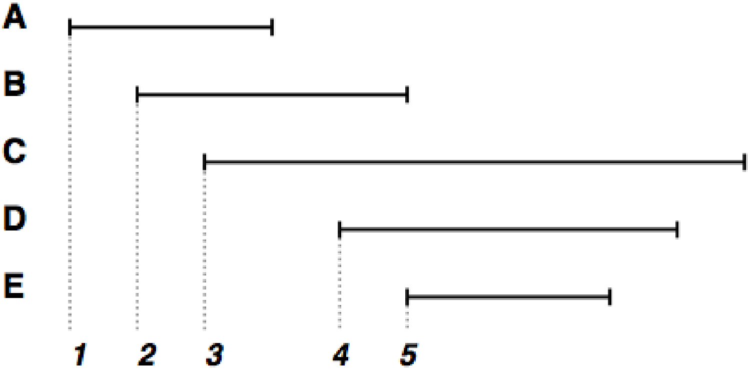
Linear Scan algorithm

```
LINEARSCANREGISTERALLOCATION
    active ← {}
    foreach live interval i, in order of increasing start point
        EXPIREOLDINTERVALS(i)
        if length(active) = R then
            SPILLATINTERVAL(i)
        else
            register[i] ← a register removed from pool of free registers
            add i to active, sorted by increasing end point

EXPIREOLDINTERVALS(i)
    foreach interval j in active, in order of increasing end point
        if endpoint[j] ≥ startpoint[i] then
            return
        remove j from active
        add register[j] to pool of free registers

SPILLATINTERVAL(i)
    spill ← last interval in active
    if endpoint[spill] > endpoint[i] then
        register[i] ← register[spill]
        location[spill] ← new stack location
        remove spill from active
        add i to active, sorted by increasing end point
    else
        location[i] ← new stack location
```

Example



- Say, available registers = 2

Linear Scan algorithm - analysis

- Each live range gets either a register or a spill location.
- Note: The number of overlapping intervals changes only at the start and end points of an interval.
- Live intervals are stored in a list that is sorted in order of increasing start point.
- The active list is kept sorted in order of increasing end point. Adv: need to scan only those intervals (+1 at most) that have to be removed.
- Complexity: $O(V)$ – if number of registers is assumed ot be a constant. Else? $O(V \times logR)$

Spilling

- We need to generate extra instructions to load variables from the stack and store them back.
- The load and store may require registers again:
  - Naive approach: Keep a separate register (wasteful).
  - Rewrite the code - by introducing a temporary; rerun the liveness + ra.
    (Note: the new temp has much smaller live range).

Example: rewrite code

Consider: `add t1 t2`

- Suppose `t2` has to be spilled, say to `[sp-4]`.
- Invent a new temp `t35`, and rewrite:

```
mov t35 [sp-4]
add t1 t35
```

- `t35` has a very short live range and less likely to interfere.
- Now rerun the algo.

Criteria for spilling

During register allocation, we identify that one of the live ranges from a given set, has to be spilled. Criteria?

- Random! Adv? Disadv?
- One with maximum degree
- One that has the longest life
- One with the shortest life (take advantage of the cache).
- One with least cost.
  - Cost = Dynamic (load cost + store cost)
  - How to handle loops, conditionals?
  - Cost of load, store