CS3300 - Compiler Design

Liveness analysis and Register allocation

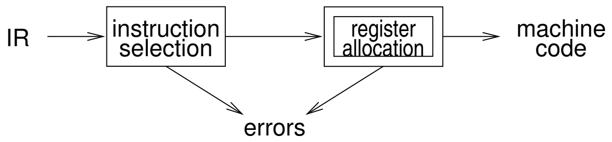
V. Krishna Nandivada

IIT Madras

Register allocation

Copyright © 2023 by Antony L. Hosking. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and full citation on the first page. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or fee. Request permission to publish from hosking@cs.purdue.edu.

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

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

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:
  - $def(v)$ = set of graph nodes that define v
  - $def[n]$ = set of variables defined by n
- occurrences of v in expressions use it:
  - $use(v)$ = set of nodes that use v
  - $use[n]$ = set of variables used in n

Definitions

- 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
- $v \in use[n] \Rightarrow v$ live-in at n
- (For programs with statically established no uninitialized variables) v live-in at $n \Rightarrow v$ live-out at all $m \in pred[n]$
- v live-out at $n, v \notin def[n] \Rightarrow v$ live-in at n

Liveness analysis

Define:

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

Then:

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

$$
succ[n] = \phi \Rightarrow out[n] = \phi
$$

Note:

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

$use[n]$ 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 solution for liveness

N : Set of nodes of CFG;
foreach $n \in N$ **do**
  $in[n] \leftarrow \phi$;
  $out[n] \leftarrow \phi$;
end
repeat
  foreach n ∈ Nodes **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 $\forall n, in'[n] = in[n] \wedge out'[n] = out[n]$;

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*
  $\Rightarrow O(N)$ time per set-union
- **for** loop performs constant number of set operations per node
  $\Rightarrow O(N^2)$ 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 $O(N^4)$

- ordering can cut **repeat** loop down to 2-3 iterations
  $\Rightarrow O(N)$ or $O(N^2)$ 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
  $\Rightarrow v \in out(n)$
- 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.

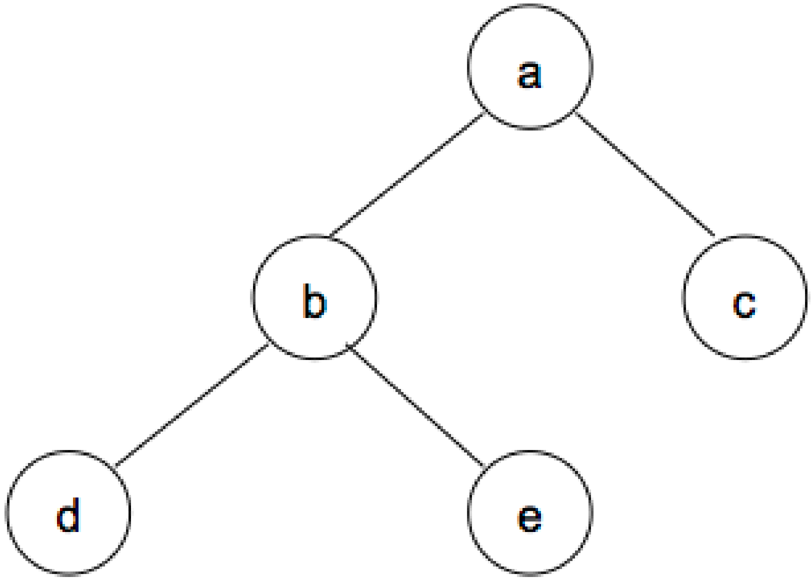
Register allocation - by Graph coloring

- 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 colored if each each pair of neighboring nodes have different colors.

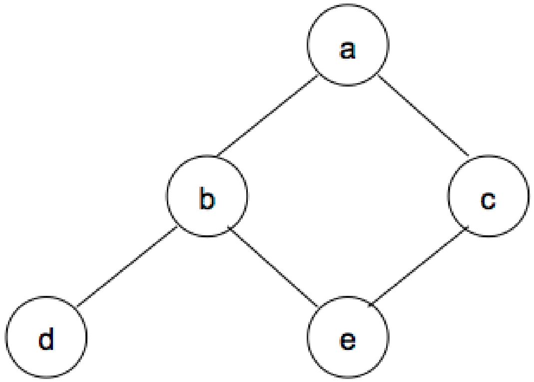
Graph coloring - a simplistic approach

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 G has no node with degree less than K;
    `//` G `is either empty or all of its nodes have degree` $\geq$ `K`
    if G is not empty **then**
        Take one node m out of G, and mark it for spilling;
        Remove all the edges of m from G;
    end
until G is empty;
Take one node at a time from the stack and assign a non conflicting color.

Example 1, available colors = 2



Example 2



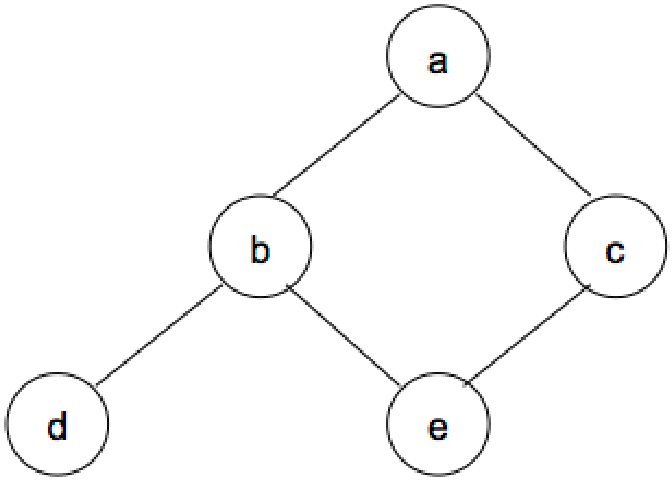
We have to spill.

Graph coloring - Kempe's heuristic

- Algorithm dating back to 1879.

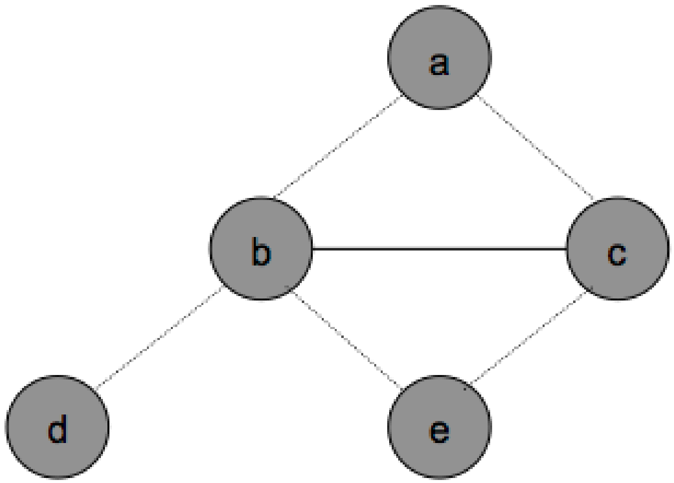
Input: G - the interference graph, K - number of colors
repeat
  repeat
    Remove a node n and all its edges from G, such that degree of n is less than K;
    Push n onto a stack;
  until G has no node with degree less than K;
  // G is either empty or all of its nodes have degree $\geq$ K
  if G is 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

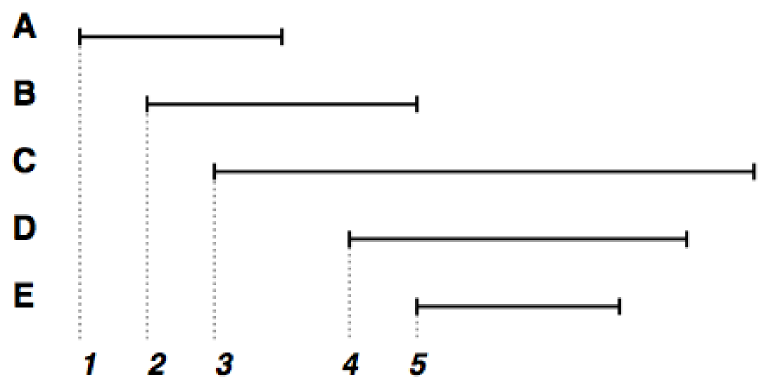
Linear Scan algorithm

```
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