Liveness Analysis and Register Allocation

Leonidas Fegaras
Liveness Analysis

- So far we assumed that we have a very large number of temporary variables stored in registers.
- This is not true for real machines:
  - CISC machines have very few registers.
  - Pentium has 6 general registers only.
- It's highly desirable to use one machine register for multiple temporary variables.
- For example, variables \(a\) and \(b\) do not interfere so they can be assigned to the same register \(R_1\):

\[
\begin{align*}
&a := 0 \\
&L1: b := a + 1 \\
&c := c + b \\
&a := b \times 2 \\
&\text{if } a < 10 \text{ goto } L1 \\
&\text{return } c
\end{align*}
\]

\[
\begin{align*}
&R1 := 0 \\
&L1: R1 := R1 + 1 \\
&R2 := R2 + R1 \\
&R1 := R1 \times 2 \\
&\text{if } R1 < 10 \text{ goto } L1 \\
&\text{return } R2
\end{align*}
\]
Checking Variable Liveness

- A variable $x$ is *live* at a particular point (statement) in a program, if it holds a value that may be needed in the future.
- That is, $x$ is live at a particular point if there is a path (possibly following gotos) from this point to a statement that uses $x$ and there is no assignment to $x$ in any statement in the path.
  - because if there was an assignment to $x$, the old value of $x$ is discarded before it is used.

\[
x := \ldots
\]

no assignments to $x$

use of $x$

$x$ is live

---

### Table

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1: $a := 0$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$b := a+1$</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$c := c+b$</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>$a := b*2$</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>if $a&lt;10$ goto L1</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>return $c$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

X means live
Control Flow Graph (CFG)

• The CFG nodes are individual statements (or maybe basic blocks)

• The CFG edges represent potential flow of control
  – the outgoing edges of a node \( n \) are \( \text{succ}[n] \)
  – the ingoing edges are \( \text{pred}[n] \)

• For each CFG node \( n \) we define
  – \( \text{use}[n] \) to be all the variables used (read) in this statement
  – \( \text{def}[n] \) all the variables assigned a value (written) in this statement

• For example,

1. \( a := 0 \)
2. \( L1: \ b := a+1 \)
3. \( c := c+b \)
4. \( a := b*2 \)
5. \( \text{if } a<10 \text{ goto } L1 \)
6. \( \text{return } c \)

\[ \text{succ}[5]=[6,2] \]
\[ \text{pred}[5]=[4] \]
\[ \text{use}[3]=[b,c] \]
\[ \text{def}[3]=[c] \]
Using CFG

- A variable \( v \) is live at a statement \( n \) if there is a path in the CFG from this statement to a statement \( m \) such that \( v \in \text{use}[m] \) and for each \( n \leq k < m: \ v \not\in \text{def}[k] \)
- That is, there is no assignment to \( v \) in the path from \( n \) to \( m \)
  - For example, \( c \) is live in 4 since it is used in 6 and there is no assignment to \( c \) in the path from 4 to 6
- Liveness analysis analyzes a CFG to determine which places variables are live or not
  - it is a data flow analysis since it flows around the edges of the CFG information the liveness of variables
- For each CFG node \( n \) we derive two sets:
  - \textit{Live-in}: \( \text{in}[n] \) gives all variables that are live before the execution of statement \( n \)
  - \textit{Live-out}: \( \text{out}[n] \) gives all variables that are live after the execution of statement \( n \)
• We compute in/out from the sets: succ, use and def using the following properties of in and out:

• \( v \in \text{use}[n] \Rightarrow v \in \text{in}[n] \)
  
  – ie, if \( v \) is used in \( n \) then \( v \) is live-in in \( n \) (regardless whether it is defined in \( n \))

• \( v \in (\text{out}[n]-\text{def}[n]) \Rightarrow v \in \text{in}[n] \)
  
  – ie, if \( v \) is live after the execution of \( n \) and is not defined in \( n \), then \( v \) is live before the execution of \( n \)

• for each \( s \in \text{succ}[n] \): \( v \in \text{in}[s] \Rightarrow v \in \text{out}[n] \)
  
  – this reflects the formal definition of the liveness of variable \( v \)
Algorithm

• We repeatedly execute the loop until we can't add any more elements:
  
  ```
  foreach n: in[n] := ∅; out[n] := ∅;
  repeat
    foreach n:
      in'[n] := in[n]
      out'[n] := out[n]
      in[n] := use[n] \cup (out[n]-def[n])
      out[n] := \bigcup_{s \in succ[n]} in[s]
  until in'=in and out'=out
  ```

• The algorithm converges very fast if we consider the CFG nodes in the reverse order (when is possible)

• The life of a variable can be directly derived from vector in[]:
  if v \in in[n] then v is live at statement n
Example

1. \( a := 0 \)
2. L1: \( b := a+1 \)
3. \( c := c+b \)
4. \( a := b*2 \)
5. if \( a<10 \) goto L1
6. return \( c \)

<table>
<thead>
<tr>
<th>1st</th>
<th>2nd</th>
</tr>
</thead>
<tbody>
<tr>
<td>use</td>
<td>def</td>
</tr>
<tr>
<td>6</td>
<td>c</td>
</tr>
<tr>
<td>5</td>
<td>a</td>
</tr>
<tr>
<td>4</td>
<td>b</td>
</tr>
<tr>
<td>3</td>
<td>bc</td>
</tr>
<tr>
<td>2</td>
<td>a</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
</tr>
</tbody>
</table>
Interference Graph

- Nodes are the program variables
- For each node \( v \) and \( w \) there is an interference edge if the lives of the variables \( v \) and \( w \) overlap on at least one program point (statement)
- For each program point \( n \), and for each \( x \in \text{in}[n] \) and \( y \in \text{in}[n] \), we draw an edge \((x, y)\)
- For example, the previous program has an interference graph:

```
  a
 / \    \   \
  b    c
```

Example

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>y</th>
<th>z</th>
<th>w</th>
<th>u</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>v := 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td>z := v+1</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td>x := z * v</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>4.</td>
<td>y := x * 2</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>5.</td>
<td>w := x+z*y</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.</td>
<td>u := z+2</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>7.</td>
<td>v := u+w+y</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>8.</td>
<td>return v * u</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>
Register Allocation

- Recall: if there is an edge between two variables in the interference graph then these variables interfere.
- The interference graph is used for assigning registers to temporary variables:
  - If two variables do not interfere then we can use the same register for both of them, thus reducing the number of registers needed.
  - If there is a graph edge between two variables, then we should not assign the same register to them since this register needs to hold the values of both variable at one point of time.
Graph Coloring

• Graph coloring problem: we try to assign one of the \( n \) different colors to graph nodes so that no two adjacent nodes have the same colors
  – Used in map drawing where we have countries or states on the map and we want to color them using a small fixed number of colors so that states that have a common border are not painted with the same color
  – The graph in this case has one node for each state and an edge between two states if they have common borders
Register Allocation

• Is basically graph coloring: registers are colors
• We use a stack of graph nodes. Each time:
  1) we select a node from the interference graph that has fewer than n neighbours
  2) we remove the selected node from the graph (along with its edges)
  3) we push the selected node in the stack
• We continue selecting nodes until we remove all nodes
• This is called simplification of the graph
• The idea is that if we can color the graph after we remove this node, then we can color the original graph (with the node included)
  – Why? because the neighbours of this node can have n-1 different colors in the worst case; so we can just assign the available nth color to the node
Spilling

- Sometimes though we cannot simplify any further because all nodes have n or more neighbours.
- In that case, we select one node (ie. variable) to be spilled into memory instead of assigning a register to it.
- This is called *spilling* and the spilled victim can be selected based on priorities, eg
  - which variable is used less frequently
  - is it outside a loop, etc
- The spilled node is also pushed on the stack.
Selection

• When the graph is completely reduced
  1) we pop the stack one node at a time
  2) we rebuild the interference graph and at the same time we assign a color to
     the popped-out node so that its color is different from the colors of its
     neighbours

• This is called the *selection phase*

• If we can't assign a color to a node, we spill out the node into
  memory
  – a node selected to be spilled out during the spill phase does not necessarily
    mean that it will actually spilled into memory at the end

• If there are spilled nodes, we use a memory access for each
  spilled variable
  – eg. we can use the frame location $fp-24$ to store the spilled temporary
    variable and we replace all occurrences of this variable in the program
    with $M[fp-24]$

Example

Selection: x v z w u y
Registers: x=R2, y=R0, z=R1, w=R2, u=R1, v=R0
Coalescing

• If there is a move instruction in a program $X := Y$ and there is no conflict between $X$ and $Y$, we can use the same register for both $X$ and $Y$ and remove the move entirely from the program
  – we merge the graph nodes for $X$ and $Y$ in the graph into one node
  – nodes are now labelled by sets of variables, instead of just one variable

• It is a good:
  – it reduces the number of registers needed and it removes the move instructions

• It is bad:
  – it increases the number of neighbours of the merged nodes, which may lead to an irreducible graph and a potential spilling

• We add another phase to the register allocation algorithm, called *coalescing*, that coalesces move related nodes

• If we derive an irreducible graph at some point of time, we do *freezing*, that de-coalesces one node
Why is it Useful

- Coalescing is very useful when handling callee-save registers in a procedure
  - Suppose that r3 is a callee-save register. The procedure needs to
    - save this register into a temporary variable at the beginning of the procedure (eg. A := r3)
    - restore it at the end of the procedure (ie. r3 := A)
  - That way, if r3 is not used at all during the procedure body, it will be coalesced with A and the move instructions will be removed
- Coalescing can happen in many other different situations as long as there is no interference
- Note that registers in a program are handled as temporary variables with a preassigned color (precolored nodes)
  - This means that precolored nodes can only be coalesced with other nodes (they cannot be simplified or spilled)
Criteria for Coalescing

• Let $n$ be the number of available registers

• Briggs criterion:
  – we coalesce two nodes if the merged node has fewer than $n$ neighbours of degree greater than or equal to $n$

• George criterion:
  – we coalesce nodes if all the neighbours of one of the nodes with degree greater than or equal to $n$ already interfere with the other node
Example

r1, r2 are caller-save registers
r3 is callee-save register

int f ( int a, int b ) {
    int d = 0;
    int e = a;
    do {
        d = d+b;
        e = e-1;
    } while (e>0);
    return d;
}

enter: c = r3
       a = r1
       b = r2
       d = 0
       e = a

loop: d = d+b
      e = e-1
      if e>0 goto loop
      r1 = d
      r3 = c
      return

(r1, r3 live out)
Example (cont.)

```
enter:  c = r3
       a = r1
       b = r2
       d = 0
       e = a

loop:  d = d+b
       e = e-1
       if e>0 goto loop
       r1 = d
       r3 = c
       return

(r1, r3 live out)
```

Cannot simplify now
Need to spill a variable
Calculating Spill Priorities

Assume that the loop is done 10 times
Spill priority = (uses+defs) / degree

<table>
<thead>
<tr>
<th>node</th>
<th>uses+defs</th>
<th>degree</th>
<th>spill priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>2</td>
<td>4</td>
<td>0.50</td>
</tr>
<tr>
<td>b</td>
<td>1+10</td>
<td>4</td>
<td>2.75</td>
</tr>
<tr>
<td>c</td>
<td>2</td>
<td>5</td>
<td>0.40</td>
</tr>
<tr>
<td>d</td>
<td>2+2*10</td>
<td>4</td>
<td>5.50</td>
</tr>
<tr>
<td>e</td>
<td>1+3*10</td>
<td>3</td>
<td>10.33</td>
</tr>
</tbody>
</table>

c has the lowest priority (ie, is used the least)
So we spill c
After Spilling c

We coalesce a and e because the merged node (ae) will have fewer than n neighbours of degree $\geq n$

We can now coalesce r2 with b and r1 with ae

We can now simplify

$$\begin{align*}
    r3 &\quad a = r1 \\
    r2b &\quad b = r2 \\
    d &\quad d = r3 \\
    r1ae &\quad e = r1
\end{align*}$$
Code Generation for Trees

• Goal: generate assembly code for complex expression trees using the fewest number of registers to store *intermediate results*

• Suppose that we have two-address instructions of the form
  \[ \text{op} \; \text{Ri, T} \]

• where
  – \text{op} is an operation (add, sub, mult, etc)
  – \text{Ri} is a register (R1, R2, R3, etc)
  – \text{T} is an address mode such as a memory reference, a register, indirect access, indexing etc

• We also have a move instruction of the form:
  \[ \text{load} \; \text{Ri, T} \]
Example

- For example, for the expression \((A-B)+((C+D)+(E*F))\), which corresponds to the AST:

```
  +
  / \                             load R2, C
 /   \ add R2, D
/     \\
-     + load R1, E
/ \   / \ mult R1, F
/ \   / \ add R2, R1
/ \   / \ load R1, A
A B + * sub R1, B
/ \   / \ add R1, R2
/ \   / \ C D E F
```

- we want to generate the assembly code at the right
- That is, we used only two register
Sethi-Ullman Algorithm

- Generates code with the least number of registers
- two phases:
  1) The *numbering phase* assigns a number to each tree node that indicates how many registers are needed to evaluate the subtree of this node
  2) The *code generation* phase generates code for each subtree recursively (bottom-up)
How do we Know How Many Registers we Need?

- Suppose that for a tree node T, we need l registers to evaluate its left subtree and r registers to evaluate its right subtree.
- Then if one of these numbers is larger, say \( l > r \), then we can evaluate the left subtree first and store its result into one of the registers, say \( R_k \):
  - Now we can evaluate the right subtree using the same registers we used for the left subtree, except of course \( R_k \) since we want to remember the result of the left subtree.
  - This means that we need l registers to evaluate T too.
- The same happens if \( r > l \) but now we need to evaluate the right subtree first and store the result to a register.
- If \( l = r \) we need an extra register \( r+1 \) to remember the result of the left subtree.
- If T is a tree leaf, then the number of registers to evaluate T is either 1 or 0 depending whether T is a left or a right subtree.
Numbering Phase

- Algorithm:
  
  if T is a left leaf then \( \text{regs}(T) = 1 \)
  
  else if T is a right leaf then \( \text{regs}(T) = 0 \)
  
  else let \( l = \text{regs}(T.\text{left}) \), \( r = \text{regs}(T.\text{right}) \)
  
  if \( l = r \) then \( \text{regs}(T) = r + 1 \)
  
  else \( \text{regs}(T) = \max(l, r) + 1 \)

- Example:

```
+  2
 / \    
/  \     
/   \    
-    +  1  2
/ \  / \    /  
/  \ /  \    / \
A   B   +     *
/ \ / \  1 0 1 1
/ \ /   / \    /  
C   D   E   F  1 0 1 0
```


Code Generation

• We use a stack of available registers
  – it contains all the registers in order (lower register at the top)

  generate(T) =
    if T is a leaf write “load top(), T”
    if T is an internal node with children l and r then
      if regs(r) = 0 then { generate(l); write “op top(), r” }
      if regs(l) ≥ regs(r)
      then {
        generate(l)
        R := pop()
        generate(r)
        write “op R, top()”
        push(R) }
      if regs(l) < regs(r)
      then {
        swap the top two elements of the stack
        generate(r)
        R := pop()
        generate(l)
        write “op top(), R”
        push(R)
        swap the top two elements of the stack }