Global Register Allocation
- Part 2

Y N Srikant
Computer Science and Automation
Indian Institute of Science
Bangalore 560012

NPTEL Course on Compiler Design
Outline

- Issues in Global Register Allocation
- The Problem
- Register Allocation based in Usage Counts
- Linear Scan Register allocation
- Chaitin’s graph colouring based algorithm

Topics 1,2,3, and part of 4 were covered in part 1 of the lecture.
A Fast Register Allocation Scheme

- Linear scan register allocation (Poletto and Sarkar 1999) uses the notion of a live interval rather than a live range.
- Is relevant for applications where compile time is important such as in dynamic compilation and in just-in-time compilers.
- Other register allocation schemes based on raph colouring are slow and are not suitable for JIT and dynamic compilers.
Linear Scan Register Allocation

- Assume that there is some numbering of the instructions in the intermediate form.
- An interval $[i,j]$ is a **live interval** for variable $v$ if there is no instruction with number $j' > j$ such that $v$ is live at $j'$ and no instruction with number $i' < i$ such that $v$ is live at $i$.
- This is a conservative approximation of live ranges: there may be subranges of $[i,j]$ in which $v$ is not live but these are ignored.
Live Interval Example

\[ v \text{ not live} \]
\[ \vdots \]
\[ v \text{ not live} \]
\[ i': \]
\[ \vdots \]
\[ i' \text{ does not exist} \]

\[ \vdots \]
\[ i: \]
\[ \vdots \]
\[ i - j : \text{live interval for variable } v \]

\[ j: \]
\[ \vdots \]
\[ j': \]
\[ \vdots \]
\[ j' \text{ does not exist} \]
Example

If (cond)
then A=
else B=

X: if (cond)
then =A
else = B

A NOT LIVE HERE

LIVE INTERVAL FOR A
Live Intervals

- Given an order for pseudo-instructions and live variable information, live intervals can be computed easily with one pass through the intermediate representation.
- Interference among live intervals is assumed if they overlap.
- Number of overlapping intervals changes only at start and end points of an interval.
The Data Structures

- Live intervals are stored in the sorted order of increasing start point.
- At each point of the program, the algorithm maintains a list (active list) of live intervals that overlap the current point and that have been placed in registers.
- active list is kept in the order of increasing end point.
Example

Active lists (in order of increasing end point):

Active(A) = \{i1\}
Active(B) = \{i1, i5\}
Active(C) = \{i8, i5\}
Active(D) = \{i7, i4, i11\}

Sorted order of intervals (according to start point):
i1, i5, i8, i2, i9, i6, i3, i10, i7, i4, i11

Three registers enough for computation without spills
{ active := [ ];
    for each live interval i, in order of increasing start point do
    { ExpireOldIntervals (i);
        if length(active) == R then SpillAtInterval(i);
        else { register[i] := a register removed from the pool of free registers;
                add i to active, sorted by increasing end point
        }
    }
}
The Algorithm (2)

ExpireOldIntervals (i)
{ for each interval j in active, in order of increasing end point do
  { if endpoint[j] > startpoint[i] then return
  else { remove j from active;
         add register[j] to pool of free registers;
       }
  }
}
The Algorithm (3)

SpillAtInterval (i)
{ spill := last interval in active;
  \textit{if} endpoint [spill] \geq \text{endpoint } [i] \textit{then}
  { register [i] := register [spill];
    location [spill] := new stack location;
    remove spill from active;
    add i to active, sorted by increasing end point;
  } \textit{else} location [i] := new stack location;
}
**Example 1**

Active lists (in order of increasing end pt)

Active(A) = \{i1\}

Active(B) = \{i1, i5\}

Active(C) = \{i8, i5\}

Active(D) = \{i7, i4, i11\}

Sorted order of intervals (according to start point):

i1, i5, i8, i2, i9, i6, i3, i10, i7, i4, i11

Three registers enough for computation without spills
Example 2

1, 2: give A, B register
3: Spill C since endpoint [C] > endpoint [B]

2 registers available

4: A expires, give D register
5: B expires, E gets register
Example 3

1, 2: give A, B register
3: Spill B since endpoint [B] > endpoint [C]
give register to C

4: A expires, give D register
5: C expires, E gets register
Complexity of the Linear Scan Algorithm

- If $V$ is the number of live intervals and $R$ the number of available physical registers, then if a balanced binary tree is used for storing the active intervals, complexity is $O(V \log R)$.

- Empirical results reported in literature indicate that linear scan is significantly faster than graph colouring algorithms and code emitted is at most 10% slower than that generated by an aggressive graph colouring algorithm.
Chaitin’s Formulation of the Register Allocation Problem

- A graph colouring formulation on the interference graph
- Nodes in the graph represent live ranges of variables or entities called webs
- An edge connects two live ranges that interfere or conflict with one another
- Usually both adjacency matrix and adjacency lists used to represent the graph.
Chaitin’s Formulation of the Register Allocation Problem

- Assign colours to the nodes such that two nodes connected by an edge are not assigned the same colour.
  - The number of colours available is the number of registers available on the machine.
  - A k-colouring of the interference graph is mapped into an allocation with k registers.
Example

- Two colourable
- Three colourable
Idea behind Chaitin’s Algorithm

- Choose an arbitrary node of degree less than $k$ and put it on the stack
- Remove that vertex and all its edges from the stack
  - This may decrease the degree of some other nodes and cause some more nodes to have degree less than $k$
- At some point, if all vertices have degree greater than or equal to $k$, some node has to be spilled
- If no vertex needs to be spilled, successively pop vertices off stack and colour them in lowest colour not used by neighbour.
Simple example – Given Graph

STACK

3 REGISTERS
Simple Example – Delete Node 1
Simple Example – Delete Node 2
## Simple Example – Delete Node 4

### Stack

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

### Graph

![Graph Image]

- **3 REGISTERS**
- **Nodes:** 1, 2, 3, 4, 5
- **Edges:**
  - 1 to 2
  - 1 to 3
  - 2 to 4
  - 2 to 5
  - 3 to 4
  - 3 to 5

By deleting Node 4, the graph changes as follows:

- **Edges:**
  - 1 to 2
  - 1 to 3
  - 2 to 4
  - 2 to 5
  - 3 to 4
  - 3 to 5

---

Y.N. Srikant
Simple Example – Delete Nodes 3

STACK

3
4
2
1

3 REGISTERS

1 - 2 - 4 - 5

3 - 2 - 5

1 - 3 - 5

1 - 4 - 5

1 - 2 - 5

1 - 3 - 5

1 - 4 - 5

1 - 2 - 5

1 - 3 - 5

1 - 4 - 5

1 - 2 - 5
Simple Example – Delete Nodes 5

STACK

3 REGISTERS

5
3
4
2
1

1
2
3
4
5
Simple Example – Colour Node 5

STACK

COLOURS

3 REGISTERS

1

2

4

3
Simple Example – Colour Node 3
Simple Example – Colour Node 4

STACK

3
2
1

COLOURS

3 REGISTERS

4
5

3

1
Simple Example – Colour Node 2

STACK

COLOURS

3 REGISTERS
Simple Example – Colour Node 1

STACK

COLOURS

3 REGISTERS
Steps in Chaitin’s Algorithm

- Identify units for allocation (sometimes called renumbering)
- Build the interference graph
- Coalesce by removing unnecessary move or copy instructions
- Colour the graph, thereby selecting registers
- Compute spill costs, simplify and add spill code till graph is colourable
The Chaitin Framework
## An Example

<table>
<thead>
<tr>
<th>Original code</th>
<th>Code with symbolic registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = 2$</td>
<td>1. $S1 = 2$; (lv of $S1$: 1-5)</td>
</tr>
<tr>
<td>$y = 4$</td>
<td>2. $S2 = 4$; (lv of $S2$: 2-5)</td>
</tr>
<tr>
<td>$w = x + y$</td>
<td>3. $S3 = s1 + s2$; (lv of $S3$: 3-4)</td>
</tr>
<tr>
<td>$z = x + 1$</td>
<td>4. $S4 = s1 + 1$; (lv of $S4$: 4-6)</td>
</tr>
<tr>
<td>$u = x * y$</td>
<td>5. $S5 = s1 * s2$; (lv of $S5$: 5-6)</td>
</tr>
<tr>
<td>$x = z * 2$</td>
<td>6. $S6 = s4 * 2$; (lv of $S6$: 6-... )</td>
</tr>
</tbody>
</table>
INTERFERENCE GRAPH
HERE ASSUME VARIABLE Z (s4) CANNOT OCCUPY r1
Example (continued)

Final register allocated code

\[ r1 = 2 \]
\[ r2 = 4 \]
\[ r3 = r1 + r2 \]
\[ r3 = r1 + 1 \]
\[ r1 = r1 \times r2 \]
\[ r2 = r3 + r2 \]

Three registers are sufficient for no spills
The definition points and the use points for each variable $v$ are assumed to be known.

Each definition with its set of uses for $v$ is a du-chain.

A web is a maximal union of du-chains such that, for each definition $d$ and use $u$, either $u$ is in the du-chain of $d$, or there exists a sequence $d = d_1, u_1, d_2, u_2, \ldots, d_n, u_n$ such that for each $i$, $u_i$ is in the du-chains of both $d_i$ and $d_{i+1}$. 
Renumbering - Webs

- Each web is given a unique symbolic register.
- Webs arise when variables are redefined several times in a program.
- Webs have intersecting du-chains, intersecting at the points of join in the control flow graph.
Example of Webs

W1: def x in B2, def x in B3, use x in B4, Use x in B5
W2: def x in B5, use x in B6
W3: def y in B2, use y in B4
W4: def y in B1, use y in B3
Build Interference Graph

- Create a node for each web and for each physical register in the interference graph.
- If two distinct webs interfere, that is, a variable associated with one web is live at a definition point of another, add an edge between the two webs.
- If a particular variable cannot reside in a register, add an edge between all webs associated with that variable and the register.
Copy Subsumption or Coalescing

- Consider a copy instruction: \( b := e \) in the program.
- If the live ranges of \( b \) and \( e \) do not overlap, then \( b \) and \( e \) can be given the same register (colour).
  - Implied by lack of any edges between \( b \) and \( e \) in the interference graph.
- The copy instruction can then be removed from the final program.
- Coalesce by merging \( b \) and \( e \) into one node that contains the edges of both nodes.
Copy Subsumption or Coalescing

- Copy subsumption is not possible; lr(e) and lr(new b) interfere.
- Copy subsumption is possible; lr(e) and lr(new b) do not interfere.
Example of coalescing

Copy inst: \( b := e \)

BEFORE

AFTER
Coalescing

- Coalesce all possible copy instructions
  - Rebuild the graph
    - may offer further opportunities for coalescing
    - build-coalesce phase is repeated till no further coalescing is possible.

- Coalescing reduces the size of the graph and possibly reduces spilling
Simple fact

- Suppose the no. of registers available is R.
- If a graph G contains a node $n$ with fewer than R neighbors then removing $n$ and its edges from G will not affect its R-colourability.
- If $G' = G - \{n\}$ can be coloured with R colours, then so can G.
- After colouring $G'$, just assign to $n$, a colour different from its R-1 neighbours.
Simplification

- If a node $n$ in the interference graph has degree less than $R$, remove $n$ and all its edges from the graph and place $n$ on a colouring stack.
- When no more such nodes are removable then we need to spill a node.
- Spilling a variable $x$ implies
  - loading $x$ into a register at every use of $x$
  - storing $x$ from register into memory at every definition of $x$
Spilling Cost

- The node to be spilled is decided on the basis of a spill cost for the live range represented by the node.
- Chaitin’s estimate of spill cost of a live range $v$

$$\text{cost}(v) = \sum_{\text{all load or store operations in a live range } v} c \times 10^d$$

- where $c$ is the cost of the op and $d$, the loop nesting depth.
- 10 in the eqn above approximates the no. of iterations of any loop.
- The node to be spilled is the one with $\text{MIN}(\text{cost}(v)/\text{deg}(v))$
Spilling Heuristics

- Multiple heuristic functions are available for making spill decisions (cost(v) as before)
  1. \( h_0(v) = \frac{\text{cost}(v)}{\text{degree}(v)} \): Chaitin’s heuristic
  2. \( h_1(v) = \frac{\text{cost}(v)}{[\text{degree}(v)]^2} \)
  3. \( h_2(v) = \frac{\text{cost}(v)}{[\text{area}(v) \times \text{degree}(v)]} \)
  4. \( h_3(v) = \frac{\text{cost}(v)}{[\text{area}(v) \times (\text{degree}(v))^2]} \)

where \( \text{area}(v) = \sum_{\text{all instructions } I \text{ in the live range } v} \text{width}(v, I) \times 5^{\text{depth}(v,I)} \)

- \( \text{width}(v,I) \) is the number of live ranges overlapping with instruction I and \( \text{depth}(v,I) \) is the depth of loop nesting of I in v
Spilling Heuristics

- area(v) represents the global contribution by v to register pressure, a measure of the need for registers at a point

- Spilling a live range with high area releases register pressure; i.e., releases a register when it is most needed

- Choose v with MIN(h_i(v)), as the candidate to spill, if h_i is the heuristic chosen

- It is possible to use different heuristics at different times
Here $R = 3$ and the graph is 3-colourable
No spilling is necessary
A 3-colourable graph which is not 3-coloured by colouring heuristic
Spilling a Node

- To spill a node we remove it from the graph and represent the effect of spilling as follows (It cannot just be removed from the graph).
  - Reload the spilled object at each use and store it in memory at each definition point
  - This creates new webs with small live ranges but which will need registers.

- After all spill decisions are made, insert spill code, rebuild the interference graph and then repeat the attempt to colour.

- When simplification yields an empty graph then select colours, that is, registers
Effect of Spilling

- **W1: def x in B2, def x in B3, use x in B4, use x in B5**
- **W2: def x in B5, use x in B6**
- **W3: def y in B2, use y in B4**
- **W4: def y in B1, use y in B3**

- **x is spilled in web W1**
Effect of Spilling

W1

Def x

x = tmp

Use x

Use y

Def y

B2

W4

W6

W7

x = tmp

Use x

Def x

B4

B5

x = tmp

Use x

Def y

B1

W5

W2

W3

Interference Graph

w4

w8

w5

w3

w6

w1

w2

w7
Repeat
V = pop(stack).
Colours_used(v) = colours used by neighbours of V.
Colours_free(v) = all colours - Colours_used(v).
Colour (V) = any colour in Colours_free(v).
Until stack is empty

- Convert the colour assigned to a symbolic register to the corresponding real registers name in the code.
Drawbacks of the Algorithm

- Constructing and modifying interference graphs is very costly as interference graphs are typically huge.
- For example, the combined interference graphs of procedures and functions of gcc in mid-90’s have approximately 4.6 million edges.
Some modifications

- **Careful coalescing**: Do not coalesce if coalescing increases the degree of a node to more than the number of registers.

- **Optimistic colouring**: When a node needs to be spilled, put it into the colouring stack instead of spilling it right away.
  - spill it only when it is popped and if there is no colour available for it.
  - this could result in colouring graphs that need spills using Chaitin’s technique.
A 3-colourable graph which is not 3-coloured by colouring heuristic, but coloured by optimistic colouring

Example

Say, 1 is chosen for spilling. Push it onto the stack, and remove it from the graph. The remaining graph (2,3,4,5) is 3-colourable. Now, when 1 is popped from the colouring stack, there is a colour with which 1 can be coloured. It need not be spilled.