Global Register Allocation - 3

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

NPTEL Course on Principles of Compiler Design
Outline

- Issues in Global Register Allocation (in part 1)
- The Problem (in part 1)
- Register Allocation based in Usage Counts (in part 2)
- Linear Scan Register allocation (in part 2)
- Chaitin’s graph colouring based algorithm
Chaitin’s Formulation of the Register Allocation Problem

- A graph colouring formulation on the interference graph
- Nodes in the graph represent either 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 are 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 onto an allocation with k registers
Example

- Two colourable
- Three colourable

![Diagram showing two and three colourable graphs]
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 graph
  - 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 a colour not used by neighbours (reuse colours as far as possible)
Simple example – Given Graph

STACK

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

STACK

3 REGISTERS

Y.N. Srikant
Simple Example – Delete Node 4

STACK

3 REGISTERS
Simple Example – Delete Nodes 3

3 REGISTERS

STACK

3
4
2
1
Simple Example – Delete Nodes 5

STACK

| 5 | 3 | 4 | 2 | 1 |

3 REGISTERS

1 2 3 4 5
Simple Example – Colour Node 5

STACK

| 3 | 4 | 2 | 1 |

COLOURS

3 REGISTERS

5
Simple Example – Colour Node 3
Simple Example – Colour Node 4

STACK

COLOURS

3 REGISTERS
Simple Example – Colour Node 2

STACK

COLOURS

3 REGISTERS

1
Simple Example – Colour Node 1

STACK

COLOURS

3 REGISTERS

1
2
3
4
5
Steps in Chaitin’s Algorithm

- Identify units for allocation
  - Renames variables/symbolic registers in the IR such that each live range has a unique name (number)
  - A live range is entitled to get a register
- 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
Chaitin’s Framework
Example of Renaming

\[ \begin{align*}
  a &= a \\
  s1 &= s1 \\
  s2 &= s2 \\
  a &= a \\
  s1 &= s1 \\
  s2 &= s2
\end{align*} \]
### An Example

<table>
<thead>
<tr>
<th>Original code</th>
<th>Code with symbolic registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. x = 2</td>
<td>1. s1 = 2; (lv of s1: 1-5)</td>
</tr>
<tr>
<td>2. y = 4</td>
<td>2. s2 = 4; (lv of s2: 2-5)</td>
</tr>
<tr>
<td>3. w = x + y</td>
<td>3. s3 = s1 + s2; (lv of s3: 3-4)</td>
</tr>
<tr>
<td>4. z = x + 1</td>
<td>4. s4 = s1 + 1; (lv of s4: 4-6)</td>
</tr>
<tr>
<td>5. u = x * y</td>
<td>5. s5 = s1 * s2; (lv of s5: 5-6)</td>
</tr>
<tr>
<td>6. 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

\[
\begin{align*}
r_1 &= 2 \\
r_2 &= 4 \quad \text{Three registers are sufficient for no spills} \\
r_3 &= r_1 + r_2 \\
r_3 &= r_1 + 1 \\
r_1 &= r_1 \times r_2 \\
r_2 &= r_3 + r_2
\end{align*}
\]
More Complex Example

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 LV and for each physical register in the interference graph.
- If two distinct LVs interfere, that is, a variable associated with one LV is live at a definition point of another add an edge between the two LVs.
- If a particular variable cannot reside in a register, add an edge between all LVs 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
Example of coalescing

Copy inst: \( b := e \)

BEFORE

AFTER
Copy Subsumption or Coalescing

Copy subsumption is not possible; \( \text{lr}(e) \) and \( \text{lr}(\text{new } b) \) interfere.

Copy subsumption is possible; \( \text{lr}(e) \) and \( \text{lr}(\text{new } b) \) do not interfere.
Copy Subsumption Repeatedly

\[ \text{l.r of } x \quad b = e \quad \text{l.r of } e \]
\[ \text{l.r of } b \quad a = b \quad \text{l.r of } a \]

Copy subsumption happens twice - once between \( b \) and \( e \), and second time between \( a \) and \( b \). \( e \), \( b \), and \( a \) are all given the same register.
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 $G' = G\setminus\{n\}$ 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 $\text{MIN}($cost($v$)/deg($v$))
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 be simply removed from the graph).
  - Reload the spilled object at each use and store it in memory at each definition point
  - This creates new small live ranges which will also 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
Effect of Spilling

Def x
store x
Def y

load x
Use x
Use y

load x
Use x
Def x

Use x

Def y

Def x
store x
Use y

Interference Graph

w4
w5
w3
w6
w1
w2
w7
Colouring the Graph (selection)

Repeat

v = pop(stack).

Colours_used(v) = colours used by neighbours of v.

Colours_free(v) = all colours - Colours_used(v).

Colour (v) = choose any colour in Colours_free(v).

Until stack is empty

- Convert the colour assigned to a symbolic register to the corresponding real register’s name in the code.
A Complete Example

1. \( t1 = 202 \)
2. \( i = 1 \)
3. L1: \( t2 = i > 100 \)
4. if \( t2 \) goto L2
5. \( t1 = t1 - 2 \)
6. \( t3 = \text{addr}(a) \)
7. \( t4 = t3 - 4 \)
8. \( t5 = 4 \times i \)
9. \( t6 = t4 + t5 \)
10. \( *t6 = t1 \)
11. \( i = i + 1 \)
12. goto L1
13. L2:

<table>
<thead>
<tr>
<th>variable</th>
<th>live range</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>1-10</td>
</tr>
<tr>
<td>i</td>
<td>2-11</td>
</tr>
<tr>
<td>t2</td>
<td>3-4</td>
</tr>
<tr>
<td>t3</td>
<td>6-7</td>
</tr>
<tr>
<td>t4</td>
<td>7-9</td>
</tr>
<tr>
<td>t5</td>
<td>8-9</td>
</tr>
<tr>
<td>t6</td>
<td>9-10</td>
</tr>
</tbody>
</table>
A Complete Example

<table>
<thead>
<tr>
<th>variable</th>
<th>live range</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>1-10</td>
</tr>
<tr>
<td>i</td>
<td>2-11</td>
</tr>
<tr>
<td>t2</td>
<td>3-4</td>
</tr>
<tr>
<td>t3</td>
<td>6-7</td>
</tr>
<tr>
<td>t4</td>
<td>7-9</td>
</tr>
<tr>
<td>t5</td>
<td>8-9</td>
</tr>
<tr>
<td>t6</td>
<td>9-10</td>
</tr>
</tbody>
</table>
A Complete Example

Assume 3 registers. Nodes t6, t2, and t3 are first pushed onto a stack during reduction.

This graph cannot be reduced further. Spilling is necessary.
A Complete Example

<table>
<thead>
<tr>
<th>Node V</th>
<th>Cost(v)</th>
<th>deg(v)</th>
<th>h₀(v)</th>
</tr>
</thead>
<tbody>
<tr>
<td>t₁</td>
<td>31</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>i</td>
<td>41</td>
<td>3</td>
<td>14</td>
</tr>
<tr>
<td>t₄</td>
<td>20</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>t₅</td>
<td>20</td>
<td>3</td>
<td>7</td>
</tr>
</tbody>
</table>

\[
t₁: 1 + (1+1+1) \times 10 = 31
i: 1 + (1+1+1+1) \times 10 = 41
\]
\[
t₄: (1+1) \times 10 = 20
\]
\[
t₅: (1+1) \times 10 = 20
\]
t₅ will be spilled. Then the graph can be coloured.
A Complete Example

1. \( R1 = 202 \)
2. \( R2 = 1 \)
3. L1: \( R3 = i > 100 \)
4. if \( R3 \) goto L2
5. \( R1 = R1 - 2 \)
6. \( R3 = \text{addr}(a) \)
7. \( R3 = R3 - 4 \)
8. \( t5 = 4 * R2 \)
9. \( R3 = R3 + t5 \)
10. \( *R3 = R1 \)
11. \( R2 = R2 + 1 \)
12. goto L1
13. L2:

\( t5 \): spilled node, will be provided with a temporary register during code generation
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, push it onto 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.