Global Register Allocation

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
Some Issues in Register Allocation

- Which values in a program reside in registers? (register allocation)
- In which register? (register assignment)
  - The two together are usually loosely referred to as register allocation
- What is the unit at the level of which register allocation is done?
  - Typical units are basic blocks, functions and regions.
  - RA within basic blocks is called local RA
  - The other two are known as global RA
  - Global RA requires much more time than local RA
Some Issues in Register Allocation

- Phase ordering between register allocation and instruction scheduling
  - Performing RA first restricts movement of code during scheduling – not recommended
  - Scheduling instructions first cannot handle spill code introduced during RA
    - Requires another pass of scheduling

- Tradeoff between speed and quality of allocation
  - In some cases e.g., in Just-In-Time compilation, cannot afford to spend too much time in register allocation.
The Problem

- Global Register Allocation assumes that allocation is done beyond basic blocks and usually at function level.
- Decision problem related to register allocation:
  - Given an intermediate language program represented as a control flow graph and a number $k$, is there an assignment of registers to program variables such that no conflicting variables are assigned the same register, no extra loads or stores are introduced, and at most $k$ registers are used.
- This problem has been shown to be NP-hard (Sethi 1970).
- Graph colouring is the most popular heuristic used.
- However, there are simpler algorithms as well.
Conflicting variables

- Two variables interfere or conflict if their live ranges intersect
  - A variable is live at a point $p$ in the flow graph, if there is a use of that variable in the path from $p$ to the end of the flow graph
  - A live range of a variable is the smallest set of program points at which it is live.
  - Typically, instruction no. in the basic block along with the basic block no. is the representation for a point.
Example

If (cond)
then A =
else B =

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

A and B both live

Live range of A: B2, B4, B5
Live range of B: B3, B4, B6
Global Register Allocation via Usage Counts (for Single Loops)

- Allocate registers for variables used within loops
- Requires information about liveness of variables at the entry and exit of each basic block (BB) of a loop
- Once a variable is computed into a register, it stays in that register until the end of the BB (subject to existence of next-uses)
- Load/Store instructions cost 2 units (because they occupy two words)
Global Register Allocation via Usage Counts (for Single Loops)

1. For every usage of a variable $v$ in a BB, until it is first defined, do:
   - $\text{savings}(v) = \text{savings}(v) + 1$
   - after $v$ is defined, it stays in the register any way, and all further references are to that register

2. For every variable $v$ computed in a BB, if it is live on exit from the BB,
   - count a savings of 2, since it is not necessary to store it at the end of the BB
Global Register Allocation via Usage Counts (for Single Loops)

- Total savings per variable $v$ are
  \[
  \sum_{B \in \text{Loop}} (\text{savings}(v,B) + 2 \times \text{liveandcomputed}(v,B))
  \]
  - $\text{liveandcomputed}(v,B)$ in the second term is 1 or 0

- On entry to (exit from) the loop, we load (store) a variable live on entry (exit), and lose 2 units for each
  - But, these are “one time” costs and are neglected

- Variables, whose savings are the highest will reside in registers
Global Register Allocation via Usage Counts (for Single Loops)

Savings for the variables

<table>
<thead>
<tr>
<th></th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
<th>B4</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>(0+2)+(1+0)+(1+0)+(0+0) = 4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>(3+0)+(0+0)+(0+0)+(0+2) = 5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>(1+0)+(1+0)+(0+0)+(1+0) = 3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>(0+2)+(1+0)+(0+0)+(1+0) = 4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>(0+2)+(0+2)+(1+0)+(0+0) = 5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f</td>
<td>(1+0)+(1+0)+(0+2)+(0+0) = 4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If there are 3 registers, they will be allocated to the variables, a, b, and e.
Global Register Allocation via Usage Counts (for Nested Loops)

- We first assign registers for inner loops and then consider outer loops. Let $L_1$ nest $L_2$

- For variables assigned registers in $L_2$, but not in $L_1$
  - load these variables on entry to $L_2$ and store them on exit from $L_2$

- For variables assigned registers in $L_1$, but not in $L_2$
  - store these variables on entry to $L_2$ and load them on exit from $L_2$

- All costs are calculated keeping the above rules
Global Register Allocation via Usage Counts (for Nested Loops)

- **case 1**: variables x, y, z assigned registers in L2, but not in L1
  - Load x, y, z on entry to L2
  - Store x, y, z on exit from L2
- **case 2**: variables a, b, c assigned registers in L1, but not in L2
  - Store a, b, c on entry to L2
  - Load a, b, c on exit from L2
- **case 3**: variables p, q assigned registers in both L1 and L2
  - No special action
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 graph 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

sequentially numbered instructions

\[ v \text{ live } \]

\[ \begin{array}{cccc}
   & \vdots & i': & \cdots \\
   \vdots & \vdots & \vdots & \vdots \\
   & \vdots & \vdots & \vdots \\
   i: & \vdots & \vdots & \vdots \\
   \vdots & \vdots & \vdots & \vdots \\
   j: & \vdots & \vdots & \vdots \\
   \vdots & \vdots & \vdots & \vdots \\
   j': & \vdots & \vdots & \vdots \\
\end{array} \]

\{ \]

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

\[ v \text{ live } \]

\[ \times \]

\[ \begin{array}{cccc}
   & \vdots & i': & \cdots \\
   \vdots & \vdots & \vdots & \vdots \\
   & \vdots & \vdots & \vdots \\
   i: & \vdots & \vdots & \vdots \\
   \vdots & \vdots & \vdots & \vdots \\
   j: & \vdots & \vdots & \vdots \\
   \vdots & \vdots & \vdots & \vdots \\
   j': & \vdots & \vdots & \vdots \\
\end{array} \]

\[ \times \]

\[ i' \text{ does not exist} \]

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

If (cond)
then A=
else B=

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

A NOT LIVE HERE

LIVE INTERVAL FOR A

If (cond)
A=
= A

If (cond)
B=
= B
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 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
The Algorithm (1)

```plaintext
{ 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* continue

  *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]; \\
  \text{location} [spill] := \text{new stack location}; \\
  \text{remove spill from active}; \\
  \text{add} \ i \ \text{to active, sorted by increasing end point}; \\
  \} \ \textit{else} \ \text{location} [i] := \text{new stack location}; \\
\}
Example 1

- 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
Example 2

1, 2: give A, B register
3: Spill C since endpoint[C] > endpoint [B]
4: A expires, give D register
5: B expires, E gets register

2 registers available
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

2 registers available
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 $O(V \log R)$.
  - Active list can be at most ‘$R$’ long
  - Insertion and deletion are the important operations

- 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 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
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

1

2

3

4

5

STACK

3 REGISTERS
Simple Example – Delete Node 1

STACK

3 REGISTERS
Simple Example – Delete Node 2

STACK

3 REGISTERS

1
2
3
4
5

2
1

3 REGISTERS

Y.N. Srikant
Simple Example – Delete Node 4
Simple Example – Delete Nodes 3
Simple Example – Delete Nodes 5

STACK

1
2
3
4
5

3 REGISTERS
Simple Example – Colour Node 5

STACK

3
4
2
1

3 REGISTERS

COLOURS

5
Simple Example – Colour Node 3

STACK

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

3 REGISTERS

<table>
<thead>
<tr>
<th>3</th>
</tr>
</thead>
</table>

COLOURS

<table>
<thead>
<tr>
<th>5</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>4</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>2</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>1</th>
</tr>
</thead>
</table>
Simple Example – Colour Node 4

STACK

3 REGISTERS

COLOURS

3

4

5

2

1

Y.N. Srikant
Simple Example – Colour Node 2

STACK

COLOURS

3 REGISTERS

1
Simple Example – Colour Node 1

STACK

COLOURS

3 REGISTERS

1 2 3

4 5

IISc
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)
- 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

- RENUMBER
- BUILD
- COALESCE
- SPILL CODE
- SPILL COST
- SIMPLIFY
- SELECT
Example of Renaming

\[ a = a = a = a = a \]

\[ s1 = s1 = s1 = s2 = s2 \]

Renaming
## 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. $s1=2; \text{ (lv of } s1: 1-5)$</td>
</tr>
<tr>
<td>$y = 4$</td>
<td>2. $s2=4; \text{ (lv of } s2: 2-5)$</td>
</tr>
<tr>
<td>$w = x + y$</td>
<td>3. $s3=s1+s2; \text{ (lv of } s3: 3-4)$</td>
</tr>
<tr>
<td>$z = x + 1$</td>
<td>4. $s4=s1+1; \text{ (lv of } s4: 4-6)$</td>
</tr>
<tr>
<td>$u = x \times y$</td>
<td>5. $s5=s1\times s2; \text{ (lv of } s5: 5-6)$</td>
</tr>
<tr>
<td>$x = z \times 2$</td>
<td>6. $s6=s4\times 2; \text{ (lv of } s6: 6- ...)$</td>
</tr>
</tbody>
</table>
INTERFERENCE GRAPH
HERE ASSUME VARIABLE Z (s4) CANNOT OCCUPY r1
Example (continued)

Final register allocated code

\[
\begin{align*}
  r_1 &= 2 \\
r_2 &= 4 \\
r_3 &= r_1 + r_2 \\
r_3 &= r_1 + 1 \\
r_1 &= r_1 \times r_2 \\
r_2 &= r_3 + r_2
\end{align*}
\]

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 $d_{i+1}$. 
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; \( \text{l.r}(e) \) and \( \text{l.r}(\text{new } b) \) interfere

Copy subsumption is possible; \( \text{l.r}(e) \) and \( \text{l.r}(\text{new } b) \) do not interfere
Example of coalescing

Copy inst: $b := e$

BEFORE

AFTER

b

f

d

e

c

be

da

c

d

a

f
Copy Subsumption Repeatedly

- l.r of x
- l.r of b
- l.r of a
- l.r of e

b = e

a = b

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 $G' = G - \{n\}$ 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 $\text{MIN}(\text{cost}(v)/\text{deg}(v))$
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 \( \text{area}(v) = \sum \text{width}(v, I) \times 5^{\text{depth}(v, I)} \)

- \( \text{width}(v, I) \) 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

Example
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

Def x store x Def y

load x Use x Use y

load x Use x Def x

Use x

Def x store x Use y

Def y

Interference Graph

W1

W4

W6

W7

W3

W5

W2

B1

B3

B2

B4

B5

B6

w4

w5

w3

w1

w2

w7

Y.N. Srikant
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) = 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. \( t5 = 4 * i \)
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)*10 = 31
i : 1+(1+1+1+1)*10 = 41
t4: (1+1)*10 = 20
t5: (1+1)*10 = 20
t5 will be spilled. Then the graph can be coloured.

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

1. \[ R_1 = 202 \]
2. \[ R_2 = 1 \]
3. \[ L1: \quad R_3 = i > 100 \]
4. \[ \text{if } R_3 \text{ goto } L2 \]
5. \[ R_1 = R_1 - 2 \]
6. \[ R_3 = \text{addr}(a) \]
7. \[ R_3 = R_3 - 4 \]
8. \[ t_5 = 4 \times R_2 \]
9. \[ R_3 = R_3 + t_5 \]
10. \[ *R_3 = R_1 \]
11. \[ R_2 = R_2 + 1 \]
12. \[ \text{goto } L1 \]
13. \[ L2: \]

\( t_5 \): 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.