Register Allocation via Graph Coloring

The basic idea behind register allocation via graph coloring is to reduce register spillage by globally assigning variables to registers across an entire program module via the five basic steps below:

I. Assign each object (intermediate result name, variable, or constant) to a distinct symbolic register called $s$. Re-constitute the intermediate code with the assigned symbolic registers.

II. Perform live range analysis using the re-constituted intermediate code. A variable is live from the time it is defined, i.e. assigned a value, until the last time that its value is used as an operand.

III. Using the live range analysis, construct an interference graph where the nodes in the graph represent the symbolic registers and the undirected edges of the graph represent interferences between the symbolic registers, i.e. two objects interfere with each other if they are live at the same time and hence cannot be placed into the same register simultaneously.

IV. Color the interference graph's nodes with $n$ colors where $n$ is the number of available registers in the target machine so that any two adjacent nodes have different colors (this is called an $n$-coloring). Often we seek a minimum $n$.

V. Allocate each object to the register that has the same color that it does.

I. Assign Each Object

In this first phase we assign each object (variable, constant or intermediate result) to a symbolic register. Let's consider the following sequence of 4-tuples where $a$, $d$, and $e$ are live after the code segment:

1. $I$\$1 + 5 a$
2. $I$\$2 / I$\$1 5$
3. $d = I$\$2$
4. $I$\$3 + I$\$1 d$
5. $d = I$\$3$
6. $I$\$4 + a d$
7. $e = I$\$4$

After assigning the objects to symbolic registers, we have

1.

2.

3.

4.

5.

6.

7.

8.

Click here for the answer.
II. Live Range Analysis

The second step is to determine the live range analysis for the code segment under consideration. The live range of an object starts when it is defined and ends at the last usage of its value. Typically live range analysis is done for a complete program module, i.e. a procedure, function, etc. Live range analysis for this example yields

1. \( s_1 = 5 \)

2. \( s_2 = a \)

3. \( s_3 + s_1 s_2 \)

4. \( s_4 / s_3 s_1 \)

5. \( s_4 + s_3 s_4 \)

6. \( s_5 + s_2 s_4 \)

Click here for the answer.

III. Interference Graph Construction

The third step in this process is to construct an interference graph from the live range completed in step 2. We do this by considering the nodes of the interference graph to be the objects in the code segment under consideration where the edges represent a conflict between the two objects, i.e. where the two objects have to be in a register simultaneously. Repeating the live range analysis from above, the interference graph constructed is

1. \( s_1 = 5 \)

2. \( s_2 = a \)

3. \( s_3 + s_1 s_2 \)

4. \( s_4 / s_3 s_1 \)

5. \( s_4 + s_3 s_4 \)

6. \( s_5 + s_2 s_4 \)

Click here for the interference graph.
IV. Graph Coloring

The heuristic approach to color the graph that we will use is called the degree < n rule, i.e. given a graph that contains a node with degree less than n, the graph is n-colorable if and only if the graph without that node is n-colorable. Why? This coloring approach is attributed to Chaitin (G.J. Chaitin, M.A. Auslander, A.K. Chandra, J. Cock, M.E. Hopkins, and P.W. Markstein, Register Allocation via Coloring, Computer Languages 6, January, 1981 and G.J. Chaitin, Register Allocation and Spilling via Graph Coloring. Proceedings of the SIGPLAN '82 Symposium on Compiler Construction, SIGPLAN Notices 17(6), June, 1982).

This coloring approach has two phases: the first phase simplifies the graph and produces a stack of nodes, the second phase assigns colors to the nodes. The simplification phase repeats the following steps until the graph is empty.

1. If there is exists a node with degree < n, then remove that node and all of its edges from the graph. Push the node on the stack for later coloring.

2. If no such node exists, choose a node (live range) to spill. In spilling a node, instructions are inserted into the intermediate result code to store to memory after each definition and to restore before each use. This additional code may result in greater register pressure resulting in more spilled nodes. After the node is spilled and the new code inserted, return to Step II (Live Range Analysis) and proceed.

In step 2 we need to decide which live range to spill. There have been many heuristics proposed. One of the most effective is taking the live range with the lowest ratio of spill cost to degree. The spill cost is approximated as the number of loads and stores that would have to be inserted appropriately weighted by the loop nesting depth of each insertion point.

After successfully removing all nodes from the interference graph, the second phase then colors the graph. This coloring process must succeed because of the work done in the first phase, i.e. the simplification phase, ensures this fact. The coloring phase starts with the nodes in the stack and proceeds as

a. Remove a node from the stack and reinsert it in the graph along with all of its edges.

b. Assign a color to that node that differs from all of its neighbors.

Suppose that we apply this technique to our example by trying a 3-coloring. Our interference graph below is used to construct the following stack of nodes:

![Interference Graph]
III.a Simplification Phase

Applying the simplification phase for a 3-coloring, we might remove the nodes in the following order: \( S_1, S_3, S_2, S_4, S_5 \). \( S_1 \) is removed because it has 2 edges (2 < 3) and pushed onto the stack. Node \( S_3 \) is removed because it now has 2 edges (one of the edges was removed when \( S_1 \) was removed). \( S_3 \) is pushed onto the stack. The stack now is

\[
S_3 \quad S_1
\]

\( S_2 \) could be removed next because it now has 2 edges (one was removed with \( S_1 \) and one with \( S_3 \)). The process continues until we have

\[
S_5 \quad S_4 \quad S_3 \quad S_2 \quad S_1
\]
as the stack.

**Note:** The order of removing the nodes is based on the heuristic degree < n rule. There are many valid orders for removing the nodes.

III.b Coloring Phase

In the coloring phase the node \( S_5 \) will be inserted into the graph first. Let's give node \( S_5 \) the color red. Next we have node \( S_4 \) added to the graph. node \( S_4 \) can be any color other than red. Let's give it the color blue. We continue this process until we have the colored graph below:

Click [here](#) for the colored graph.

V. Assigning Objects to Registers

We have seen that the above graph is 3-colorable. We would next assign each of the 3 registers in our machine to one of the colors above. For example, \( S_3 \) and \( S_5 \) would be assigned to register 1, \( S_1 \) and \( S_4 \) would be assigned to register 2, and \( S_2 \) would be assigned to register 3.

Repeating the symbolic code from step one, we see that we have the following assignments of the original entities to their respective registers:
1. \( s_1 = 5 \) mappings

2. \( s_2 = a \)

3. \( s_3 + s_1 s_2 \quad s_3 = \$1 \)

4. \( s_4 / s_3 s_1 \quad s_4 = \$2, d, \$3 \)

5. \( s_4 + s_3 s_4 \)

6. \( s_5 + s_2 s_4 \quad s_5 = \$4, e \)

In this case we have \$1, \$4, and e assigned to register 1. We have 5, \$2, \$3, and d all assigned to register 2, and we have a assigned to register 3. We know from the coloring approach that none of these entities will have to be spilled from a register and that we have the best usage of registers possible with this code.

**Note:** There are graphs that are n-colorable but not by the heuristic \( \text{degree} < n \) rule. For example

(is 2-colorable but not by the degree < n rule. We note that Preston Briggs in his PhD research at Rice University has identified techniques that will solve the above situation. Look at P Briggs, K Cooper, L Torczon, Improvements to Graph Coloring Register Allocation, *ACM Transactions on Programming Languages and Systems* 16, 3, 428-455, 1994. Preston’s approach is that when there is no node that satisfies the heuristic \( \text{degree} < n \) rule, you cut nodes where the degree = n. This heuristic is based on the idea that if the nodes are not connected, then you can n-color the nodes. This heuristic fails when you pop a node from the stack and you cannot color it. At this point you would choose a live range to spill and repeat the coloring process. The following coloring shows that the graph is 2-colorable.)
The resulting symbolic code is

1. \( s_1 = 5 \) mappings
2. \( s_2 = a \)
3. \( s_3 + s_1 s_2 \) \( s_3 = I$1 \)
4. \( s_4 / s_3 s_1 \) \( s_4 = I$2, d, I$3 \)
5. \( s_4 + s_3 s_4 \)
6. \( s_5 + s_2 s_4 \) \( s_5 = I$4, e \)

Return

The local live range analysis is

1. \( s_1 = 5 \)
2. \( s_2 = a \)
3. \( s_3 + s_1 s_2 \)
4. \( s_4 / s_3 s_1 \)
5. \( s_4 + s_3 s_4 \)
6. \( s_5 + s_2 s_4 \)

Return

The resulting interference graph is
The 3-colored graph is