Example

```c
int p(int lim1, int lim2) {
    for (i=0; i<lim1 && A[i]>0; i++){}
    for (j=0; j<lim2 && B[j]>0; j++){}
    return i+j;
}
```

We optimize array accesses by placing \&A[0] and \&B[0] in temporaries:

```c
int p(int lim1, int lim2) {
    int *T1 = \&A[0];
    for (i=0; i<lim1 && *(T1+i)>0; i++){}
    int *T2 = \&B[0];
    for (j=0; j<lim2 && *(T2+j)>0; j++){}
    return i+j;
}
```

Register Allocation via Graph Coloring

We model global register allocation as a Coloring Problem on the Interference Graph

We wish to use the fewest possible colors (registers) subject to the rule that two connected nodes can’t share the same color.

Optimal Graph Coloring is NP-Complete

Reference:

“Computers and Intractability,”
M. Garey and D. Johnson,

We’ll use a Heuristic Algorithm originally suggested by Chaitin et. al. and improved by Briggs et. al.

References:

“Register Allocation Via Coloring,”
G. Chaitin et. al., Computer Languages, 1981.

“Improvement to Graph Coloring Register Allocation,” P. Briggs et. al., PLDI, 1989.

Coloring Heuristic

To R-Color a Graph (where R is the number of registers available)

1. While any node, n, has < R neighbors:
   Remove n from the Graph.
   Push n onto a Stack.

2. If the remaining Graph is non-empty:
   Compute the Cost of each node.
   The Cost of a Node (a Live Range) is the number of extra instructions needed if the Node isn’t assigned a register, scaled by $10^{loop\_depth}$.
   Let $NB(n) = Number\ of\ Neighbors\ of\ n$.
   Remove that node n that has the smallest Cost(n)/NB(n) value.
Push n onto a Stack.
Return to Step 1.

3. While Stack is non-empty:
   Pop n from the Stack.
   If n’s neighbors are assigned fewer than R colors
   Then assign n any unassigned color
   Else leave n uncolored.

Example

```
int p(int lim1, int lim2) {
  int *T1 = &A[0];
  for (i=0; i<lim1 && *(T1+i)>0;i++){}
  int *T2 = &B[0];
  for (j=0; j<lim2 && *(T2+j)>0;j++){}
  return i+j;
}
```

<table>
<thead>
<tr>
<th>lim1</th>
<th>lim2</th>
<th>T1</th>
<th>T2</th>
<th>i</th>
<th>j</th>
<th>Cost</th>
<th>Cost/Neighbors</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>42</td>
<td>42</td>
<td>11/3</td>
<td>42/5</td>
</tr>
</tbody>
</table>

Since no node has fewer than 3 neighbors, we remove a node based on the minimum Cost/Neighbors value.

lim2 is chosen.
We now have:

Remove (say) lim1, then T1, T2, j and i (order is arbitrary).

The Stack is:

Assuming the colors we have are R1, R2 and R3, the register assignment we choose is

i:R1, j:R2, T2:R3, T1:R2, lim1:R3, lim2:spill
Color Preferences

Sometimes we wish to assign a particular register (color) to a selected Live Range (e.g., a parameter or return value) if possible.

We can mark a node in the Interference Graph with a Color Preference.

When we unstack nodes and assign colors, we will avoid choosing color c if an uncolored neighbor has indicted a preference for it. If only color c is left, we take it (and ignore the preference).

Example

Assume in our previous example that lim1 has requested register R1 and lim2 has requested register R2 (because these are the registers the parameters are passed in).

Using Coloring to Optimize Register Moves

A nice “fringe benefit” of allocating registers via coloring is that we can often optimize away register to register moves by giving the source and target the same color.

Consider

Live in: a, b
t1 = a + b
x = t1
y = x + 1
q = t1
Live out: y, q

We’d like x, t1 and q to get the same color. How do we “force” this?
We can “merge” $x$, $t_1$, and $q$

Live in: $a, b$

$t_1 = a + b$

$x = t_1$

$y = x + 1$

$q = t_1$

Live out: $y, q$

together:

Now a 2-coloring that optimizes away both register to register moves is trivial.

---

**Reckless Coalescing**

Originally, Chaitin suggested merging all move-related nodes that don’t interfere.

This is *reckless*—the merged node may not be colorable!

(Is it worth a spill to save a move??)

---

**Iterated Coalescing**

This is an intermediate approach, that seeks to be safer than reckless coalescing and more effective than conservative coalescing. It was proposed by George and Appel.

---

**Reading Assignment**

- Read George and Appel’s paper, “Iterated Register Coalescing.” (Linked from Class Web page)
- Read Larus and Hilfinger’s paper, “Register Allocation in the SPUR Lisp Compiler.”
1. Build:
   Create an Interference Graph, as usual. Mark source-target pairs with a special move-related arc (denoted as a dashed line).

2. Simplify:
   Remove and stack non-move-related nodes with < R neighbors.

3. Coalesce:
   Combine move-related pairs that will have < R neighbors after coalescing.

Repeat steps 2 and 3 until only nodes with R or more neighbors or move-related nodes remain or the graph is empty.

4. Freeze:
   If the Interference Graph is non-empty:
   Then If there exists a move-related node with < R neighbors
   Then: “Freeze in” the move and make the node non-move-related.
   Return to Steps 2 and 3.
   Else: Use Chaitin’s Cost/Neighbors criterion to remove and stack a node.
   Return to Steps 2 and 3.

5. Unstack:
   Color nodes as they are unstacked as per Chaitin and Briggs.

Example
Live in: k, j
g = mem[j+12]
h = k-1
f = g*h
e = mem[j+8]
m = mem[j+16]
b = mem[f]
c = e+8
d = c
k = m+4
j = b
goto d
Live out: d, k, j

Assume we want a 4-coloring.
Note that neither j & b nor d & c can be conservatively colored.

We simplify by removing nodes with fewer than 4 neighbors.
We remove and stack: g, h, k, f, e, m
The remaining Interference Graph is

We can now conservatively coalesce the move-related pairs to obtain

These remaining nodes can now be removed and stacked.

We can now unstack and color:

d&c:R1, j&b:R2, m:R3, e:R4, f:R1, k:R3, h:R1, g:R4

No spills were required and both moves were optimized away.

Priority-Based Register Allocation

Alternatives to Chaitin-style register allocation are presented in:


These papers suggest two innovations:

1. Use of a Priority Value to choose nodes to color in an Interference Graph.

   A Priority measures 
   (Spill cost)/(Size of Live Range)

   The idea is that small live ranges with a high spill cost are ideal candidates for register allocation. As the size of a live range grows, it becomes less attractive for register allocation (since it “ties up” a register for a larger portion of a program).

2. Live Range Splitting

   Rather than spill an entire live range that can’t be colored, the live range is split into two or more smaller live ranges that may be colorable.
Large vs. Small Live Ranges

- A large live range has less spill code. Values are directly read from and written to a register. But, a large live range is harder to allocate, since it may conflict with many other register candidates.
- A small live range is easier to allocate since it competes with fewer register candidates. But, more spill code is needed to load and save register values across live ranges.
- In the limit a live range can shrink to a single definition or use of a register. But, then we really don’t have an effective register allocation at all!

Terminology

In an Interference Graph:

- A node with fewer neighbors than colors is termed unconstrained. It is trivial to color.
- A node that is not unconstrained is termed constrained. It may need to be split or spilled.

PriorityRegAlloc(proc, regCount) {
    ig ← buildInterferenceGraph(proc)
    unconstrained ← { n ∈ nodes(ig) | neighborCount(n) < regCount }
    constrained ← { n ∈ nodes(ig) | neighborCount(n) ≥ regCount }

    while( constrained ≠ ∅ ) {
        for ( c ∈ constrained such that not colorable(c) and canSplit(c) ) {
            c1, c2 ← split(c)
            constrained ← constrained - {c}
            if ( neighborCount(c1) < regCount )
                unconstrained ← unconstrained U {c1}
            else constrained ← constrained U {c1}
            if ( neighborCount(c2) < regCount )
                unconstrained ← unconstrained U {c2}
            else constrained ← constrained U {c2}
        }
        for ( d ∈ neighbors(c) such that d ∈ unconstrained and neighborCount(d) ≥ regCount ){
            unconstrained ← unconstrained - {d}
            constrained ← constrained U {d}
        }
    } // End of While
    color all nodes ∈ unconstrained
}

/* At this point all nodes in constrained are colorable or can’t be split */

Select p ∈ constrained such that priority(p) is maximized
if ( colorable(p) )
    color(p)
else spill(p)
} // End of While

How to Split a Constrained Node

- There are many possible partitions of a live range; too many to fully explore.
- Heuristics are used instead. One simple heuristic is:
  1. Remove the first basic block (or instruction) of the live range. Put it into a new live range, NR.
  2. Move successor blocks (or instructions) from the original live range into NR, as long as NR remains colorable.
  3. Single Basic Blocks (or instructions) that can’t be colored are spilled.

Example

```c
int sum(int a[], int b[]) {
    int sum = 0;
    for (int i=0; i<1000; i++)
        sum += a[i];
    for (int j=0; j<1000; j++)
        sum += b[j];
    return sum;
}
```

Assume we want a 3-coloring.

We first simplify the graph by removing unconstrained nodes (those with < 3 neighbors).

Node j is removed. We now have:

At this point, each node has 3 neighbors, so either spilling or splitting is necessary.

A spill really isn’t attractive as each of the 4 register candidates is used within a loop, magnifying the costs of accessing memory.

Coloring by Priorities

We’ll color constrained nodes by priority values, with preference given to large priority values.
Variables i, sum and a are assigned colors R1, R2 and R3.

Variable b can’t be colored, so we will try to split it. b’s live range is blocks 1 to 6, with 1 as b’s entry point.

Blocks 1 to 3 can’t be colored, so b is spilled in block 1. However, blocks 4 to 6 form a split live range that can be colored (using R3).

We will reload b into R3 in block 4, and it will be register-allocated throughout the second loop. The added cost due to the split is minor—a store in block 1 and a reload in block 4.

### Choice of Spill Heuristics

We have seen a number of heuristics used to choose the live ranges to be spilled (or colored).

These heuristics are typically chosen using one’s intuition of what register candidates are most (or least) important. Then a heuristic is tested and “fine tuned” using a variety of test programs.

Recently, researchers have suggested using machine learning techniques to automatically determine effective heuristics.

In “Meta Optimization: Improving Compiler Heuristics with Machine Learning,” Stephenson, Amarasinghe, et al, suggest using genetic programming techniques in

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>sum</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>11</td>
<td>42</td>
<td>41</td>
</tr>
<tr>
<td>Cost</td>
<td>Cost/Size</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11/3</td>
<td>11/6</td>
<td>42/7</td>
<td>41/3</td>
</tr>
</tbody>
</table>