Register Allocation

Assign an unbounded number of temporaries to a fixed number of registers.

Example:

```
-------------
load a, %r1
load b, %r2
sub %r1, %r2, %r1
load c, %r3
add %r2, %r3, %r2
load d, %r3
mul %r1, %r3, %r3
-------------
add %r2, %r3, %r2
mul %r1, %r2, %r1
sub %r2, %r1, %r1
```
Register Allocation Approaches

- **A Naive First-Come-First-Serve Approach** — Treat all registers equal; allocate registers along code-generation; first-come-first-serve; if run out of registers, dump some to memory. Just keep track of the usage of each register and handle requests; very simple and fast.

- **Category-Based Approach** — Classify program values into categories (i.e. basic addresses, arithmetic computations, etc.); and assign a group of registers to each category. From there, follow the naive approach.

But these approaches are in general not good enough. In allocating registers, we have an optimization goal: to minimize memory operations (loads & stores) and register moves, which motivates the following approach:

- **Usage-Based Approach** — Analyze program to collect values-usage information; then try to put (and keep) frequently-used values in registers.

Usage-Based Register Allocation

The usage analysis can be performed at different scope levels:

- **Local Register Allocation** —
  Analyze variable usages in a basic block, and optimize based on the information. It is simple and fast.

- **Register Allocation for Loops** —
  Focus usage analysis on a loop. Need to identify loops; otherwise still simple and fast.

- **Global Register Allocation** —
  Analyze variable usages across a whole procedure (via dataflow analysis), and optimize based on the information.
  Can provide really efficient usage of registers. Very useful for machines with only a small number of registers available.
Local Register Allocation

Consider a single basic block. Assign registers based on usage count of temps. Reserve a few registers (2-3) to handle memory accesses (in cases where the number of registers is smaller than the number of temps).

**Algorithm:**

1. *Compute a priority for each temp* — In a linear pass over the instructions in the block, tally the number of occurrences of each temp. The occurrence count for a temp becomes its priority.
2. *Sort the temps into priority order.*
3. *Assign registers in priority order* — If there are \( k \) registers available, the first \( k \) temps get assigned to them.
4. *Rewrite code* — Walk over the block a second time. Replace each reg-bound temp with the register’s name, and replace other temps with load/store instructions.

**An Example**

Assume two regs are reserved for memory accesses, and two regs are available for assignment.

\[
\begin{align*}
\text{t1} &:= \text{a-b;} \\
\text{t2} &:= \text{c+b;} \\
\text{t3} &:= \text{t1*d;} \\
\text{t4} &:= \text{t2+t3;} \\
\text{t5} &:= \text{t1*t3;} \\
\text{t6} &:= \text{t4-t5;} \\
\end{align*}
\]

**Usage counts:**

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>t1</th>
<th>t2</th>
<th>t3</th>
<th>t4</th>
<th>t5</th>
<th>t6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

**Assignments:**

<table>
<thead>
<tr>
<th>t1</th>
<th>t3</th>
</tr>
</thead>
<tbody>
<tr>
<td>%r3</td>
<td>%r4</td>
</tr>
</tbody>
</table>

**Observation:** Further optimization possible, but needs temps’ liveness information.
Register Allocation for Loops

Usage Counts:

\( \text{use}(x, B) \) — the number of times variable \( x \) is used in basic block \( B \) prior to any definition of \( x \).

\( \text{live}(x, B) \) — is 1 if \( x \) is alive on exit from \( B \) and is assigned a value in \( B \); is 0 otherwise.

\[
\begin{align*}
\text{use} & : \begin{cases}
  a & : 0 \\
  b & : 2 \\
  c & : 1 \\
  d & : 1 \\
  e & : 0 \\
  f & : 1
\end{cases} \\
\text{live} & : \begin{cases}
  a & : 0 \\
  b & : 0 \\
  c & : 1 \\
  d & : 1 \\
  e & : 0 \\
  f & : 0
\end{cases}
\end{align*}
\]

Calculating the Benefits:

The benefit from allocating a register to \( x \) within loop \( L \):

\[
\sum_{B \in L} \text{cost}_{ld} \times \text{use}(x, B) + \text{cost}_{st} \times \text{live}(x, B)
\]

The first term is the cost of addressing value in memory and the second term is the cost of storing value into memory.

Assume \( \text{cost}_{ld} = \text{cost}_{st} = 1 \), then

benefit\((a, L)\) = 3  \hspace{1cm} \text{benefit}(b, L) = 4  \\
benefit\((c, L)\) = 3  \hspace{1cm} \text{benefit}(d, L) = 5  \\
benefit\((e, L)\) = 1  \hspace{1cm} \text{benefit}(f, L) = 3

Assume that we have two registers available for this loop. Then \( b \) and \( d \) would be the winners.
RegAlloc for Loops (cont.)

Register Assignment: b and d are kept in registers.

Register Allocation by Coloring

- **Perform dataflow liveness analysis over a CFG:**
  - collect variable liveness info at every program point

- **Build an interference graph:**
  - each node represents a temporary value
  - each edge represents an interference between two temporaries—they can’t be assigned to the same register

- **Color the interference graph:**
  - colors = registers; want to use as few colors as possible
  - for a machine with K registers, decide whether the interference graph is K-colorable
  - if yes, the coloring gives a register assignment
  - if no, need to **spill** some temporaries to memory

Can be used either at the local scope or at the global scope.
**An Example**

- **Liveness Analysis Result and Interference Graph:**

<table>
<thead>
<tr>
<th>Statement</th>
<th>Live Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1 := a-b;</td>
<td>a, b</td>
</tr>
<tr>
<td>t2 := c+b;</td>
<td>t1, c, b</td>
</tr>
<tr>
<td>t3 := t1*d;</td>
<td>t1, t2, d</td>
</tr>
<tr>
<td>t4 := t2+t3;</td>
<td>t1, t2, t3</td>
</tr>
<tr>
<td>t5 := t1*t3;</td>
<td>t1, t3, t4</td>
</tr>
<tr>
<td>t6 := t4-t5;</td>
<td>t4, t5</td>
</tr>
</tbody>
</table>

- **Coloring — How many colors do we need?**

  **Two Problems Remaining:**
  - K-colorability problem is NP-complete!
  - How to handle spilling, if needed?

---

**Coloring by Simplification**

To overcome the NP-complete problem, a *heuristic* algorithm is used.

**Fact** — Let $m$ be a node in $G$ with fewer than $K$ neighbors; let $G'$ be the graph $G - \{m\}$. If $G'$ is $K$-colorable, then so is $G$.

A *Heuristic Coloring Algorithm*: Repeatedly simplify the graph by removing (and push on a stack) nodes of degree less than $K$; until either (1) no more node is left, which means the graph is $K$-colorable; or (2) all remaining nodes are of degree more than $K$, which means the graph is not $K$-colorable.

**Example:** Is the first graph 3-colorable?

Note that this algorithm may give wrong answers: Is the second graph 2-colorable?
A Coloring Example

<table>
<thead>
<tr>
<th>Statement</th>
<th>Live Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>$g := \text{mem}[j+12]$</td>
<td>$k \ j$</td>
</tr>
<tr>
<td>$h := k - 1$</td>
<td>$k \ g \ j$</td>
</tr>
<tr>
<td>$f := g * h$</td>
<td>$g \ h \ j$</td>
</tr>
<tr>
<td>$e := \text{mem}[j+8]$</td>
<td>$j \ f$</td>
</tr>
<tr>
<td>$m := \text{mem}[j+16]$</td>
<td>$j \ f \ e$</td>
</tr>
<tr>
<td>$b := \text{mem}[f]$</td>
<td>$f \ e \ m$</td>
</tr>
<tr>
<td>$c := e + 8$</td>
<td>$e \ m \ b$</td>
</tr>
<tr>
<td>$d := c$</td>
<td>$c \ m \ b$</td>
</tr>
<tr>
<td>$k := m + 4$</td>
<td>$m \ b \ d$</td>
</tr>
<tr>
<td>$j := b$</td>
<td>$b \ d \ k$</td>
</tr>
<tr>
<td></td>
<td>$d \ j \ k$</td>
</tr>
</tbody>
</table>

A Coloring Example (cont.)

Let $K = 4$.

Remove nodes $c$, $f$, $g$, and $h$:  
Remove nodes $e$, $j$, $k$, and $m$:

All remaining nodes have a degree less than 4. The graph is 4-colorable.
Spilling

If there are $k$ registers but the interference graph is not $k$-colorable, we need to spill temporaries to stack.

The Spilling Action:
- After the temp’s definition, it is stored to stack.
- Before each use, the temp needs to be loaded back to a reg.

Effects of Spilling:
- To handle data on stack, 2 or 3 registers are need.
- The node corresponding to the spilled temp can be removed from the interference graph.

How to Select a Variable:
- Choose least frequently accessed variables.
- Choose nodes with the most conflicts in the interference graph.

Register Allocation with Coloring (Chaitin’s Algorithm)

1. Build — Construct the interference graph.
2. Simplify — Repeatedly simplify the graph.
3. Spill — When no more nodes can be simplified, choose a node to spill; add it to the spill set, and remove it from the graph. Repeat Steps 2&3 until no more nodes left in the graph.
4. Start Over — If the spill set is not empty, insert spill code for all spilled nodes (i.e. load before each use, store after each definition); reconstruct the interference graph. Repeat the algorithm on this rewritten program.
5. Select — Assign colors to nodes. Starting with an empty graph, repeatedly adding a node from the top of the stack. There should always be a new color available.
An Example

Delaying Spilling [Brigg]

Recall that when the heuristic algorithm says that a graph is not \(k\)-colorable, it may not be true. When the algorithm gets stuck in the simplification step, we do not have to spill right away.

1. **Build** — Construct the interference graph.
2. **Simplify** — Repeatedly simplify the graph.
3. **Spill** — When no more nodes can be simplified, choose a node for potential spill; remove it from the graph and push it on the stack.
   
   Repeat Steps 2&3 until all nodes are on the stack.
4. **Select** — Assign colors to nodes. Starting with an empty graph, repeatedly adding a node from the top of the stack. For a simplified node, there is always a new color available; for a potential spill node, there may still be a new color available.
5. **Start Over** — If the Select step is unable to find a color for some node(s), then the program must be rewritten to fetch them from memory just before each use, and store them back after each definition. The algorithm is repeated on this rewritten program.
Node Coalescing

Fact — If there is no edge in the interference graph between the source and destination of a MOVE instruction, then the source and destination nodes can be coalesced into a new node whose edges are the union of the two. (Effectively delete the MOVE instruction.)

In principle, any pair of nodes not connected by an interference edge could be coalesced. However, coalescing can make a $K$-colorable graph not $K$-colorable.

Conservative Coalescing Strategies — Guarantees that coalescing will preserve the graph’s $k$-colorability property.

- Briggs — Nodes $a$ and $b$ can be coalesced if the resulting node $ab$ will have fewer than $k$ neighbors of significant degree (i.e. having $\geq K$ edges).
- George & Appel — Nodes $a$ and $b$ can be coalesced if, for every neighbor $t$ of $a$, either $t$ already interferes with $b$ or $t$ is of insignificant degree.

Register Allocation with Coloring (Improved Version)

1. Build — Construct the interference graph, and categorize each node as either move-related or non-move-related.
2. Simplify — Repeatedly simplify the graph by removing non-move-related nodes of $< K$ degree.
3. (Conservative) Coalesce — Coalesce two nodes according to Briggs’s or George&Appel’s conditions.
   Repeat Simplify and Coalesce steps until only $\geq K$-degree or move-related nodes remain.
4. Freeze — Look for a move-related node of low degree, and freeze the moves, i.e. giving up future coalesce attempts on them. Resume simplify and coalesce.
5. Spill — If there is no low-degree nodes, select a high-degree node for potential spill; remove it from the graph and push it on the stack.
6. Select — Pop the entire stack, assigning colors.
7. Start Over — If an actual spill occurs, rewrite the program and rerun the algorithm.
Back to the Example

Node \( j \) and \( b \) are MOVE related, so are \( d \) and \( c \).

Remove nodes \( f \), \( g \), and \( h \):  

Remove nodes \( e \) and \( k \), and then \( m \):

Coalesce nodes \( j \) & \( b \), and \( d \) & \( c \).

Final Coloring

<table>
<thead>
<tr>
<th></th>
<th>stack</th>
<th>color</th>
</tr>
</thead>
<tbody>
<tr>
<td>( d &amp; c )</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>( j &amp; b )</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>( m )</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>( k )</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>( e )</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>( h )</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>( g )</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>( f )</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>
Pre-Colored Nodes

In modeling a realistic register allocation, some nodes in the interference graph need to be *pre-colored*, i.e. assigned to specific machine registers (for example, the frame pointer register or the return value register).

- Pre-colored nodes cannot be simplified, nor spilled.
- The original register allocation algorithm can be modified to work with pre-colored nodes — just leave pre-colored nodes to the end.
- Pre-colored nodes can also make an otherwise $K$-colorable graph not $K$-colorable.

Optimizations with Pre-Colored Nodes

- **Creating temporary copies of pre-colored registers:**
  
  ```
  enter: def(r7)       enter: def(r7)
  ...                r231 <- r7
  ... =>            ...
  ...               r7 <- r231
  exit: use(r7)       exit: use(r7)
  ```

  These copies will shorten the live range of pre-colored registers. The extra move instructions will automatically be removed if they are not actually needed.

- **Distinguishing caller-save and callee-save registers:**

  Force a `def` at the entry of the subroutine and a `use` at the end for every callee-save registers.
Example

int f(int a, int b)
{
    int d=0, e=a;
    do { d = d+b;
         e = e-1;
    } while (e>0);
    return d;
}

enter: c ← r3
a ← r1
b ← r2
d ← 0
e ← a

loop: d ← d + b
      e ← e - 1
if e > 0 goto loop
r1 ← d
r3 ← c
return (r1, r3 live out)

No opportunity for simplify, freeze, or coalesce, so we must spill. The priority for spilling can be computed: c, a, b, d, e. So we select node c for spilling.

Example (cont.)

Now we can coalesce a and e.

Two possible coalescing pairs: ae&r₁ or d&r₂. Choose the latter.

Coalesce ae&r₁.

The graph can now be fully simplified.
Since there is a spilling, we need to modify the program:

```
enter: c1 ← r3
       M[loc] ← c1
       a ← r1
       b ← r2
       d ← 0
       e ← a
loop:  d ← d + b
       e ← e − 1
       if e > 0 goto loop
       r1 ← d
       c2 ← M[loc]
       r3 ← c2
return
```

This graph can be fully simplified/coalesced. The final coloring for the nodes are:

<table>
<thead>
<tr>
<th>node</th>
<th>a</th>
<th>b</th>
<th>c1</th>
<th>c2</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>color</td>
<td>r1</td>
<td>r2</td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
<td>r1</td>
</tr>
</tbody>
</table>

The resulting program:

```
enter: r3 ← r3
       M[loc] ← r3
       r1 ← r1
       r2 ← r2
       r3 ← 0
       r1 ← r1
loop:  r3 ← r3 + r2
       r1 ← r1 − 1
       if r1 > 0 goto loop
       r1 ← r3
       r3 ← M[loc]
return
```

After removing trivial move instructions:

```
enter: M[loc] ← r3
       r3 ← 0
       r1 ← r1
loop:  r3 ← r3 + r2
       r1 ← r1 − 1
       if r1 > 0 goto loop
       r1 ← r3
       r3 ← M[loc]
return
```