Register allocation
Register allocation

Overview

- Variables may be stored in the main memory or in registers.
  - Main memory is much slower than registers.
  - The number of registers is strictly limited.
- The goal of register allocation is to decrease the number of memory accesses by keeping as many as possible variables in registers.
  - Decides which values to keep in registers and which in memory.
  - Assigns concrete registers for values which are kept there.
Register allocation

Observations

- Usually there are less registers than variables.
- Simultaneously alive variables cannot be allocated to the same register.
- Variables which life times do not overlap can be allocated to the same register.
- These constraints can be represented as an interference graph:
  - nodes are variables;
  - edges are between simultaneously alive variables.
- Register allocation can be stated as a graph coloring problem of the interference graph with $k$ colors (Lavrov 1962, Chaitin 1981)
  - $k = \text{the number of registers}$. 
Register allocation

\[ b = a + 2 \]
\[ c = b \times b \]
\[ b = c + 1 \]
\[ \text{return } b \times a \]
Register allocation

```
    b = a + 2
    c = b * b
    b = c + 1
    return b*a
```

Code

<table>
<thead>
<tr>
<th>Live sets</th>
<th>Live ranges</th>
</tr>
</thead>
<tbody>
<tr>
<td>{a}</td>
<td>a</td>
</tr>
<tr>
<td>{a, b}</td>
<td>b</td>
</tr>
<tr>
<td>{a, c}</td>
<td>c</td>
</tr>
<tr>
<td>{a, b}</td>
<td></td>
</tr>
</tbody>
</table>

Live sets

Interference graph

- a
- b
- c
Register allocation

\[
\begin{align*}
  &b = a + 2 \\
  &c = b \times b \\
  &b = c + 1 \\
  &\text{return } b \times a
\end{align*}
\]
Register allocation

Construction of the interference graph

- To build the interference graph we need to determine live ranges of variables.
- In case of local register allocation inside basic blocks, all live ranges are linear.
  - Discovering live ranges and checking whether they overlap is very easy.
  - Variables have to be read from the memory before entering to the basic block, and to be stored to the memory when leaving.
- In case of global register allocation, live ranges form a web.
  - Discovering live ranges is more complex.
  - Allows more efficient usage of registers.
Register allocation

def x
def y

use x
use y

use x
def x

use x
def y

use x

use y
Register allocation
Register allocation
Register allocation
Register allocation

Webs s1 and s2 overlap
Webs s1 and s4 overlap
Graph coloring

**Definition**
A graph $G$ is $k$-colorable iff its nodes can be labeled with integers $1 \ldots k$ so that no edge in $G$ connects two nodes with the same label.

**Main questions**
- How to find efficiently $k$-coloring of a graph?
- Whether and how to find an optimal coloring (ie. a coloring with the minimum number of colors)?
- What to do when there are not enough colors (ie. registers)?
Graph coloring

**Problem**

The graph coloring problem is NP-complete.

**Observations**

- Optimal algorithm works with all graphs.
  - The "worst case graph" doesn’t appear in practice.
- It always finds a minimal coloring.
  - Often, an approximate coloring is enough.
# Graph coloring

## Problem

What to do if the graph is not $k$-colorable?  
- Ie. there is not enough registers?  
- Happens very often.

## Spilling

- Choose a variable and keep its values in the memory (ie. in the stack) instead of an register.  
  - The process is called spilling.  
- Places where the variable is accessed, generate an extra code for reading from and storing to the memory.
Graph coloring

Idea

- Pick a node which has degree $< k$.
  - This node is $k$-colorable!
- Remove the node (and all its edges) from the graph.
  - All its neighbours have now degree decremented by one.
  - May result to new nodes with degree $< k$.
- If all nodes have degree $\geq k$, then pick a node, spill it to the memory, and continue.
Graph coloring

Chaitin’s algorithm

1. Until there are nodes with degree $< k$:
   - choose such node and push it into the stack;
   - delete the node and all its edges from the graph.

2. If the graph is non-empty (and all nodes have degree $\geq k$), then:
   - choose a node (using some heuristics) and spill it to the memory;
   - delete the node and all its edges from the graph.
   - if this results to some nodes with degree $< k$, then go to the step 1;
   - otherwise continue with the step 2.

3. Successively pop nodes off the stack and color them in the lowest color not used by some neighbor.
Example:

Stack

Chaitin’s algorithm
Chaitin’s algorithm

Example:

Stack

1

1

2

3

4

5
Chaitin’s algorithm

Example:

Stack

```
1 2 3 4 5
1 2
```
Chaitin’s algorithm

Example:

Stack

```
4
2
1
```
Chaitin’s algorithm

Example:

```
3
4
2
1
Stack
```

```
1 2 3 4 5
```

Diagram:

```
1 -- 2 -- 5
|    |    |
3    4   |
```

```
2
3
4
5
```
Chaitin’s algorithm

Example:

Stack

<table>
<thead>
<tr>
<th></th>
<th>5</th>
<th>3</th>
<th>4</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
</table>

Colors

1
2
3
4
5

Graph representation:

1 -- 2 -- 5
1 -- 3
1 -- 4

Chaitin’s algorithm

Example:

Stack:

| 3 | 4 | 2 | 1 |

Colors:

- Green
- Red
- Blue
Example:

Stack

Colors
Chaitin’s algorithm

Example:

Stack

Colors
Chaitin’s algorithm

Example:

Stack

Colors
Chaitin’s algorithm

Example:
Graph coloring

Optimistic coloring (Briggs et al)

- If all nodes have a degree $\geq k$, then instead of spilling order the nodes and push them into stack.
  - When taking nodes back from the stack they may still be colorable!

- The following graph is 2-colorable:
Graph coloring

Optimistic coloring (Briggs et al)

- If all nodes have a degree \( \geq k \), then instead of spilling order the nodes and push them into stack.
  - When taking nodes back from the stack they may still be colorable!
- The following graph is 2-colorable:

```
1
\[ \begin{array}{c}
2 \\
\end{array} \]
\[ \begin{array}{c}
1 & 4 \\
\end{array} \]
\[ \begin{array}{c}
3 \\
\end{array} \]
```
Graph coloring

Chaitin-Briggs’i algorithm

1. Until there are nodes with degree \(< k\):
   - choose such node and push it into the stack;
   - delete the node and all its edges from the graph.

2. If the graph is non-empty (and all nodes have degree \(\geq k\)), then:
   - choose a node, push it into the stack, and delete it (together with edges) from the graph;
   - if this results to some nodes with degree \(< k\), then go to the step 1;
   - otherwise continue with the step 2.

3. Pop a node from the stack and color it by the least free color.
   - If there is no free colors, then choose an uncolored node, spill it into the memory, and go to the step 1.
Graph coloring

**Spilling heuristics**

- Choosing a node for spilling is a critical for efficiency.
- Chaitin’s heuristics:
  - to minimize the value of \( \frac{\text{cost}}{\text{degree}} \), where \( \text{cost} \) is a spilling cost and \( \text{degree} \) is a current degree of the node;
  - i.e. choose for spilling a "cheapest" possible node which decreases the degree of most other nodes.
- Alternative popular metrics: \( \frac{\text{cost}}{\text{degree}^2} \).
- Variations:
  - spilling of interference regions;
  - partitioning of live ranges;
  - rematerialization.