Register Allocation

Registers: small number of fast memories, close to processor
Register to register operations fast
Register allocation: Determine which values should be kept in registers at each program point

Usually performed on LIL
Addressing, loads and stores, temps exposed

Candidates for register allocation
Large constants, local and non-local vars, array elements,...

What shouldn’t be allocated in registers?

Global Register Alloc. (Graph Coloring)

Overview
1. Allocate symbolic registers to assignable objects
2. Determine the candidates for allocation
3. Construct the Interference Graph
   Nodes: allocatable objects
   Undirected edges: between nodes that cannot be assigned same reg.
4. Color the IG with R colors, where R = # available regs, so nodes connected by an edge have different colors
5. Allocate objects with the same color to the same real reg.
Global Register Alloc. Example

\[
\begin{align*}
x &:= 2 \\
y &:= 4 \\
w &:= x + y \\
z &:= x + 1 \\
u &:= x \cdot y \\
x &:= 2 \\
S_1 &:= 2 \\
S_2 &:= 1 \\
S_3 &:= S_1 + S_2 \\
S_4 &:= S_1 + 1 \\
S_5 &:= S_1 \cdot S_2 \\
S_6 &:= S_4 \cdot 2
\end{align*}
\]
\(y, \ w\) dead

Global Register Alloc. Example

\[
\begin{align*}
x &:= 2 \\
y &:= 4 \\
w &:= x + y \\
z &:= x + 1 \\
u &:= x \cdot y \\
x &:= 2 \\
S_1 &:= 2 \\
S_2 &:= 1 \\
S_3 &:= 1 + S_2 \\
S_4 &:= S_1 + 1 \\
S_5 &:= S_1 \cdot S_2 \\
S_6 &:= S_4 \cdot 2
\end{align*}
\]
\(y, \ w\) dead
Assignable Objects

Candidates:
- Variable names
- Def-use edge
- Web: maximal union of def-use chains that share common use

Ex. BB

X = 2
Y = 4
w = x + y
z = x + 1
u = x
x = x + 2

Ex. Loop

y = 1
y = 2
y = 3

Advantages?

Interference Graph

Nodes: symbolic registers

Undirected Edges:

X \longrightarrow Y

X and Y simultaneously live, or
Y live at a def of X

Which is better?

Ex.

ai, bi
sim. live
Spilling Registers

May not be possible to put all desirable objects in registers
If cannot, must spill: store value, then later reload from memory
Spilling may decrease # registers needed

Split one node into 2

Ex.  
| def X | def X |
| def Y | store X |
| use Y | def Y |
| def Z | use Y |
| use Z | def Z |
| use X | use Z |

X ——— Y

Z

R-Coloring the ISG

NP-complete problem for R>2
Heuristics needed

Heuristic 1: Remove nodes with degree < R from graph.
Resulting graph is R-colorable iff original was.

Note: graph may be R-colorable, but all nodes have degree R.

Ex

S3 ——— S4

S1 ——— S2

2-colorable

Heuristic 1 may by itself make graph R-colorable, for real R
Put nodes on stack as remove; if graph empty, then R-colorable
R-Coloring the ISG

Heuristic 2: If no nodes left of degree < R, remove node with
\[ \min \text{ spill cost } (N) \]
\[ \text{current deg}(N) \]

Put nodes on stack as remove.
When done, color as pop off the stack

Example:

\[
\begin{array}{c}
S3 \quad S4 \\
| \\
S1 \quad S2 \\
\end{array}
\]

S1 S2 S3 S4

In this case, can color with 2 colors

Coloring and SSA

\[
\begin{array}{c}
\text{Def}(A) \quad \text{Def}(C) \quad \text{Def}(B) \\
\text{Use}(A) \quad \text{Use}(C) \quad \text{Use}(B) \\
\end{array}
\]

\[
\begin{array}{c}
A \quad B \quad C \\
\end{array}
\]

Maxlive = 2, need 3 colors

Maxlive = Maximal number of live variables at any point p

SSA:

\[
\begin{array}{c}
\text{Def}(C1) \quad \text{Def}(C2) \\
\text{Use}(A) \quad \text{Use}(B) \\
C3 := \text{Phi}(C1, C2) \\
\text{Use}(C3) \\
\end{array}
\]

\[
\begin{array}{c}
A \quad B \quad C3 \\
C1 \quad C2 \\
\end{array}
\]

2 colors sufficient!
Coloring and SSA (2)

Can color the IG for SSA with Maxlive colors in polynomial time

*idea: Can use join point to start new color, if needed*

Vf: Coloring IG with minimum # colors is NP-complete

Getting out of SSA:

Useful to do coloring after copies inserted

Code may have changed due to optimization

Ex.  
while (P) do  
read(V)  
W := V + W  
V := 6  
W := V + W  
endwhile  

while (P) do  
read(V'1)  
V 2 := V 1 + W  
V 2 := 6  
W 2 := V 2 + W  
endwhile

Building the Dependence DAG

Need # cycles each resource needed for each instruction

\[ \text{Latency}(t_1, t_2, n) = \# \text{ latency cycles incurred by beginning } t_2 \text{ at } n^\text{th cycle of } t_1 \]

Linear pass over instructions \( t_1 \) in BB.

Determine if any \( t_2 \) conflicts with \( t_1 \).

If \( t_1 \) has no conflicts, then it is a root of the DAG.

If conflict with \( t_2 \), need new edge (labelled with latency).

Ex. (1) \( r_3 := [r_{15}] \)
(2) \( r_4 := [r_{15} + 4] \)
(3) \( r_2 := r_3 + 4 \)
(4) \( i_5 := [r_{12}] \)
(5) \( r_{12} := r_{12} + 4 \)
(6) \( r_6 := r_3 \)
(7) \( [r_{15} + 4](4) := r_3 \)
(8) \( r_5 := r_6 + 2 \)
Instruction Scheduling on Dep. DAG

Any topological order will work.

_Can only execute node when all preds are already executed_

Want to minimize execution time

NP-hard: Need heuristic

Ex.

![Diagram of a directed acyclic graph (DAG) with nodes labeled 1 to 6 and arrows indicating dependencies. The graph shows node 1 dependent on node 2, node 3 dependent on node 1, node 4 dependent on node 2, node 5 dependent on node 4, and node 6 dependent on node 4.]

Delay

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

Schedule

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Exact time of node 6 = 2, all others = 1

When choosing a node, choose the largest delay.