Register Allocation

Important opt. for all platforms

→ part of code generator

(assume: code selection &
scheduling taken care of)

2 models in use

1) all variables start in memory
   promotion to registers

2) all variables start in registers
   (move) demotion to memory
   when there is shortage of
   registers
"in memory" model

\[ x \leftarrow \ldots \text{ (first appearance) } \]

1st region

update "home" location

\[ x \text{ is "unallocated" } \]

2nd region

c.g. must store variable, generate extra code if \( x \) is accessed in 1st region

"in register" model

\[ x \leftarrow \ldots \text{ allocated 1st reg. } \]

'spilling'

save var in memory, find location ... \( x \) is "spilled"

allocated 2nd reg.

all traffic is under control of register file

easier to integrate into c.g.
Issues for register allocator

. which variables are live
. when (where) are the variables live

Two variables \( x, y \) interfere if

. both are initially live (e.g., upon entry into a function)
. both are live at end of a basic block
. one variable \( (x) \) is defined and \( y \) is live at the end of this block

\[ x = y + c \quad \text{with \( x \) dead at the end of this block} \]

\[ x, y \text{ interfere} \]
'interfere' means conflict, i.e. the variables cannot reside in the same register.

Interference graph
nodes: variables
edges: "interfere" relation

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


Kempe [1879]

1) if a node (in the I.C.) has fewer than \( N \) neighbors: can be removed

   (put on a stack for later processing)

   a color can be found eventually

2) removing a node removes edges

   best case: I.C. is empty

   graph can be colored with \( N \) colors, deal with the

   colors...
Example

Stock

d
b
a
e
c
This approach is not guaranteed to work may end up with set of nodes wk \( \geq N \) neighbors

```
Stack

c  

a
b
d

picked by ...
```

"Stack" — remove nodes (charting: random)
works some times — bad in general problem is NP complete
will not work in general

the colors not enough

cheating: remove more nodes (pick at random)
The I.G. captures the worst case - variables interfere everywhere.

a is "split" - broken i range

Real view of a:

no use of a.

a is read & written

a is read & written
After splitting, the I.G. must be rebuilt.

Problem: there exist many options for splitting
- which variables?
- where

Another split decision may yield

I loop... in this case putting $q_1, q_2$ into registers (and not using $q_3$). For the loop a BIG win

Need a "cut" function to decide when and where to split.
I G. ignore the details of the CFG.
Handling of spilling

deals with variables that do not
reside in a register

( remove nodes with $\geq N$ neighbors - they
may not be a register)

- keep extra registers to handle variables not
  (permanently) allocated

- rewrite code with temporary variables
  to save registers allocation

  often, interface graph is smaller

  can be colored

  (may have to spill on the spot)
global register allocation may take 10\%-20\% of compilation time...

There is a market for simpler algorithms.

One attractive solution: Linear Scan register alloc.

Interval \((i, j)\) \(\rightarrow\) live range extends from

\[\text{inst } i \text{ to inst } j\]

\[\text{live range}\]

\[\text{process one instr at a time}\]
Keep out of all "active" live ranges.

(allocated to a register)

Look at inst. n

Rj could be the end of an active live range

A new live range starts

We must free a register.

Good heuristic: pick live range with max "j" (end)

Need to keep active live ranges sorted so that the last one can be found.