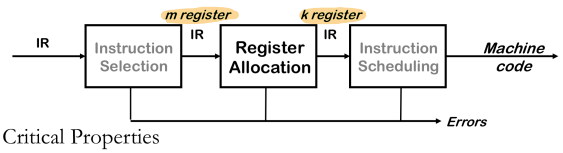
Register Allocation

What is Register Allocation?

Part of the Compiler's Back End



Critical Properties

- Produce Correct Code that Uses *k* (or fewer) Registers
- Minimize Added Loads and Stores
- Minimize Space Used to Hold *Spilled Values*
- Operate Efficiently
  O(n), O(n logn), maybe $O(n^2)$, but not $O(2^n)$

Optimal global allocation is NP-Complete, under almost any assumptions.

At each point of the code:
1. Allocation: Determine which values will reside in registers
2. Assignment: Select a register for each such value

Goal: Allocation that minimizes Running Time

Importance of RA

- *Optimally* Use of one of the Most Critical Processor Resources
  - Affects almost every Statement of the Program
  - Register Accesses are much faster than Memory Accesses
    - Eliminates expensive memory instructions
    - Wider gap in faster newer processors
    - Number of instructions goes down due to direct manipulation of registers (no need for load and store instructions)

Optimization with the most impact! (probably)

Trade-Off:
- Registers: Fast storage with small capacity
- Main Memory: Slow storage with high capacity

- Allocation of Variables (including temporaries) up-to-now stored in Memory to Hardware Registers
  - *Pseudo* or *Virtual* Registers
    - unlimited number of registers
    - space is typically allocated on the stack with the stack frame
  - Hard Registers
    - Set of Registers Available in the Processor
    - Usually need to Obey some Usage Convention

Register Allocation Approaches

- Local Allocators: use Instruction-level knowledge
  - Top-Down: Use Frequency of Variables Use for Allocation
  - Bottom-Up: Evaluate Instructions Needs and Reuse Registers

- Global Allocators: use a Graph-Coloring Paradigm
  - Build a "conflict graph" or "interference graph"
  - Find a *k*-coloring for the graph, or change the code to a nearby problem that can be *k*-colored → k is the number of registers needed

- Common Algorithmic Trade-Off
  - Local Allocators are Fast
  - Some Problems with the Generated Code as they lack more Global Knowledge

Local Register Allocation

code generation for more than a single register is NP-Complete → Hard Problem

Top Down Allocator (Local)

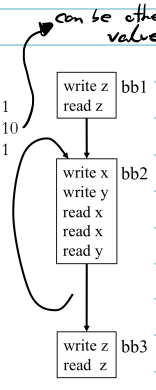
- Estimate the Benefits of Putting each Variable in a Register in a Particular Basic Block
  - Benefit(V, B) = Number of *uses* and *defs* of the var V in basic block B

- Estimate the Overall Benefit
  - TotBenefit(V) = Benefit(V, B)*freq(B) for all basic block B
  - If freq(B) is not known, use 10^{depth} where *depth* represents the nesting depth of B in the CFG of the code.

- Assign the (R-*feasible*) Highest-payoff Variables to Registers
  - Reserve *feasible* registers for basic calculations and evaluation.
  - Rewrite the code inserting load/store operation where appropriate.

Example

- Benefit for basic blocks
  - Benefit(x, bb1) = 0
  - Benefit(x, bb2) = 3
  - Benefit(x, bb3) = 0
  - Benefit(y, bb1) = 0
  - Benefit(y, bb2) = 2
  - Benefit(y, bb3) = 0
  - Benefit(z, bb1) = 2
  - Benefit(z, bb2) = 0
  - Benefit(z, bb3) = 2
- Frequency
  - freq(bb1) = 1
  - freq(bb2) = 10
  - freq(bb3) = 1

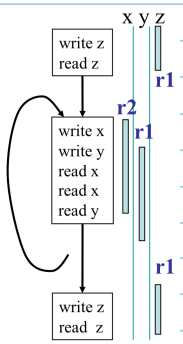


- Total Benefit (bb1 bb2 bb3)
  - TotBenefit(x) = 0*1 + 3*10 + 0*1 = 30
  - TotBenefit(y) = 0*1 + 2*10 + 0*1 = 20
  - TotBenefit(z) = 2*1 + 0*10 + 2*1 = 4

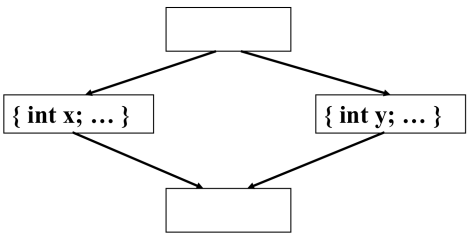
↳ benefit x frequency

Problems referents to the example (Assume that 2 registers are available)

1.
- Allocation is same as above
  - x, and y get registers, but not z
- The variables need to occupy the registers even when it does not need it
- All x, y and z can have registers

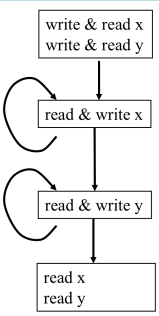


2.
- Even non-interfering variables don't share registers



- x and y can not use the same register

3.
- Different phases of the program behave differently
- One register available
  - Register for x in the first loop
  - Register for y in the second loop
  - Don't care too much about the rest
- Need to spill
  - Top-Down "All or Nothing" will not work



Basic Block

- Def: A *Maximal Sequence* of Instructions such that:
  1. Only the first instruction can be reached from outside the basic block → store of vars to be used
  2. All the instructions are executed consecutively *iff* the first instruction is executed
     - No branch or jump instructions in the basic block
     - Except the last instruction
     - No labels within the basic block
     - Except before the first instruction

Bottom-Up Allocator

- Basic Idea:
  - Focus on the Needs of Each Instructions in a Basic Block
    - Ensure Each Instruction Can Execute
    - Instruction Operands and Results in Registers
  - Transitions Between Instructions
    - Observe Which Values are Used Next - in the Future
- On-Demand Allocation:
  - Iterate Through the Instructions of a ***Basic Block***
  - Allocate the Value of the Operand, if Not Already in a Register
  - Allocate Register for Result
  - When Out of Registers:
- Details:
  - Instructions in format: $vr_z \leftarrow vr_x$ op vr_y using virtual registers or temps
  - Data Structures:
    - The Number of Physical Registers Available
    - The Virtual Name associated with Each Physical Register
    - For Each Virtual Name a Distance to the Next Use in the Basic Block
    - A Flag Indicating if the Corresponding Physical Register is in Free or in Use
    - A Stack of Free Physical Registers with a Stack Pointer (Integer Index)

```
initialize(size)
  Size ← size;
  for i ← size-1 to 0 do
    Name[i] ← -1;
    Next[i] ← ∞;
    Free[i] ← true;
    push(i);
  StackTop = size-1;
```

  - Auxiliary Functions:
    - `ensure(vr_x)` allocates a physical register to ensure storage for vr_x
    - `free(vr_x)` releases the physical register holding vr_x
    - `allocate(vr_x)` just sets some flags claiming register being used.
    - `dist(vr_x)` returns the distance to the next reference to vr_x in the basic block.

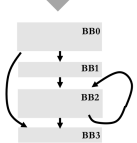
See example of Bottom-Up Allocator ncs slides. (38-54)

Basic Block - Algorithm

- Input: Sequence of Three-Address Instructions
- Output: A list of Basic Blocks
- Algorithm:
  - Determine the set of *leader instructions* - the head of each basic block - using the following:
    - The first statement of the program is a *leader*
    - Any statement that is the target of a goto (either conditional or not) is a *leader instructions*
    - Any statement that immediately follows a goto or unconditional goto statement is a *leader instruction*
  - For each *leader instruction*, its basic block consists of the *leader instruction* and all the statements up to but not including the next *leader instruction* or the end of the program.

Example of Basic Block Algorithm

```
Sum: a = arg0
     b = arg1
     c = 0
     if (a <= 0) goto L2
     i = arg0
L1:  i = i - 1
     a = b[i]
     b = b + 4
     c = c + a
     if (i >= 0) goto L1
L2:  ret c
```



Step 1: Detect Leader Instructions
- The first statement of the program is a *leader*
- Any statement that is the target of a goto (either conditional or not) is a *leader instructions*
- Any statement that immediately follows a goto or unconditional goto statement is a *leader instruction*

Step 2: Determine Block Instructions
- For each *leader instruction*, its basic block consists of the *leader instruction* and all the statements up to but not including the next *leader instruction* or the end of the program

Graph of Basic Blocks

- Define the Control-Flow of a Program
- *More Later…*

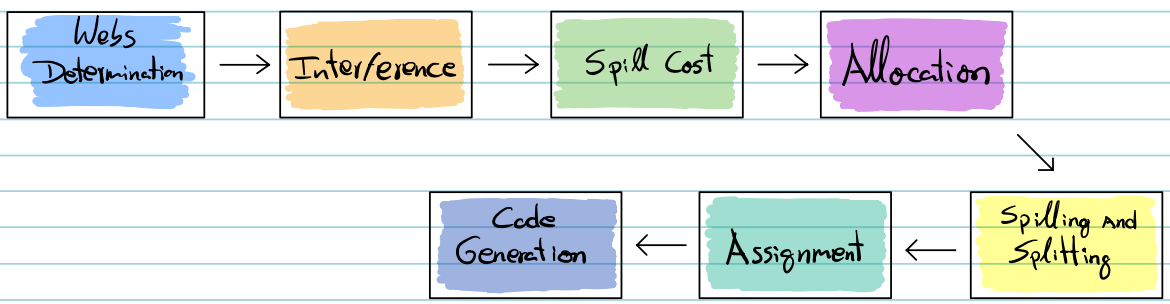
Dirty and Clean Registers

- Choosing which register to reuse:
  - Dirty → need to update memory;
  - Clean → just reuse the physical register.
- Idea: give preference to clean registers → there is no need to save contents to memory.

↳ **Not Always the Best Approach**
- when Dirty value is reused far away, is better to restore it to memory rather than holding on to the register

Smart Allocator Tasks

- Determine ranges, so that each variable can benefit from using a register (Webs);
- Determine which of these ranges overlap (Interference);
- Find the benefit of keeping each web in a register (Spill Cost);
- Decide which webs gets a register (Allocation);
- Split Webs if needed (Spilling and Splitting);
- Assign hard registers to webs (Assignment);
- Generate code including spills (Code Generation).



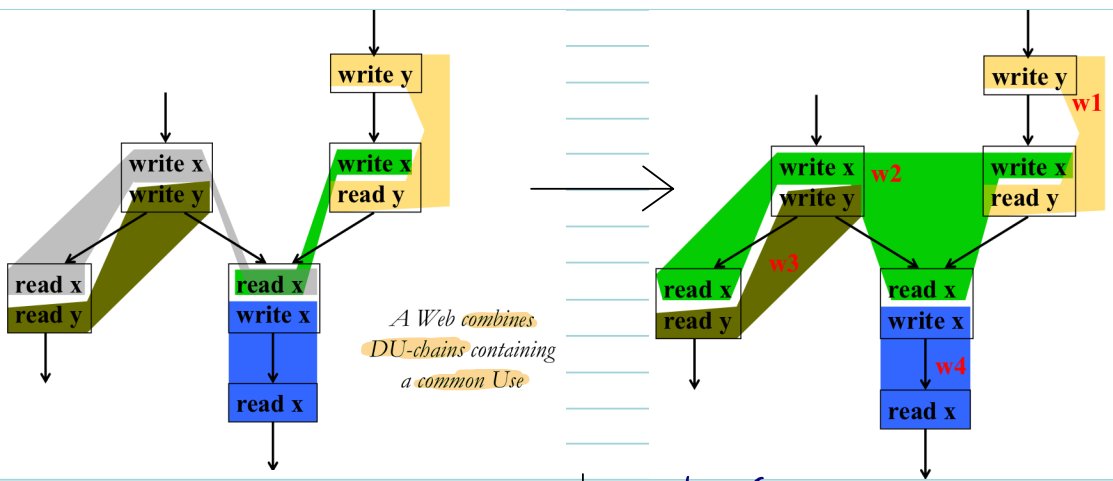
First Summary

- Register Allocation and Assignment
  - Very Important Transformations and Optimization
  - In General Hard Problem (NP-Complete)
- Many Approaches
  - Local Methods: Top-Down and Bottom-Up
  - Quick but not Necessarily Very Good ✗

Global Register Allocation

Webs

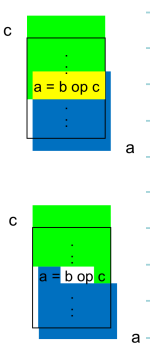
- The Value needs to be memorized.
- Divide Accesses to a Variable into Multiple Webs:
  - All Definitions that Reach a Use are in the same Web;
  - All uses of a Definition are in the same Web;
  - Divide the Variable into Live Ranges.
- Implementation: use DU (Def-Use) chains
  - A DU-chain connects a definition to all uses reached by each definition;
  - A Web combines DU-chains containing a common use.



- In two Webs of the same Variable:
  - No use in one web will ever use a value defined by the other web
  - Thus, no value need to be carried between webs
  - Each web can be treated independently as values are independent
- Web is used as the Unit of Register Allocation
  - If a web is allocated to a register, all the uses and definitions within that web don't need to load and store from memory
  - Solves the issue of cross Basic Block register assignment issue
  - Different webs may be assigned to different registers or one to register and one to memory

Interference

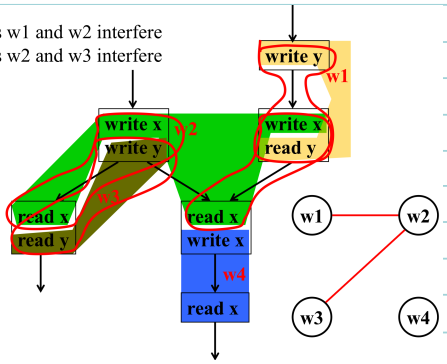
- Two webs interfere if their live ranges overlap in Execution Time
  - What does time Mean, precisely?
  - There exists an instruction common to both ranges where
    - They variable values of webs are operands of the instruction
    - If there is a single instruction in the overlap
      - and the variable for the web that ends at that instruction is an operands and
      - the variable for the web that starts at the instruction is the destination of the instruction
    - then the webs do not interfere



- Non-interfering webs can be assigned to the same register

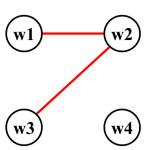
Example

Webs w1 and w2 interfere
Webs w2 and w3 interfere



Interference Graph

- Representation of Webs & their Interference
  - Nodes are the webs
  - An edge exists between two nodes if they interfere



Graph Coloring

→ Classic Problem in Graph Theory

- Each Web is Allocated a Register
  - each node gets a register (color)
- If two webs interfere they cannot use the same register
  - if two nodes have an edge between them, they cannot have the same color

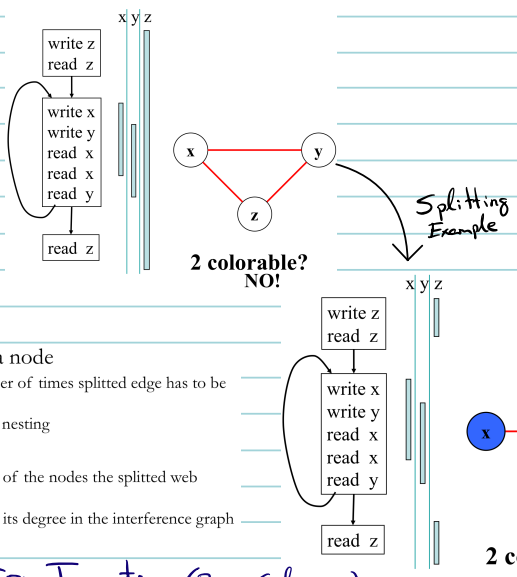
Heuristics for Graph Coloring

- Remove nodes that have degree < N
  - Push the removed nodes onto a stack
- If all the nodes have degree ≥ N
  - Find a node to spill (no color for that node)
  - Remove that node
- When empty, start the coloring step
  - pop a node from stack back
  - Assign it a color that is different from its connected nodes (since degree < N, a color should exist)

- Coloring a Graph with N colors
- If degree < N (degree of a node = # of edges)
  - Node can always be colored
  - After coloring the rest of the nodes, you'll have at least one color left to color the current node
- If degree ≥ N
  - still may be colorable with N colors
  - exact solution is NP complete

Spilling and Splitting

- When the Graph is non-N-colorable
- Select a Web to Spill
  - Find the least costly Web to Spill
  - Use and Defs of that Web are read and writes to memory
- Split the Web
  - Split a web into multiple webs so that there will be less interference in the interference graph making it N-colorable
  - Spill the value to memory and load it back at the points where the web is split



- Cost of splitting a node
  - Proportion to number of times splitted edge has to be crossed dynamically
  - Estimate by its loop nesting
- Benefit
  - \+ Increase colorability of the nodes the splitted web interferes with
  - \+ Can approximate by its degree in the interference graph

Register Coalescing

- Find register copy instructions $s_j = s_i$
- If s_j and s_i do not interfere, combine their webs
- Pros
  - \+ Similar to copy propagation
  - \+ Reduce the number of instructions
- Cons
  - − May increase the degree of the combined node (>N)
  - − A colorable graph may become non-colorable

Register Targeting (Pre-Coloring)

- Some Variables need to be in Special Registers at Specific Points in the Execution
  - first 4 arguments to a function
  - return value
- Pre-color those webs and bind them to the appropriate register
- Will eliminate unnecessary copy instructions

Pre-Splitting of the Webs

- Some Ranges have Very Large "dead" Regions
  - Large region where the variable is unused
- Break-up the Ranges
  - need to pay a small cost in spilling
  - but the graph will be very easy to color
- Can find Strategic Locations to Break-up
  - at a call site (need to spill anyway)
  - around a large loop nest (reserve registers for values used in the loop)

Inter-Procedural Register Allocation

- Saving Registers across Procedure boundaries is expensive
  - especially for programs with many small functions
- Calling convention is too general and inefficient
- Customize calling convention per function by doing inter-procedural register allocation

Second Summary

- Register Allocation and Assignment
  - Very Important Transformations and Optimization
  - In General Hard Problem (NP-Complete)
- Many Approaches
  - Local Methods: Top-Down and Bottom-Up
  - Global Methods: Graph Coloring
    - Webs
    - Interference Graphs
    - Coloring
  - Other Transformations → last 4 sections (optimizations)