Grouping Register Candidates

We now have an estimate of the benefit of allocating a register to a candidate. Call this estimate

cost(candidate)

We estimate potential interprocedural sharing of register candidates by assigning each candidate to a *Group*.

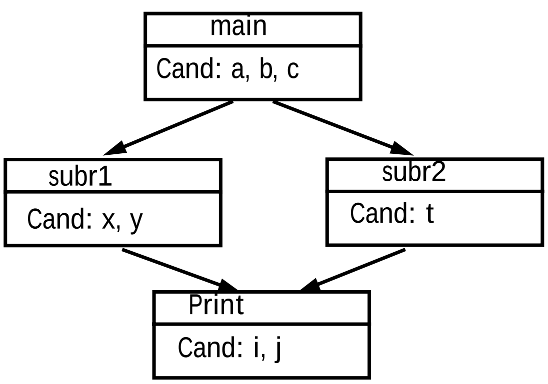
All candidates within a group can share a register. No two candidates in any subprogram are in the same group.

Groups are assigned during a reverse depth-first traversal of the call graph.

```
AsgGroup(node n) {
   if (n is a leaf node)
      grp = 0
   else { for (each c ∈ children(n)) {
            AsgGroup(c) }
      grp =
          1+ Max (Max group used in c)
          c ∈ children(n)
   }
   for (each r ∈ registerCandidates(n)){
       assign r to grp
       grp++  }
}
```

Global variables are assigned to a singleton group.

Example



At Print: grp(i)=0, grp(j)=1

At subr1: Max grp used in print is 1
grp(x)=2, grp(y)=3

At subr2: Max grp used in print is 1
grp(t)=2

At main: Max grp used in children is 3
grp(a)=4, grp(b)=5, grp(c)=6

If A calls B (directly or indirectly), then none of A's register candidates are in the same group as any of B's register candidates.

This *guarantees* that A and B will use different registers.

Thus no saves or restores are needed across a call from A to B.

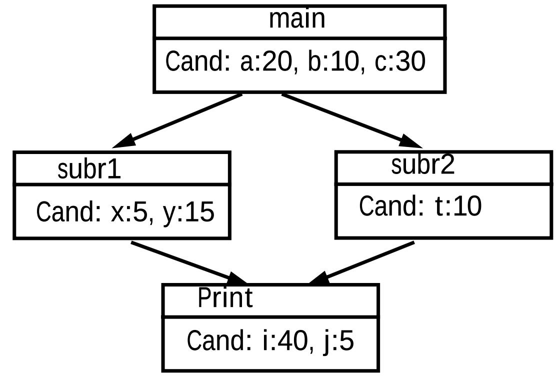
Assigning Registers to Groups

$$Cost(group) = \sum_{candidates \in group} cost(candidates)$$

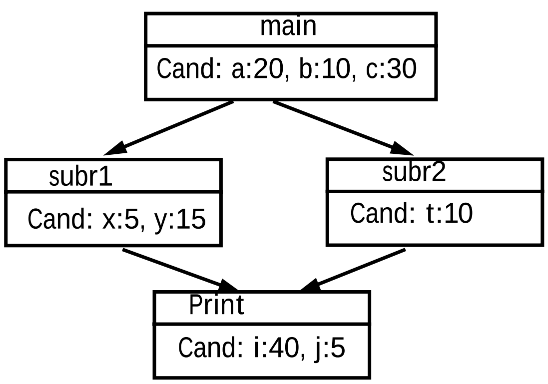
We assign registers to groups based on the cost of each group, using an “auction.”

```
for (r=0; r < RegisterCount; r++) {
    Let G be the group with the
      greatest cost that has not yet
      been assigned a register.
    Assign r to G
}
```


Example (3 Registers)



Group	Members	Cost
0	i	40
1	j	5
2	x, t	15
3	y	15
4	a	20
5	b	10
6	c	30



The 3 registers are given to the groups with the highest weight,

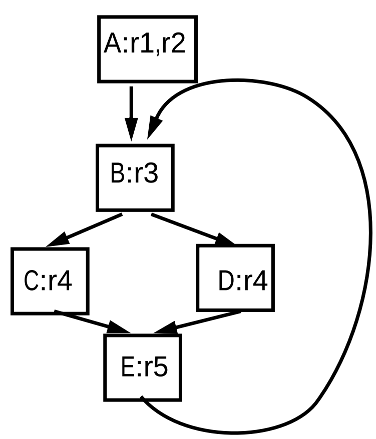
i (40), c(30), a(20).

Is this optimal?

No! If y and t are grouped together, y and t (cost=25) get the last register.

Recursion

To handle recursion, any call to a subprogram “up” in the call graph must save and restore all registers possibly in use between the caller and callee.



A call from E to B saves r3 to r5.

Performance

Wall found interprocedural register allocation to be very effective (given 52 Registers!).

Speedups of 10-28% were reported.

Even with only 8 registers, speedups of 5-20% were observed.

Optimal Interprocedural Register Allocation

Wall's approach to interprocedural register allocation isn't optimal because register candidates aren't grouped to achieve maximum benefit.

Moreover, the placement of save and restore code *if needed* isn't considered.

These limitations are addressed by Kurlander in "Minimum Cost Interprocedural Register Allocation."

Optimal Save-Free Interprocedural Register Allocation

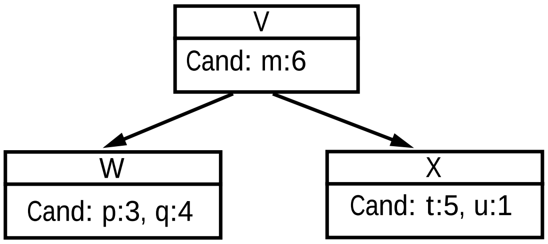
- Allocation is done on a cycle-free call graph.
- Each subprogram has one or more register candidates, c_i.
- Each register candidate, c_i, has a cost (or benefit), w_i, that is the improvement in performance if c_i is given a register. (This w_i value is scaled to include nested loops and expected call frequencies.)

Interference Between Register Candidates

The notion of interference is extended to include interprocedural register candidates:

- Two Candidates in the same subprogram always interfere.
  (Local non-interfering variables and values have already been grouped into interprocedural register candidates.)
- If subprogram P calls subprogram Q (directly or indirectly) then register candidates within P always interfere with register candidates within Q.

Example



The algorithm can group candidate p with either t or u (since they don’t interfere). It can also group candidate q with either t or u.

If two registers are available, it must “discover” that assigning R1 to q&t, and R2 to m is optimal.

Non-interfering register candidates are grouped into registers so as to solve:

$$\text{Maximize} \sum_{c_j \in \bigcup_{i=1}^{k} R_i} w_j$$

That is, we wish to group sets of non-interfering register candidates into k registers such that the overall benefit is maximized.

But how do we solve this?

Certainly examining all possible groupings will be prohibitively expensive!

Kurlander solved this problem by mapping it to a known problem in Integer Programming:

the Dual Network Flow Problem.

Solution techniques for this problem are well known—libraries of standard solution algorithms exist.

Moreover, this problem can be solved in *polynomial time*.

That is, it is “easier” than optimal global (intraprocedural) register allocation, which is NP-complete!

Reading Assignment

- Read Section 15.4 (Code Scheduling) of Chapter 15.
- Read Gibbon’s and Muchnick’s paper, “Efficient Instruction Scheduling for a Pipelined Architecture.”
- Read Kerns and Eggers’ paper, “Balanced Scheduling: Instruction Scheduling When Memory Latency is Uncertain.” (Linked from the class Web page.)

Adding Saves & Restores

Wall designed his save-free interprocedural allocator for a machine with 52 registers.

Most computers have far fewer registers, and hence saving and restoring across calls, *when profitable*, should be allowed.

Kurlander's Technique can be extended to include save/restore costs. If the cost of saving and restoring is *less* than the benefit of allocating an extra register, saving is done. Moreover, saving is done where it is *cheapest* (not closest!).

Example

```
main() { ... p(); ...}

p(){ ...
     for (i=0; i<1000000; i++){
         q():
   }
}
```

We first allocate registers in a save-free mode. After all Registers have been allocated, **q** may need additional registers.

Most allocators would add save/restore code at **q**'s call site (or **q**'s prologue and epilogue).

An optimal allocator will place save/restore code at **p**'s call site, freeing a register that **p** doesn't even want (but that **q** does want!)

Extending the Cost Model

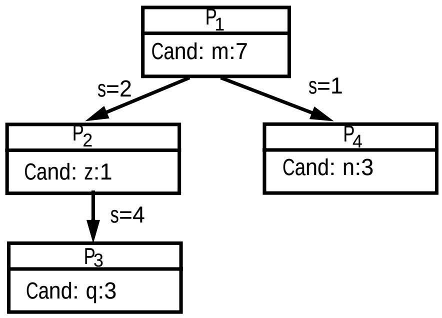
- As before, we group register candidates of different subprograms into registers.
- Now only candidates within the same subprogram automatically interfere.
- Saves are placed on the edges of the call graph.
- We aim to solve

$$\text{Maximize} \sum_{c_j \in \bigcup_{i=1}^{k} R_i} w_j - \sum_{e_m \in \text{ Edges}} s_m * \text{Saved}_m$$

where s_m is the per/register save/restore cost and Saved_m is the number of registers saved on edge e_m.

- As registers are saved, they may be reused in child subprograms.
- This optimization problem can be solved as a Network Dual Flow Problem.
- Again, the solution algorithm is *polynomial*.

Example (One Register)



P_1 gets R1 save-free for m.

A save on $P_1 \rightarrow P_4$ costs 1 and gives a register to n (net profit =2), so we do it.

A save on $P_1 \rightarrow P_2$ for z costs 2, and yields 1, which isn't profitable.

A save on $P_2 \rightarrow P_3$ for q costs 4, and yields 3, which isn't profitable.

A save on $P_1 \rightarrow P_2$ for q costs 2, and yields 3, which *is* a net gain.

Handling Global Variables

- Wall's technique handled globals by assuming they interfere with all subprograms and all other globals.
- Kurlander's approach is incremental (and non-optimal):

  First, an optimal allocation for r registers is computed.

  Next, one register is "stolen" and assigned interprocedurally to the most beneficial global. (Subprograms that don't use the global can save and restore it locally, allowing local reuse). An optimal allocation using R-1 registers is computed. If this solution plus the shared global is more profitable than the R register

solution, the global allocation is "locked in."

Next, another register is "stolen" for a global, leaving R-2 for interprocedural allocation.

This process continues until stealing another register for a global isn't profitable.

Why is Optimal Interprocedural Register Allocation Easier than Optimal IntraProcedural Allocation?

This result seems counter-intuitive. How can allocating a whole program be *easier* (computationally) than allocating only one subprogram.

Two observations provide the answer:

- Interprocedural allocation assumes some form of local allocation has occurred (to identify register candidates).
- Interprocedural interference is *transitive* (if A interferes with B and B interferes with C then A interferes with B). But intraprocedural interference *isn't* transitive!