Reading Assignment

- Read “Minimum Cost Interprocedural Register Allocation,” by S. Kurlander et al. (linked from class Web page).
- Get Handout #4 from DoIt.

Call Graphs

A *Call Graph* represents the calling structure of a program.

- Nodes are subprograms (procedures and functions).
- Arcs represent calls between subprograms. An arc from A to B denotes that a call to B appears within A.
- For an indirect call (a function parameter or a function pointer) an arc is added to all potential callees.

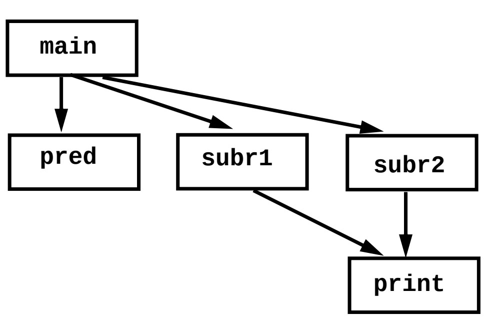
Example

```
main() {
  if (pred(a,b))
       subr1(a)
  else subr2(b);}

bool pred(int a, int b){
   return a==b; }

subr1(int a){ print(a);}

subr2(int x){ print(2*x);}
```



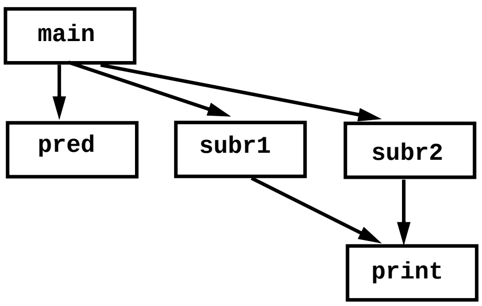
Wall’s Interprocedural Register Allocator

Operates in two phases:

1. Register candidates are identified at the subprogram level.
   Each candidate (a single variable *or* a set of non-interfering live ranges) is compiled as if it *won’t* get a register. At link-time unnecessary loads and stores are edited away *if* the candidate *is* allocated a register.
2. At link-time, when all subprograms are known and available, register candidates are allocated registers.

Identifying Interprocedural Register Sharing

If two subprograms are not connected in the call graph, a register candidate in each can share the same register without any saving or restoring across calls.



A register candidate from **`pred`**, **`subr1`** and **`subr2`** can all share one register.

At the interprocedural level we must answer 2 questions:

1. A local candidate of one subprogram can share a register with candidates of what other subprograms?
2. Which local register candidates will yield the greatest benefit if given a register?

Wall designed his allocator for a machine with 52 registers. This is enough to divide all the registers among the subprograms without any saves or restores at call sites.

With fewer registers, spills, saves and restores will often be needed (if registers are used aggressively within a subprogram).

Restrictions on the Call Graph

Wall limited calls graphs to DAGs since cycles in a call graph imply recursion, which will force saves and restores (why?)

Cost Computations

Each register candidate is given a per-call cost, based on the number of saves and restores that can be removed, scaled by $10^{\text{loop\_depth}}$.

This benefit is then multiplied by the *expected* number of calls, obtained by summing the total number of call sites, scaled by loop nesting depth.

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*.

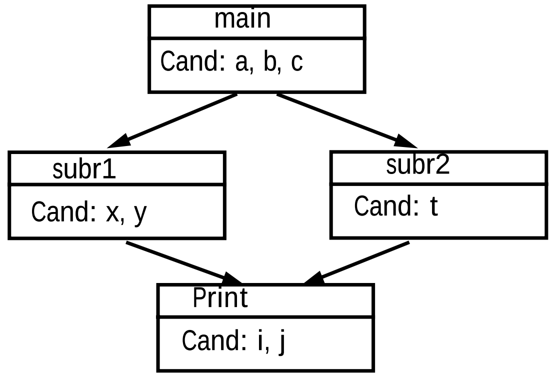
All candidates within a group can share a register. No two candidates in any subprogram are in the same group.

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.

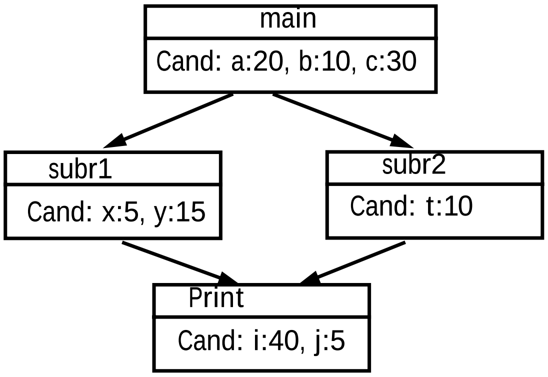
Assigning Registers to Groups

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

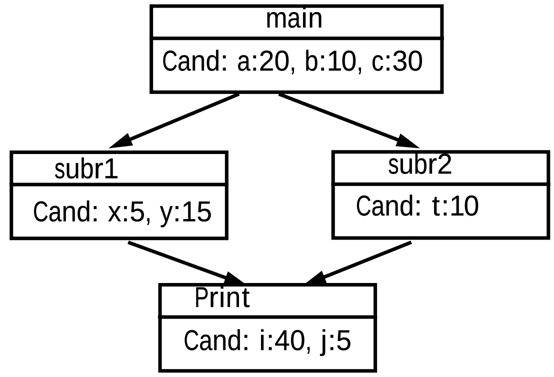
We assign registers to groups based on the cost of each group, using an "auction."

```
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,

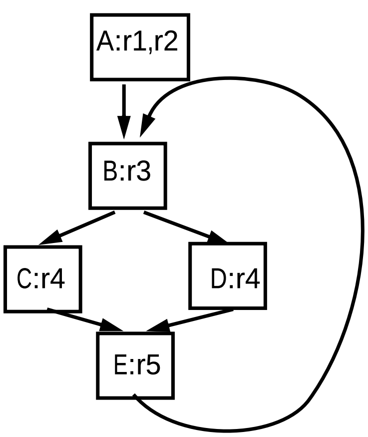
i (40), c(30), a(20).

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

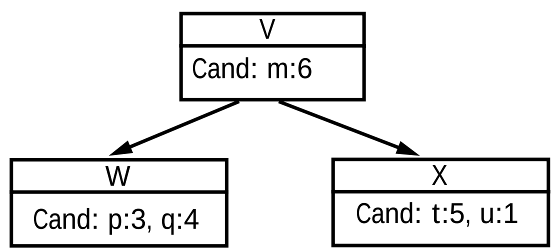
- Allocation is done on a cycle-free call graph.
- Each subprogram has one or more register candidates, c_i.
- Each register candidate, c_i, has a cost (or benefit), w_i, that is the improvement in performance if c_i is given a register. (This w_i value is scaled to include nested loops and expected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