Large vs. Small Live Ranges

- **A large live range has less spill code. Values are directly read from and written to a register.**
  ***But*, a large live range is harder to allocate, since it may conflict with many other register candidates.**
- **A small live range is easier to allocate since it competes with fewer register candidates.**
  ***But*, more spill code is needed to load and save register values across live ranges.**
- **In the limit a live range can shrink to a single definition or use of a register.**
  ***But*, then we really don't have an effective register allocation at all!**

Terminology

In an Interference Graph:

- **A node with fewer neighbors than colors is termed *unconstrained*. It is trivial to color.**
- **A node that is not unconstrained is termed *constrained*. It may need to be split or spilled.**

Reading Assignment

- **Read "Minimum Cost Interprocedural Register Allocation," by S. Kurlander et al. (linked from class Web page).**
- **Read David Wall's paper, "Global Register Allocation at Link Time."**

```
PriorityRegAlloc(proc, regCount) {
   ig ← buildInterferenceGraph(proc)
  unconstrained ←
      { n ∈ nodes(ig) | neighborCount(n) < regCount }
  constrained ←
      { n ∈ nodes(ig) | neighborCount(n) ≥ regCount }

   while( constrained ≠ ϕ )  {
      for ( c ∈ constrained such that not colorable(c)
             and canSplit(c) ) {
         c1, c2 ← split(c)
         constrained ← constrained - {c}
         if ( neighborCount(c1) < regCount )
              unconstrained ← unconstrained U { c1}
         else  constrained ← constrained U {c1}
         if ( neighborCount(c2) < regCount )
              unconstrained ← unconstrained U { c2}
         else  constrained ← constrained U {c2}
         for ( d ∈ neighbors(c) such that
              d ∈ unconstrained and
               neighborCount(d) ≥ regCount ){
                unconstrained ← unconstrained - {d}
              constrained ← constrained U {d}
      }      } // End of both for loops
```

```
    /* At this point all nodes in constrained are
        colorable or can’t be split */

     Select p ∈ constrained such that
                priority(p) is maximized
      if ( colorable(p) )
             color(p)
      else  spill(p)
  } // End of While
 color all nodes ∈ unconstrained
}
```

How to Split a Constrained Node

- **There are many possible partitions of a live range; too many to fully explore.**
- **Heuristics are used instead. One simple heuristic is:**

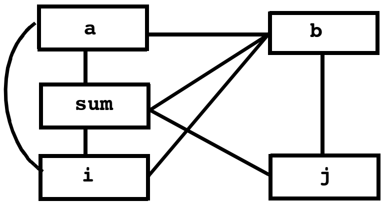
  1. Remove the first basic block (or instruction) of the live range. Put it into a new live range, NR.

  2. Move successor blocks (or instructions) from the original live range into NR, as long as NR remains colorable.

  3. Single Basic Blocks (or instructions) that can’t be colored are spilled.

Example

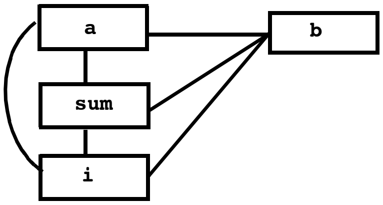
```
int sum(int a[], int b[]) {
  int sum = 0;
  for (int i=0; i<1000; i++)
     sum += a[i];
  for (int j=0; j<1000; j++)
     sum += b[j];
  return sum;
}
```



Assume we want a 3-coloring.

We first simplify the graph by removing unconstrained nodes (those with < 3 neighbors).

Node `j` is removed. We now have:

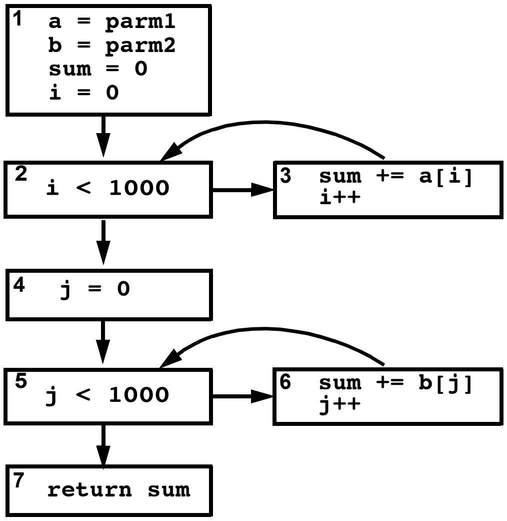


At this point, each node has 3 neighbors, so either spilling or splitting is necessary.

A spill really isn’t attractive as each of the 4 register candidates is used within a loop, magnifying the costs of accessing memory.

Coloring by Priorities

We'll color constrained nodes by priority values, with preference given to large priority values.



	a	b	sum	i
Cost	11	11	42	41
Cost/Size	11/3	11/6	42/7	41/3

Variables `i`, `sum` and `a` are assigned colors `R1`, `R2` and `R3`.

Variable `b` can't be colored, so we will try to split it. `b`'s live range is blocks 1 to 6, with 1 as `b`'s entry point.

Blocks 1 to 3 can't be colored, so `b` is spilled in block 1. However, blocks 4 to 6 form a split live range that can be colored (using `R3`).

We will reload `b` into `R3` in block 4, and it will be register-allocated throughout the second loop. The added cost due to the split is minor—a store in block 1 and a reload in block 4.

Choice of Spill Heuristics

We have seen a number of heuristics used to choose the live ranges to be spilled (or colored).

These heuristics are typically chosen using one's intuition of what register candidates are most (or least) important. Then a heuristic is tested and "fine tuned" using a variety of test programs.

Recently, researchers have suggested using machine learning techniques to automatically determine effective heuristics.

In "Meta Optimization: Improving Compiler Heuristics with Machine Learning," Stephenson, Amarasinghe, et al, suggest using *genetic programming* techniques in

which priority functions (like choice of spill candidates) are mutated and allowed to “evolve.”

Although the approach seems rather random and unfocused, it can be effective. Priority functions *better than* **those used in real compilers have been reported, with research still ongoing.**

Interprocedural Register Allocation

The goal of register allocation is to keep frequently used values in registers.

Ideally, we’d like to go to memory only to access values that may be aliased or pointed to.

For example, array elements and heap objects are routinely loaded from and stored to memory each time they are accessed.

With alias analysis, optimizations like Scalarization are possible.

```
for (i=0; i<1000; i++)
     for (j=0; j<1000; j++)
        a[i] += i * b[j];
```

is optimized to

```
for (i=0; i<1000; i++){
     int Ai = a[i];
     for (j=0; j<1000; j++)
        Ai += i * b[j];
     a[i] = Ai;
}
```

Attacking Call Overhead

- **Even with good global register allocation calls are still a problem.**
- **In general, the caller and callee may use the same registers, requiring saves and restores across calls.**
- **Register windows help, but they are inflexible, forcing all subprograms to use the same number of registers.**
- **We’d prefer a register allocator that is sensitive to the calling structure of a program.**

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.**

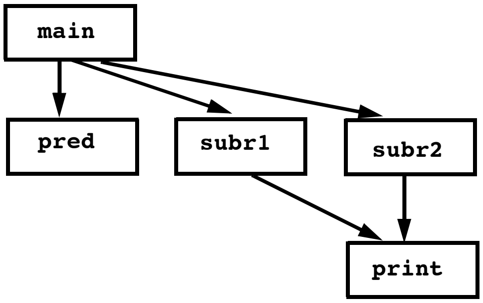
Example

```
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);}
```



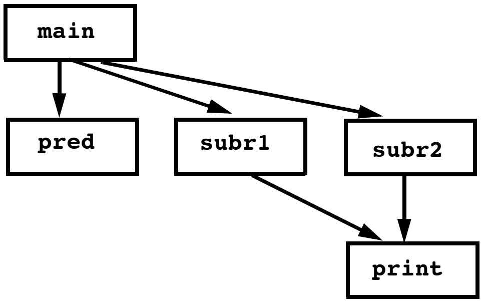
Wall’s Interprocedural Register Allocator

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^{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*.

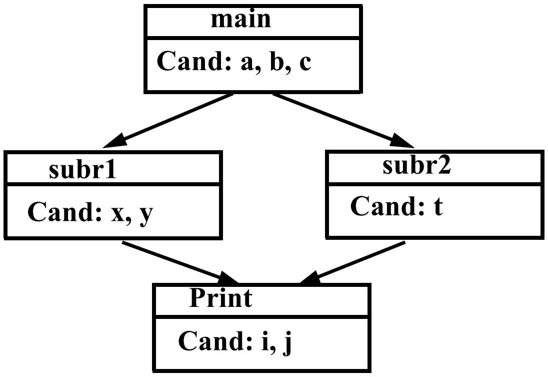
All candidates within a group can share a register. No two candidates in any subprogram are in the same group.

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+ $\max_{c \in children(n)}$ (Max group used in c)
  }
  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.

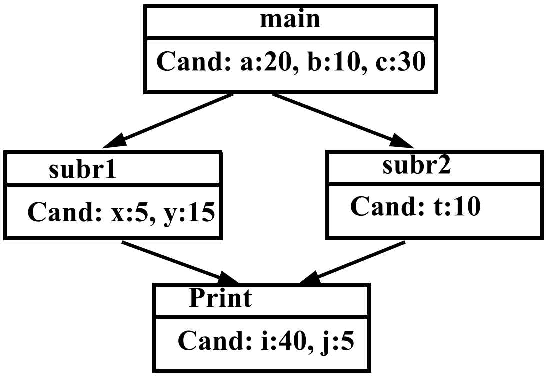
Assigning Registers to Groups

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

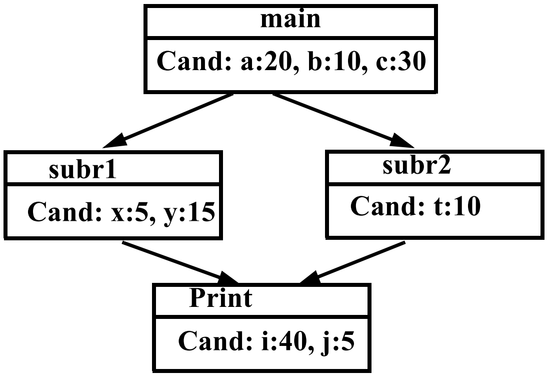
We assign registers to groups based on the cost of each group, using an "auction."

```
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


main

Cand: a:20, b:10, c:30

subr1

Cand: x:5, y:15

subr2

Cand: t:10

Print

Cand: i:40, j:5

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

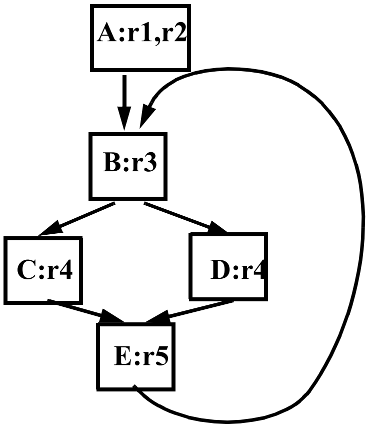
i (40), c(30), a(20).

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:r1,r2

B:r3

C:r4

D:r4

E:r5

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

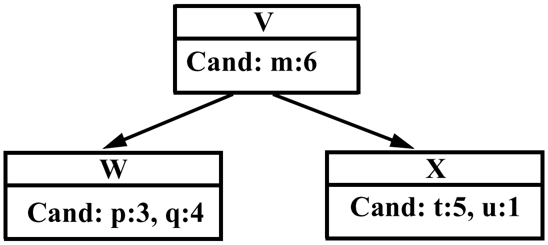
- Allocation is done on a cycle-free call graph.
- Each subprogram has one or more register candidates, c_i.
- Each register candidate, c_i, has a cost (or benefit), w_i, that is the improvement in performance if c_i is given a register. (This w_i value is scaled to include nested loops and expected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



V

Cand: m:6

W

Cand: p:3, q:4

X

Cand: t:5, u:1

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!

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

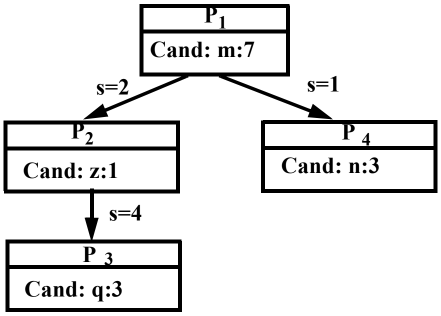
- **As before, we group register candidates of different subprograms into registers.**
- **Now only candidates within the same subprogram automatically interfere.**
- **Saves are placed on the edges of the call graph.**
- **We aim to solve**

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

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
Cand: m:7
s=2
s=1
P_2
Cand: z:1
P_4
Cand: n:3
s=4
P_3
Cand: q:3

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.