Activation Record (AR) Details

If lifetime of AR matches lifetime of invocation, **AND**
If code normally executes a "return" ⇒ Keep ARs on **STACK**

If a procedure can outlive its caller, **OR**
If it can return an object that can reference its execution state ⇒ Keep ARs on **HEAP**

If a procedure makes no Calls ⇒ AR can be allocated **statically**

Efficiency ⇒ **Static** > **Stack** > **Heap**

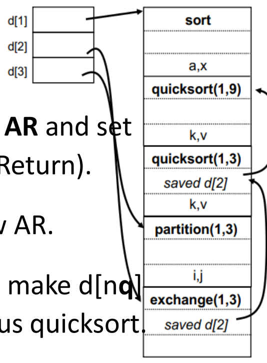
Local Variables are stored at a fixed offset in the stack. Translated into a **static coordinate** <level, offset>.

Set Up Access Links

Procedure **P** calls **Q**, we have to check the depth (n**p** and n**q**). Possible cases:

- Case n**q** > n**p**: create another level ⇒ q links to p, copy ARP to of the caller's to the callee's access link
- Case n**q** <= n**p**: Follow n**p** essentially⇒ copy ARP from the mother caller essentially

Display



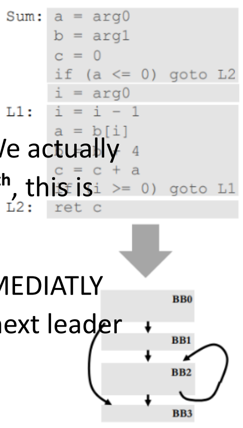
Create an auxiliar **array of pointers** to **AR** on the stack. At each depth ***i*** save the value of **d[i]** in the new **AR** and set **d[i]** to point to the new AR. Before an activation ends, d[i] is reset to the saved value (like ARP link and Return).

• Case n**q** > n**p**: then n**q** = n**p** + 1 ⇒ First np elements of the display do not change; just set d[n**q**] to new AR.

• Case n**q** ≤ n**p**: ⇒ Enclosing procedures at levels 1, ..., n**q**-1 are the same; save old d[nq] in new AR and make d[n**q**] point to new AR. Essentially, d[2] points to the new quicksort, but the quicksort AR points to the previous quicksort.

Notes: Access links can be tricky, but they should only point to the main procedure, ie: P2 and P3 can all access main "global" vars, so link is directed to main. If there is a redefinition of global vars, and the procedure uses more vars, display is needed.

When extending 2 classes (one being a parent of the other), the child of the child extended function requires an offset (+12) to access it.



```
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
```

Register Allocation

Top-down local register allocator: We calculate the Benefit(V, B) = *uses* and *defs* of the var V in block B. We actually need the TopBenefit(V) = Sum(Benefit(V,B) * freq(B)), for all blocks B. If freq(B) is unknown, we use $10^{\mathbf{depth}}$, this is amazing for nested/recursion code. We then assign the top variables to registers.

Basic block algorithm: **Step 1**: The statement target of a goto, is a leader instructions. The statement IMMEDIATLY after the goto, is a leader instruction. **Step 2**: Each block is now the leader instruction all the way to the next leader instruction/exit.

Bottom-Up Local allocator: vr_z <- vr_x OP vr_y.
We use virtual registers for this case. We store the number of physical registers available, as well as the name of the vr associated. Also store the distance to the next use in the block. Flag indicating if its free or in use. Stack with free physical registers. We also have these auxiliar 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.

When we find a dirty register, we need to store the value in memory, to be able to clean it. If we ran out of registers, we find the maximum value of next and we will store it in memory, to reuse it (just like the previous line).

We use webs to assure that the value defined by it won't be used by another web. Thus, there is no need to carry the value between webs. This solves the issue of cross Basic Blocks register assignment issues.

Two webs interfere if they overlap in execution. They do not interfere if the value of w1 is an operand at its end, and w2 starts at that instruction as the destination – if this happens, we can assign the same register without a problem.

Graph coloring

If webs interfere, they cannot use the same register (same color in graph). When coloring a graph:

• 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
• 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)

Splitting algorithm: Pick a web that is not used for the largest enclosing block around a point of the program. Then, we split that web into 2, and we redo the interference graph.

Optimizations: If there are copy instructions, if possible combine their webs; Pre-color webs that are associated with arguments and return value.

Instruction Scheduling

Fetch > Decode > Execute > Memory > Writeback

Optimizations

• Constant propagation ⇒ Change variable to constant
• Algebraic Simplification ⇒ Simplify x*1 to x, or x*0 to 0
• Copy propagation ⇒ Remove x = x
• Common Sub-Expression Elimination ⇒ (i+1) * (i+1), create a variable t = i + 1 and do t*t

- Dead code elimination ⇒ Remove unused variables
- Loop Invariant Removal ⇒ Create a variable outside and use it on the loop, instead of repeating
- Strength Reduction ⇒ Pass u*i to v, where v is v = v + u

These optimizations can greatly improve O(x) of the code