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

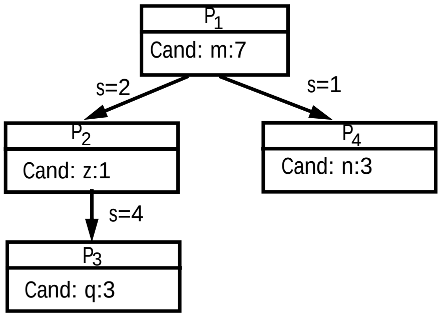
- As before, we group register candidates of different subprograms into registers.
- Now only candidates within the same subprogram automatically interfere.
- Saves are placed on the edges of the call graph.
- We aim to solve

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

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!

Code Scheduling

Modern processors are pipelined.

They give the impression that all instructions take unit time by executing instructions in *stages* (steps), as if on an assembly line.

Certain instructions though (loads, floating point divides and square roots, delayed branches) take more than one cycle to execute.

These instructions may *stall* (halt the processor) or require a nop (null operation) to execute properly.

A *Code Scheduling* phase may be needed in a compiler to avoid stalls or eliminate nops.

Scheduling Expression DAGs

After generating code for a DAG or basic block, we may wish to schedule (reorder) instructions to reduce or eliminate stalls.

A *Postpass Scheduler* is run after code selection and register allocation.

Postpass schedulers are very general and flexible, since they can be used with code generated by any compiler with any degree of optimization

But, since they can't modify register allocations, they can't always avoid stalls.

Dependency DAGs

Obviously, not all reorderings of generated instructions are acceptable.

Computation of a register value must precede all uses of that value.

A store of a value must precede all loads that might read that value.

A *Dependency Dag* reflects essential ordering constraints among instructions:

- Nodes are Instructions to be scheduled.
- An arc from Instruction i to Instruction j indicates that i must be executed before j may be executed.

Kinds of Dependencies

We can identify several kinds of dependencies:

- True Dependence:

  An operation that uses a value has a true dependence (also called a flow dependence) upon an earlier operation that computes the value. For example:

  ```
  mov  1, %l2
  add  %l2, 1, %l2
  ```

- Anti Dependence:

  An operation that writes a value has a anti dependence upon an earlier operation that reads the value. For example:

  ```
  add  %l2, 1, %l0
  mov  1, %l2
  ```

- Output Dependence:

  An operation that writes a value has a output dependence upon an earlier operation that writes the value. For example:

  ```
  mov  1, %l2
  mov  2, %l2
  ```

  Collectively, true, anti and output dependencies are called data dependencies. Data dependencies constrain the order in which instructions may be executed.

Example

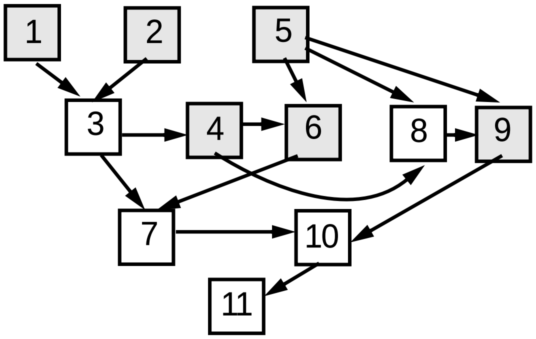
Consider the code that might be generated for

```
a = ((a+b) + (c*d)) + ((c+d) * d);
```

We’ll assume 4 registers, the minimum possible, and we’ll reuse already loaded values.

Assume a 1 cycle stall between a load and use of the loaded value and a 2 cycle stall between a multiplication and first use of the product.

```
1.  ld   [a], %r1
2.  ld   [b], %r2
3.  add  %r1,%r2,%r1   <--- Stall
4.  ld   [c], %r2
5.  ld   [d], %r3
6.  smul %r2,%r3,%r4   <--- Stall
7.  add  %r1,%r4,%r1   <--- Stall*2
8.  add  %r2,%r3,%r2
9.  smul %r2,%r3,%r2
10. add  %r1,%r2,%r1   <--- Stall*2
11. st   %r1,[a]        (6 Stalls Total)
```



Scheduling Requires Topological Traversal

Any valid code schedule is a *Topological Sort* of the dependency dag.

To create a code schedule you

(1) Pick any root of the Dag.

(2) Remove it from the Dag and schedule it.

(3) Iterate!

Choosing a *Minimum Delay* schedule is NP-Complete:

“Computers and Intractability,”
M. Garey and D. Johnson,
W.H. Freeman, 1979.

Dynamically Scheduled (Out of Order) Processors

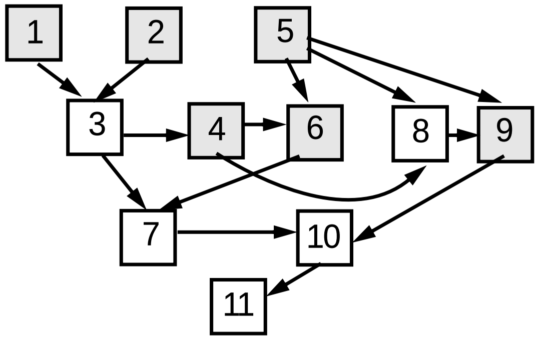
To avoid stalls, some processors can execute instructions *Out of Program Order*.

If an instruction can’t execute because a previous instruction it depends upon hasn’t completed yet, the instruction can be “held” and a successor instruction executed instead.

When needed predecessors have completed, the held instruction is released for execution.

Example

```
 1.  ld   [a], %r1
 2.  ld   [b], %r2
 5.  ld   [d], %r3
 3.  add  %r1,%r2,%r1
 4.  ld   [c], %r2      ←— Stall
 6.  smul %r2,%r3,%r4
 8.  add  %r2,%r3,%r2
 9.  smul %r2,%r3,%r2
 7.  add  %r1,%r4,%r1   ←— Stall
10.  add  %r1,%r2,%r1
11.  st   %r1,[a]    (2 Stalls Total)
```



Limitations of Dynamic Scheduling

1. Extra processor complexity.
2. Register renaming (to avoid *False Dependencies*) may be needed.
3. Identifying instructions to be delayed takes time.
4. Instructions “late” in the program can’t be started earlier.

Gibbons & Muchnick Postpass Code Scheduler

1. If there is only one root, schedule it.
2. If there is more than one root, choose that root that won't be stalled by instructions already scheduled.
3. If more than one root can be scheduled without stalling, consider the following rules (in order);
   (a) Does this root stall any of its successors?
   (If so, schedule it immediately.)

   (b) How many new roots are exposed if this node is scheduled?
   (More is better.)

(c) Which root has the longest weighted path to a leaf (using instruction delays as the weight). (The "critical path" in the DAG gets priority.)