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

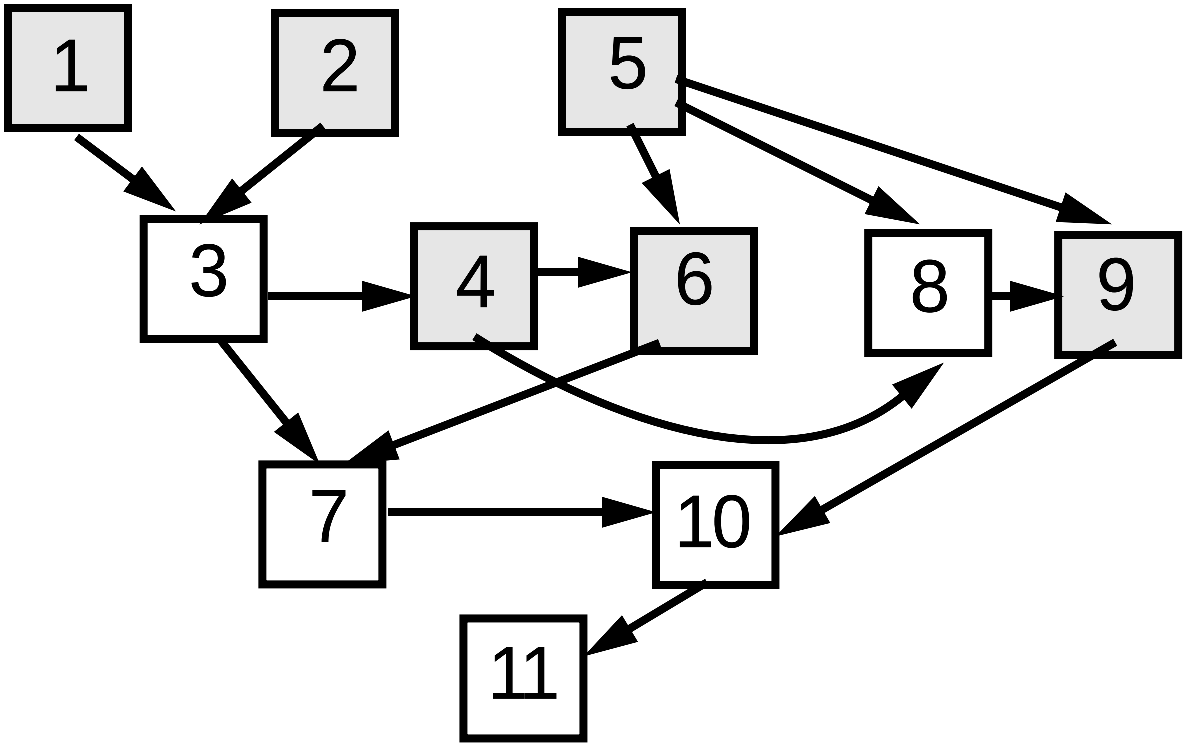
Consider the code that might be generated for

```
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

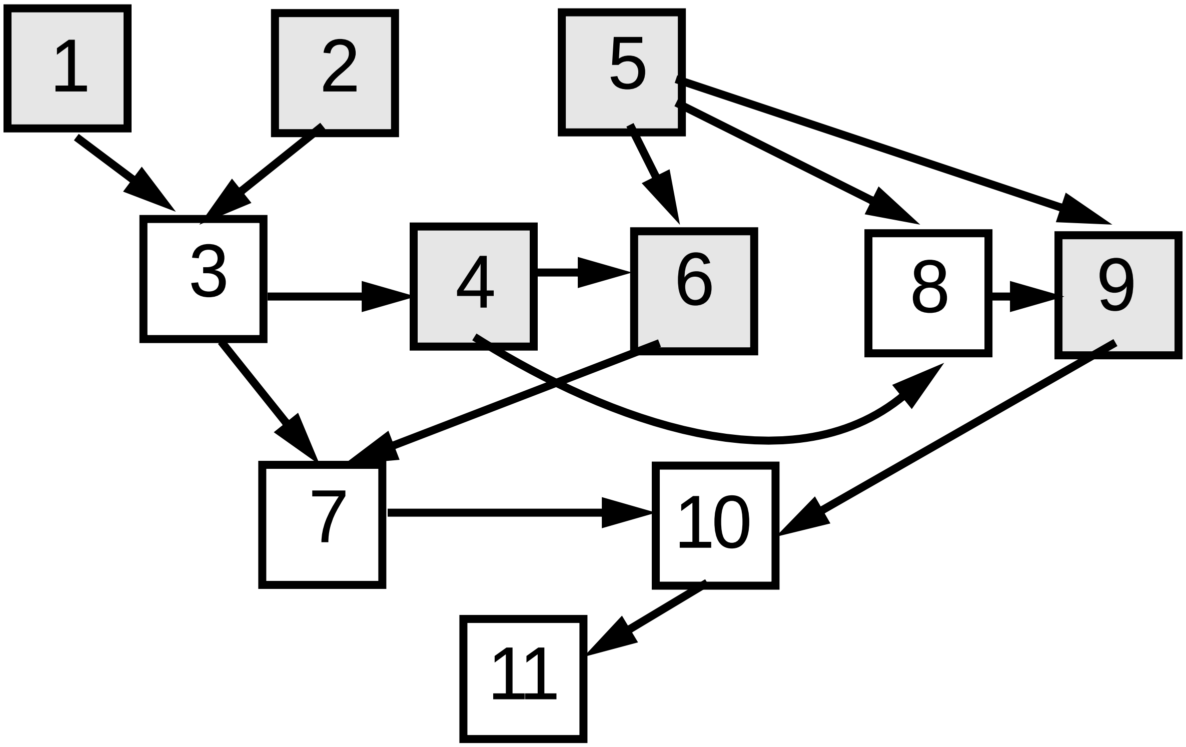
To avoid stalls, some processors can execute instructions *Out of Program Order*.

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
10. add  %r1,%r2,%r1      <-- Stall
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.

Reading Assignment

- Read Goodman and Hsu’s paper, “Code Scheduling and Register Allocation in Large Basic Blocks.”
- Read Bernstein and Rodeh’s paper, “Global Instruction Scheduling for Superscalar Machines.”
(Linked from the class Web page.)

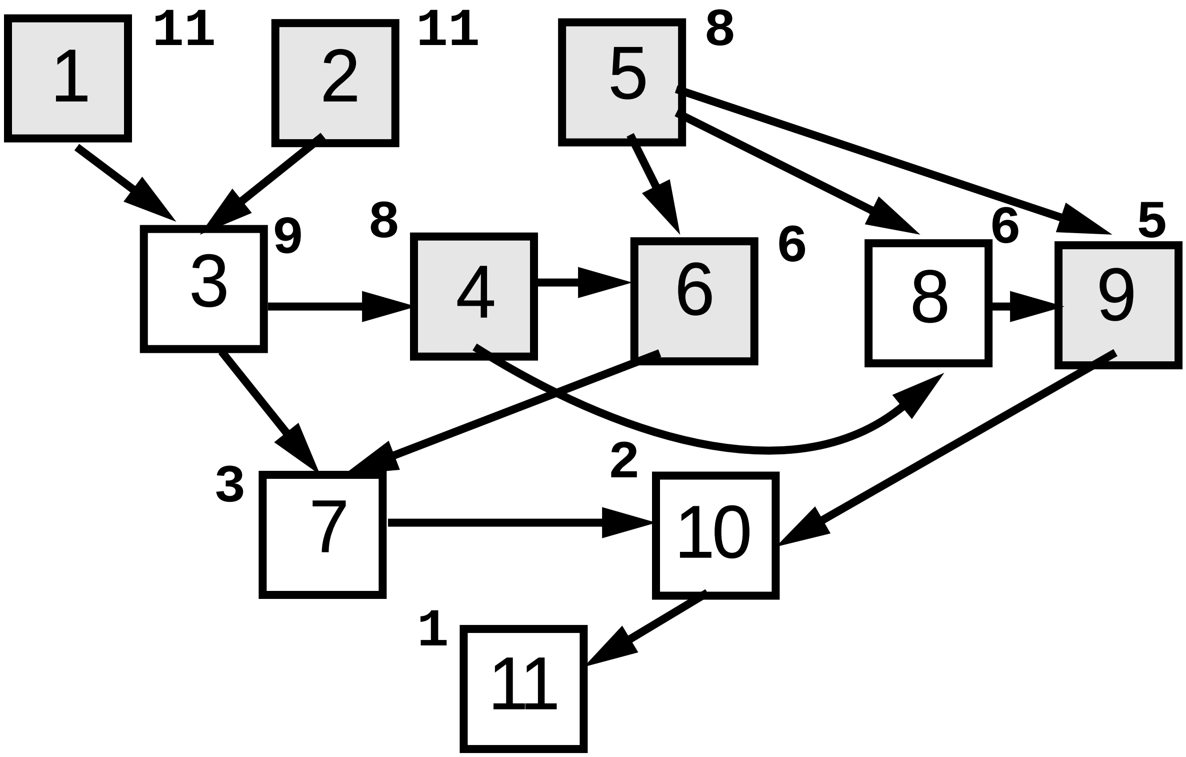
Gibbons & Muchnick Postpass Code Scheduler

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.)

Example

```
1.  ld    [a], %r1       //Longest path
2.  ld    [b], %r2       //Exposes a root
5.  ld    [d], %r3       //Not delayed
3.  add   %r1,%r2,%r1    //Only choice
4.  ld    [c], %r2       //Only choice
6.  smul  %r2,%r3,%r4    //Stalls succ.
8.  add   %r2,%r3,%r2    //Not delayed
9.  smul  %r2,%r3,%r2    //Not delayed
7.  add   %r1,%r4,%r1    //Only choice
10. add   %r1,%r2,%r1    //Only choice
11. st    %r1,[a]        (2 Stalls Total)
```



False Dependencies

We still have delays in the schedule that was produced because of “false dependencies.”

Both **`b`** and **`c`** are loaded into **`%r2`**. This limits the ability to move the load of **`c`** prior to any use of **`%r2`** that uses **`b`**.

To improve our schedule we can use a processor that renames registers *or* allocate additional registers to remove false dependencies.

Register Renaming

Many out of order processors automatically rename distinct uses of the same architectural register to distinct internal registers.

Thus

```
ld [a],%r1
ld [b],%r2
add %r1,%r2,%r1
ld [c],%r2
```

is executed as if it were

```
ld [a],%r1
ld [b],%r2
add %r1,%r2,%r3
ld [c],%r4
```

Now the final load can be executed prior to the add, eliminating a stall.

Compiler Renaming

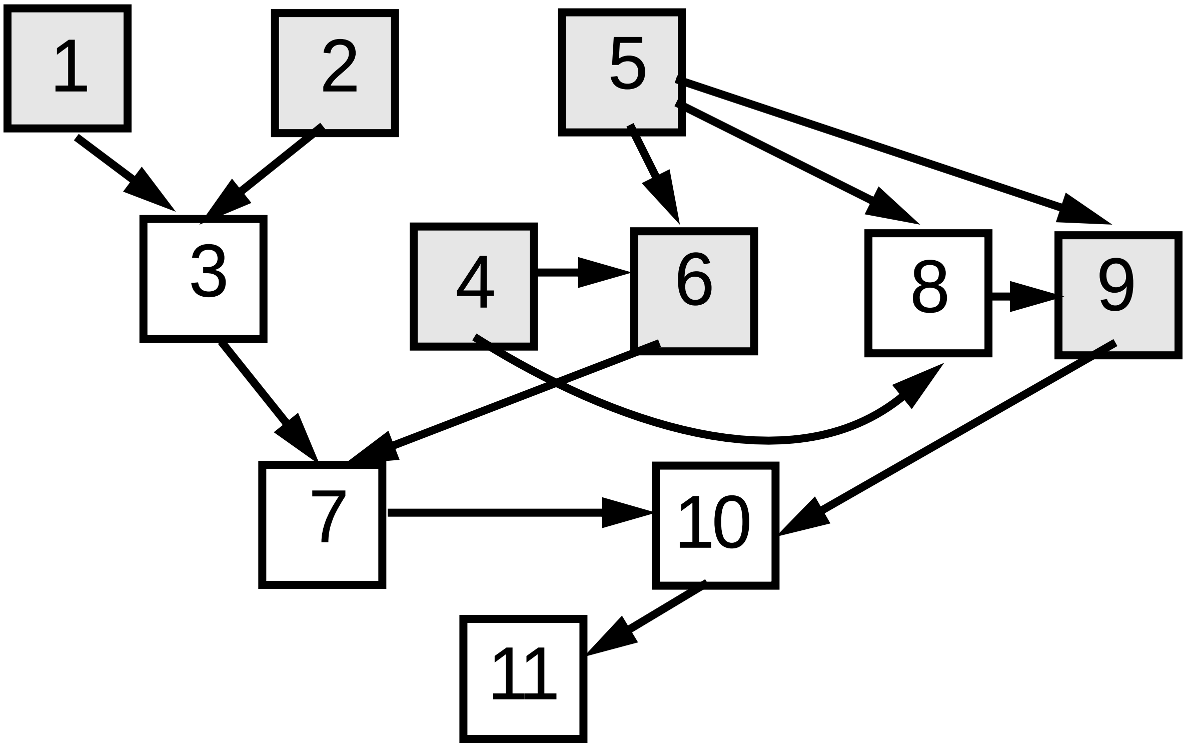
A compiler can also use the idea of renaming to avoid unnecessary stalls.

An extra register may be needed (as was the case for scheduling expression trees).

Also, a *round-robin* allocation policy is needed. Registers are reused in a *cyclic* fashion, so that the most recently freed register is reused last, not first.

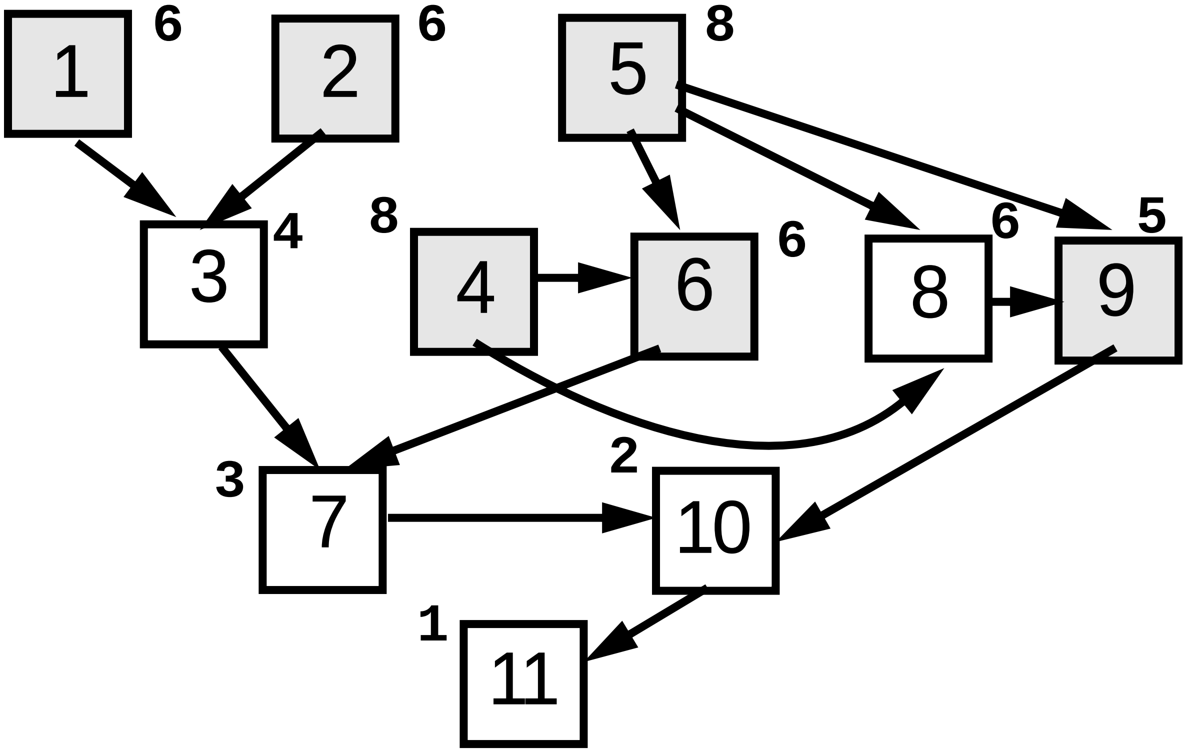
Example

```
1.  ld    [a], %r1
2.  ld    [b], %r2
3.  add   %r1,%r2,%r1   ← Stall
4.  ld    [c], %r3
5.  ld    [d], %r4
6.  smul  %r3,%r4,%r5   ← Stall
7.  add   %r1,%r5,%r2   ← Stall*2
8.  add   %r3,%r4,%r3
9.  smul  %r3,%r4,%r3
10. add   %r2,%r3,%r2   ← Stall*2
11. st    %r2,[a]        (6 Stalls Total)
```



After Scheduling:

```
4.  ld    [c],%r3       //Longest path
5.  ld    [d],%r4       //Exposes a root
1.  ld    [a],%r1       //Stalls succ.
2.  ld    [b],%r2       //Exposes a root
6.  smul  %r3,%r4,%r5   //Stalls succ.
8.  add   %r3,%r4,%r3   //Longest path
9.  smul  %r3,%r4,%r3   //Stalls succ.
3.  add   %r1,%r2,%r1   //Only choice
7.  add   %r1,%r5,%r2   //Only choice
10. add   %r2,%r3,%r2   //Only choice
11. st    %r2,[a]         (0 Stalls Total)
```



Balanced Scheduling

When scheduling a load, we normally anticipate the *best* case, a hit in the primary cache.

On older architectures this makes sense, since we stall execution on a cache miss.

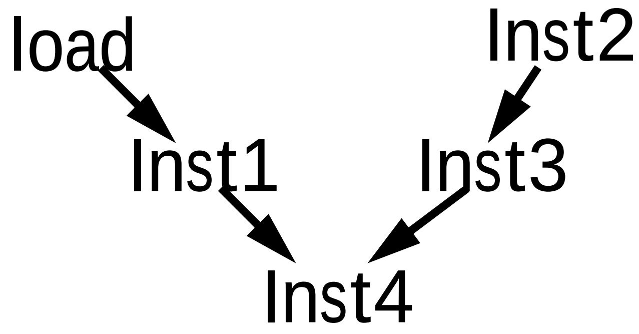
Many newer architectures are *non-blocking*. This means we can continue execution after a miss until the loaded value is used.

Assume a Cache miss takes N cycles (N is typically 10 or more).

Do we schedule a load anticipating a 1 cycle delay (a hit) or an N cycle delay (a miss)?

Neither *Optimistic Scheduling* (expect a hit) nor *Pessimistic Scheduling* (expect a miss) is *always* better.

Consider



An Optimistic Schedule is

load
Inst2
Inst1
Inst3
Inst4

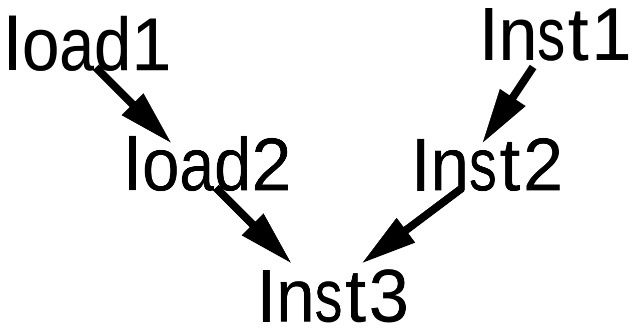
Fine for a hit;
inferior for a miss.

A Pessimistic Schedule is

load
Inst2
Inst3
Inst1
Inst4

Fine for a hit;
better for a miss.

But things become more complex with multiple loads



load1

Inst1

load2

Inst2

Inst3

An Optimistic Schedule is

load1
Inst1
load2
Inst2
Inst3

Better for hits;
same for misses.

A Pessimistic Schedule is

load1
Inst1
Inst2
load2
Inst3

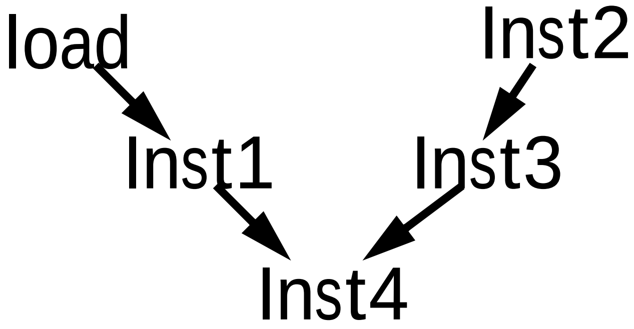
Worse for hits;
same for misses.

Balance Placement of Loads

Eggers suggests a *balanced scheduler* that spaces out loads, using available independent instructions as “filler.”

The insight is that scheduling should not be driven by worst-case latencies but rather by available *Independent* Instructions.

For



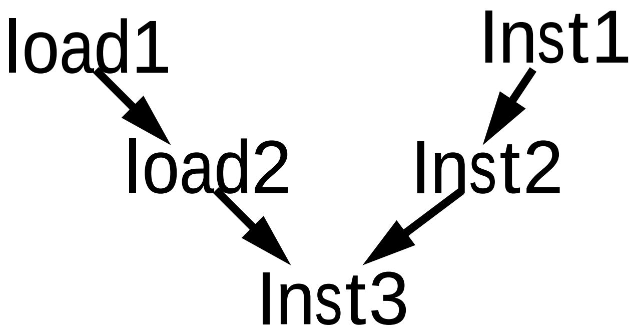
load → Inst1, Inst2 → Inst3, Inst1 → Inst4, Inst3 → Inst4

it produces

load
Inst2
Inst3
Inst1
Inst4

Good; maximum distance between load and Inst1 in case of a miss.

For



load1

Inst1

load2

Inst2

Inst3

balanced scheduling produces

load1
Inst1
load2
Inst2
Inst3

Good for hits;
as good as
possible for misses.

Idea of the Algorithm

Look at each Instruction, i, in the Dependency DAG.

Determine which loads can run in parallel with i and use all (or part) of i's execution time to cover the latency of these loads.

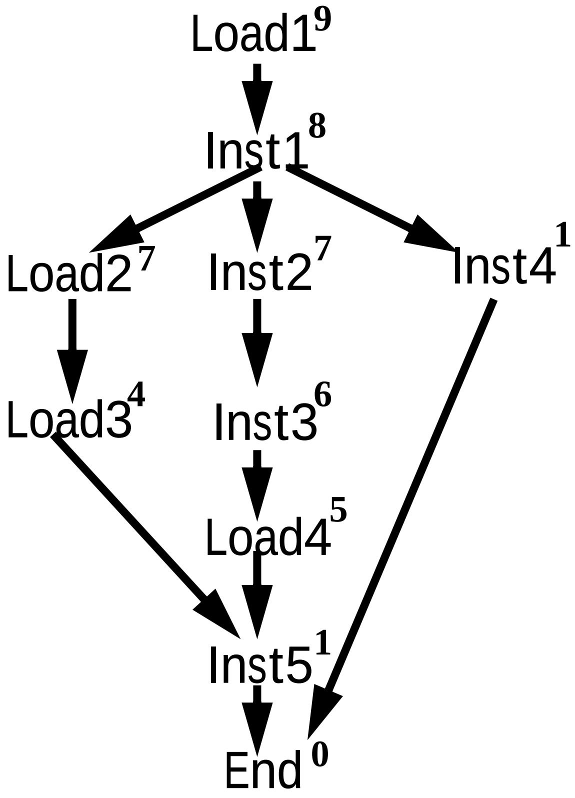
```
Compute available latency of each load:
  Give each load instruction an initial
     latency of 1.
  For (each instruction i in the
   Dependency DAG) do:
      Consider Instructions Independent
         of i:
          $G_{ind}$ = DepDAG -
            (AllPred(i) U AllSucc(i) U {i})
         For (each connected subgraph c
              in $G_{ind}$) do:
             Find m = maximum number of
                 load instructions on any
                 path in c.
             For (each load d in c) do:
                   add 1/m to d's latency.
```

Computing the Schedule Using Adjusted Latencies

Once latencies are assigned to each load (other instructions have a latency of 1), we annotate each instruction in the Dependency DAG with its critical path weight: the maximum latency (along any path) from the instruction to a Leaf of the DAG.

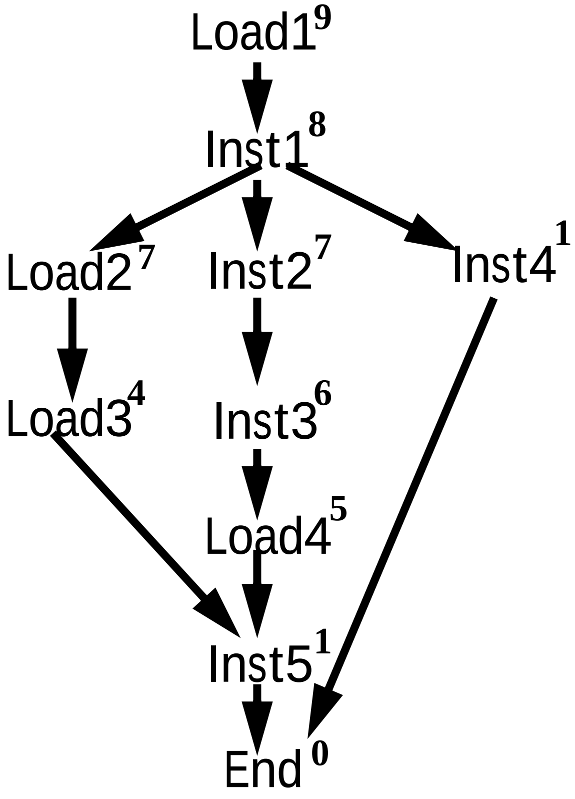
Instructions are scheduled using critical path values; the root with the highest critical path value is always scheduled next. In cases of ties (same critical path value), operations with the longest latency are scheduled first.

Example



	Ld 1	Ld 2	Ld 3	Ld 4	I1	I2	I3	I4	I5	Latency
Load1										**1+0 = 1**
Load2				**1/2**		**1/2**	**1/2**	**1/2**		**1+2 = 3**
Load3				**1/2**		**1/2**	**1/2**	**1/2**		**1+2 = 3**
Load4		**1**	**1**					**1**		**1+3 = 4**

Using the annotated Dependency Dag, instructions can be scheduled:



Load1	(0 latency; unavoidable)
Inst1	
Load2	(3 instruction latency)
Inst2	
Inst3	
Load4	(2 instruction latency)
Load3	(1 instruction latency)
Inst4	
Inst5	