Compiler Renaming

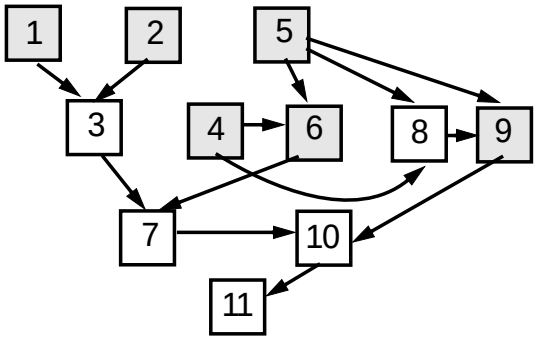
A compiler can also use the idea of renaming to avoid unnecessary stalls.

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.

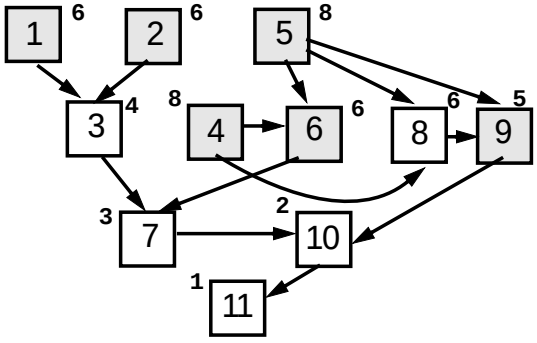
Example

```
1.  ld   [a], %r1
2.  ld   [b], %r2          <- Stall
3.  add  %r1,%r2,%r1
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.

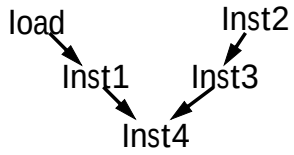
Many newer architectures are *non-blocking*. This means we can continue execution after a miss until the loaded value is used.

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



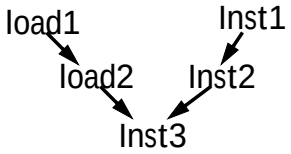
An Optimistic Schedule is

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



An Optimistic Schedule is

load1 Inst1 load2 Inst2 Inst3	Better for hits; same for misses.

A Pessimistic Schedule is

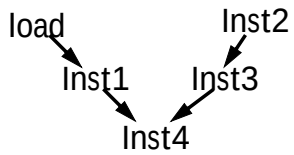
load1 Inst1 Inst2 load2 Inst3	Worse for hits; same for misses.

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.

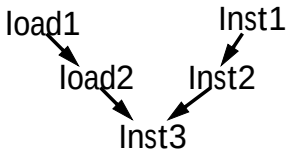
For



it produces

load Inst2 Inst3 Inst1 Inst4	Good; maximum distance between load and Inst1 in case of a miss.

For



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.

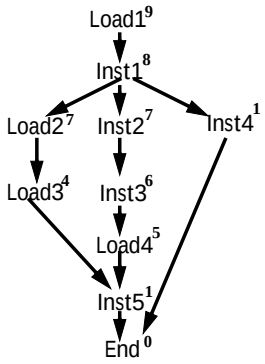
For (each load d in c) do: add 1/m to d's latency.

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.

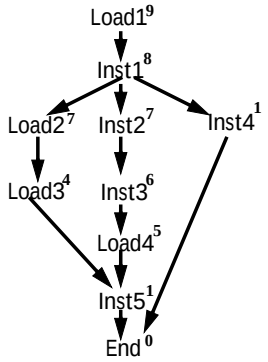
Instructions are scheduled using critical path values; the root with the highest critical path value is always scheduled next. In cases of ties (same critical path value), operations with the longest latency are scheduled first.

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	

Goodman/Hsu Integrated Code Scheduler

Prepass Schedulers:

Schedule code prior to register allocation.

Can overuse registers—Always using a "fresh" register maximizes freedom to rearrange Instructions.

Postpass Schedulers:

Schedule code after register allocation.

Can be limited by "false dependencies" induced by register reuse.

Example is Gibbons/Muchnick heuristic.

Integrated Schedulers

Capture best of both approaches.

When registers are plentiful, use additional registers to avoid stalls.

Goodman & Hsu call this *CSP*:
Code Scheduling for Pipelines.

When registers are scarce, switch to a policy that frees registers.

Goodman & Hsu call this *CSR*:
Code Scheduling to free Registers.

Assume code is generated in single assignment form, with a unique pseudo-register for each computed value.

We schedule from a DAG where nodes are operations (to be mapped to instructions), and arcs represent data dependencies.

Each node will have an associated Cost, that measures the execution and stall time of the instruction that the node represents.

Nodes are labeled with a critical path cost, used to select the "most critical" instructions to schedule.

Definitions

Leader Set:

Set of DAG nodes ready to be scheduled, possibly with an interlock.

Ready Set:

Subset of Leader Set; Nodes ready to be scheduled without an interlock.

AvailReg:

A count of currently unused registers.

MinThreshold:

Threshold at which heuristic will switch from avoiding interlocks to reducing registers in use.

Goodman/Hsu Heuristic:

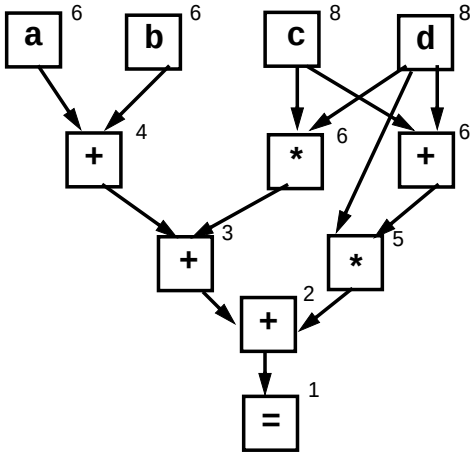
```
while (DAG ≠ ϕ) {
  if ( AvailReg > MinThreshold)
     if (ReadySet ≠ ϕ)
          Select Ready node with Max cost
     else Select Leader node with Max cost
  else  //  Reduce Registers in Use
     if (∃ node ∈ ReadySet that frees registers){
        Select node that frees most registers
        If (selected node isn't unique)
            Select node with Max cost   }
     elsif (∃ node ∈ LeaderSet that frees regs){
           Select node that frees most registers
           If (selected node isn't unique)
              Select node with fewest interlocks}
     else find a partially evaluated path and
           select a leader from this path
     else Select any node in ReadySet
     else Select any node in LeaderSet
Schedule Selected node
Update AvailReg count  }//end while
```


Example

We'll again consider

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

Again, assume a 1 cycle stall between a load and use of its value and a 2 cycle stall between a multiplication and first use of the product.



We'll try 4 registers (the minimum), then 5 registers.

Should MinThreshold be 0 or 1?

At MinThreshold = 1, we always have a register to hold a result, but we may force a register to be spilled too soon!

At MinThreshold = 0, we may be forced to spill a register to free a result register.

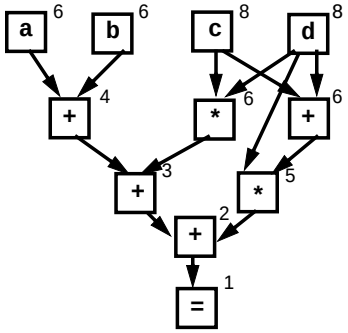
But, we'll also be able to schedule more aggressively.

Is a spill or stall worse?

Note that we may be able to "hide" a spill in a delay slot!

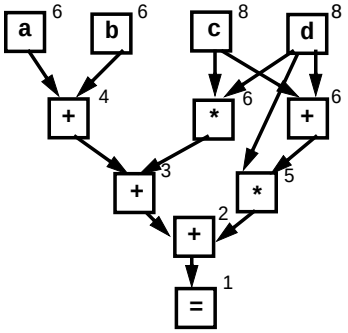
We'll be aggressive and use MinThreshold = 0.

4 Registers Used (1 Stall)



Instruction	Comment	Regs Used
`ld [c], %r1`	Choose ready, cost=8	1
`ld [d], %r2`	Choose ready, cost=8	2
`ld [a], %r3`	Choose ready, cost=6	3
`smul %r1,%r2,%r4`	Choose ready, cost=6	4
`add %r1,%r2,%r1`	Free a register	4
`smul %r1,%r2,%r1`	Free a register	3
`ld [b], %r2`	Choose ready, cost=6	4
`add %r3,%r2,%r3` ←	Choose a leader	3
`add %r3,%r4,%r3`	No choice	2
`add %r3,%r1,%r3`	No choice	1
`st %r3,[a]`	No choice	0

5 Registers Used (No Stalls)



Instruction	Comment	Regs Used
`ld [c], %r1`	Choose ready, cost=8	1
`ld [d], %r2`	Choose ready, cost=8	2
`ld [a], %r3`	Choose ready, cost=6	3
`smul %r1,%r2,%r4`	Choose ready, cost=6	4
`add %r1,%r2,%r1`	Choose ready, cost=6	4
`ld [b], %r5`	Choose ready, cost=6	5
`smul %r1,%r2,%r1`	Free a register	4
`add %r3,%r5,%r3`	Choose ready, cost=4	3
`add %r3,%r4,%r3`	No choice	2
`add %r3,%r1,%r3`	No choice	1
`st %r3,[a]`	No choice	0

Scheduling for Superscalar & Multiple Issue Machines

A number of computers have the ability to issue more than one instruction per cycle *if* the instructions are independent and satisfy constraints on available functional units.

Thus the instructions

```
add %r1,1,%r2
sub %r1,2,%r3
```

can issue and execute in parallel, but

```
add %r1,1,%r2
sub %r2,2,%r3
```

must execute sequentially.

Instructions that are linked by true or output dependencies must execute sequentially, but instructions that are linked by an anti dependence may execute concurrently.

For example,

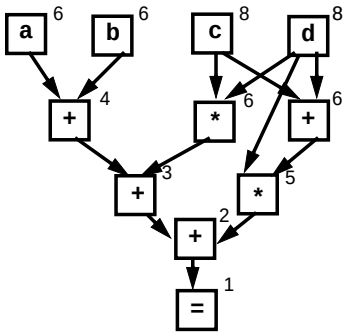
```
add %r1,1,%r2
sub %r3,2,%r1
```

can issue and execute in parallel.

The code scheduling techniques we've studied can be used to schedule machines that can issue 2 or more independent instructions simultaneously.

We select pairs (or triples or n-tuples), verifying (with the Dependence Dag) that they are independent or anti dependent.

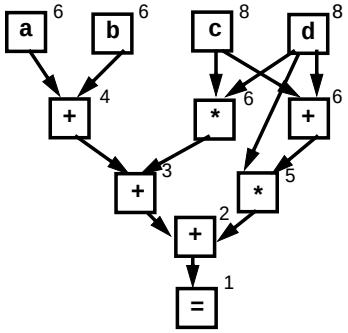
Example: 5 Registers (2 Wide Issue)



1	ld [c], %r1	ld [d], %r2
2	ld [a], %r3	ld [b], %r4
3	smul %r1,%r2,%r5	add %r1,%r2,%r1
4	add %r3,%r4,%r3	smul %r1,%r2,%r1
5	nop	nop
6	add %r3,%r5,%r3	nop
7	add %r3,%r1,%r3	nop
8	st %r3,[a]	nop

We need only 8 cycles rather than 11.

5 Registers (3 Wide Issue)



1	ld [c], %r1	ld [d], %r2	ld [a],%r3
2	ld [b], %r4	nop	nop
3	smul %r1,%r2,%r5	add %r1,%r2,%r1	nop
4	add %r3,%r4,%r3	smul %r1,%r2,%r1	nop
5	nop	nop	nop
6	add %r3,%r5,%r3	nop	nop
7	add %r3,%r1,%r3	nop	nop
8	st %r3,[a]	nop	nop

We still need 8 cycles!

Finding Additional Independent Instructions for Parallel Issue

We can extend the capabilities of processors:

- Out of order execution allows a processor to “search ahead” for independent instructions to launch.
- *But*, since basic blocks are often quite small, the processor may need to accurately predict branches, issuing instructions before the execution path is fully resolved.
- *But*, since branch predictions may be wrong, it will be necessary to “undo” instructions executed speculatively.