Selecting Instructions to Issue

- A list of “ready to issue” instructions in block A and in bocks equivalent to A (or 1-branch distant from A) is maintained.
- All data dependencies must be satisfied and stalls avoided (if possible).
- N independent instructions are selected, where N is the processor’s issue-width.
- But what if more than N instructions are ready to issue?
- Selection is by *Priority*, using two *Scheduling Heuristics*.

Delay Heuristic

This value is computed on a per-basic block basis.

It estimates the worst-case delay (stalls) from an Instruction to the end of the basic block.

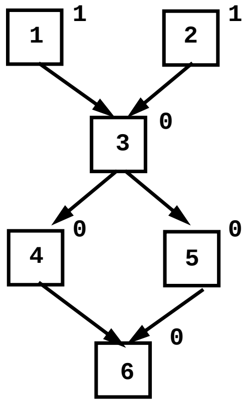
$D(I) = 0$ if I is a leaf.

Let $d(I,J)$ be the delay if instruction J follows instruction I in the code schedule.

$$D(I) = \underset{J_i \in \text{Succ}(I)}{\text{Max}} (D(J_i)+d(I,J_i))$$

Example of Delay Values

```
block1:
1.  ld   [a],Pr1
2.  ld   [b],Pr2
3.  add  Pr1,Pr2,Pr3
4.  st   Pr3,[d]
5.  cmp  Pr3,0
6.  be   block3
```



(Assume only loads can stall.)

Critical Path Heuristic

This value is also computed on a per-basic block basis.

It estimates how long it will take to execute Instruction I, and all I’s successors, assuming unlimited parallelism.

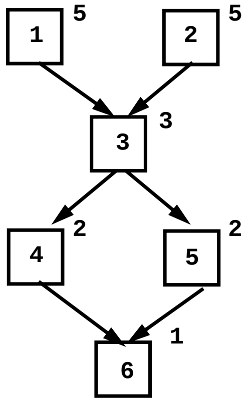
$E(I)$ = Execution time for instruction I (normally 1 for pipelined machines)

$CP(I) = E(I)$ if I is a leaf.

$$CP(I) = E(I) + \underset{J_i \in \text{Succ}(I)}{\text{Max}} (CP(J_i)+d(I,J_i))$$

Example of Critical Path Values

```
block1:
1.  ld   [a],Pr1
2.  ld   [b],Pr2
3.  add  Pr1,Pr2,Pr3
4.  st   Pr3,[d]
5.  cmp  Pr3,0
6.  be   block3
```



Selecting Instructions to Issue

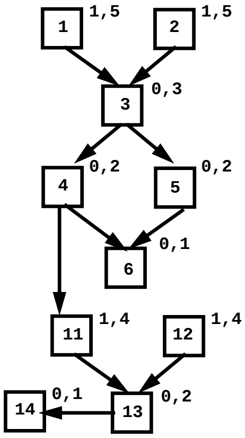
From the Ready Set (instructions with all dependencies satisfied, and which will not stall) use the following priority rules:

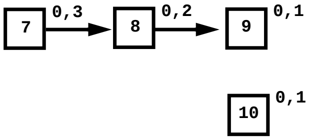
1. Instructions in block A and blocks equivalent to A have priority over other (speculative) blocks.
2. Instructions with the highest D values have priority.
3. Instructions with the highest CP values have priority.

These rules imply that we schedule useful instructions before speculative ones, instructions on paths with potentially many stalls over those with fewer stalls, and instructions on critical paths over those on non-critical paths.

Example

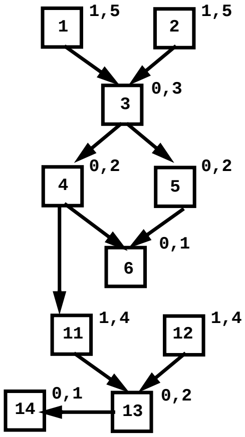
```
block1:
1.   ld   [a],Pr1
2.   ld   [b],Pr2
3.   add  Pr1,Pr2,Pr3
4.   st   Pr3,[d]
5.   cmp  Pr3,0
6.   be   block3
block2:
7.   mov  1,Pr4
8.   st   Pr4,[flag]
9.   b    block4
block3:
10.  st   0,[flag]
block4:
11.  ld   [d],Pr5
12.  ld   [g],Pr6
13.  sub  Pr5,Pr6,Pr7
14.  st   Pr7,[f]
```

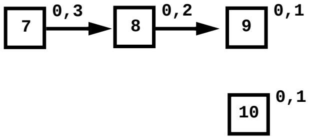




We'll schedule without speculation; highest D values first, then highest CP values.

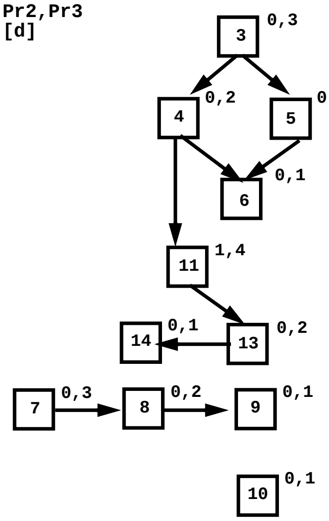
```
block1:
1.   ld   [a],Pr1
2.   ld   [b],Pr2
12.  ld   [g],Pr6
```





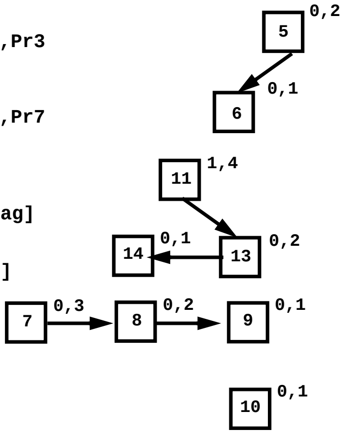
Next, come Instructions 3 and 4.

```
block1:
1.   ld   [a],Pr1
2.   ld   [b],Pr2
12.  ld   [g],Pr6
3.   add  Pr1,Pr2,Pr3
4.   st   Pr3,[d]
```



Now 11 can issue (D=1), followed by 5, 13, 6 and 14. Block B4 is now empty, so B3 and B4 are scheduled.

```
block1:
1.   ld   [a],Pr1
2.   ld   [b],Pr2
12.  ld   [g],Pr6
3.   add  Pr1,Pr2,Pr3
4.   st   Pr3,[d]
11.  ld   [d],Pr5
5.   cmp  Pr3,0
13.  sub  Pr5,Pr6,Pr7
6.   be   block3
14.  st   Pr7,[f]
block2:
7.   mov  1,Pr4
8.   st   Pr4,[flag]
9.   b    block4
block3:
10.  st   0,[flag]
block4:
```



There are no stalls. In fact, if we equivalence **Pr3** and **Pr5**, Instruction 11 can be removed.

Hardware Support for Global Code Motion

We want to be aggressive in scheduling loads, which incur high latencies when a cache miss occurs. In many cases, control and data dependencies may force us to restrict how far we may move a critical load.

Consider

```
p = Lookup(Id);
  ...
if (p != null)
   print(p.a);
```

It may well be that the object returned by **Lookup** is not in the L1 cache. Thus we'd like to schedule the load generated by **p.a** as soon as possible; ideally right after the lookup.

But moving the load above the **p != null** check is clearly unsafe.

A number of modern machine architectures, including Intel's Itanium, have proposed a *speculative load* to allow freer code motion when scheduling.

A speculative load,

```
ld.s  [adr],%reg
```

acts like an ordinary load as long as the load does not force an interrupt. If it does, the interrupt is suppressed and a special **NaT** (not a thing) bit is set in the register (a hidden 65th bit). A **NaT** bit can be propagated through instructions before being tested.

In some cases (like our table lookup example), a register containing a **NaT** bit may simply not be used because control doesn’t reach its intended uses.

However a **NaT** bit need not indicate an outright error. A load may force a TLB (translation lookaside buffer) fault or a page fault. These interrupts are probably too costly to do speculatively, but if we decide the loaded value is really needed, we will want to allow them.

A special check instruction, of the form,

```
chk.s  %reg,adr
```

checks whether **%reg** has its **NaT** bit set. If it does, control passes to **adr**, where user-supplied “fixup” code is placed. This code can redo the load non-speculatively, allowing necessary interrupts to occur.

Hardware Support for Data Speculation

In addition to supporting control speculation (moving instructions above conditional branches), it is useful to have hardware support for data speculation.

In data speculation, we may move a load above a store if we believe the chance of the load and store conflicting is slim.

Consider a variant of our earlier lookup example,

```
p = Lookup(Id);
  ...
q.a = init();
print(p.a);
```

We’d like to move the load implied by **p.a** above the assignment to **q.a**. This allows **p** to miss in the L1 cache, using the execution of **init()** to cover the miss latency.

But, we need to be sure that **q** and **p** don’t reference the same object and that **init()** doesn’t indirectly change **p.a**. Both possibilities may be remote, but proving non-interference may be difficult.

The Intel Itanium provides a special “advanced load” that supports this sort of load motion.

The instruction

```
ld.a  [adr],%reg
```

loads the contents of memory location **adr** into **%reg**. It also stores **adr** into

special *ALAT* (Advanced Load Address Table) hardware.

When a store to address **X** occurs, an ALAT entry corresponding to address **X** is removed (if one exists).

When we wish to use the contents of **%reg**, we execute a

```
ld.c  [adr],%reg
```

instruction (a *checked* load).

If an ALAT entry for **adr** is present, this instruction does nothing; **%reg** contains the correct value. If there is no corresponding ALAT entry, the **ld.c** simply acts like an ordinary load.

(Two versions of **ld.c** exist; one preserves an ALAT entry while the other purges it).

And yes, a speculative load (**`ld.s`**) and an advanced load (**`ld.a`**) may be combined to form a speculative advanced load (**`ld.sa`**).

Speculative Multi-threaded Processors

The problem of moving a load above a store that may conflict with it also appears in multi-threaded processors.

How do we know that two threads don’t interfere with one another by writing into locations both use?

Proofs of non-interference can be difficult or impossible. Rather than severely restrict what independent threads can do, researchers have proposed *speculative* multi-threaded processors.

In such processors, one thread is primary, while all other threads are secondary and speculative. Using hardware tables to remember locations read and written, a secondary thread can commit (make its updates permanent) only if it hasn’t read locations the primary thread later wrote and hasn’t written locations the primary thread read or wrote. Access conflicts are automatically detected, and secondary threads are automatically restarted as necessary to preserve the illusion of serial memory accesses.

Reading Assignment

- Read Section 15.5, “Automatic Instruction Selection,” from Chapter 15.
- Read Pelegri-Llopart and Graham’s paper, “Optimal Code Generation from Expression Trees.”
- Read Fraser, Henry and Proebsting’s paper, “BURG--Fast Optimal Instruction Selection and Tree Parsing.”

Software Pipelining

Often loop bodies are too small to allow effective code scheduling. But loop bodies, being “hot spots,” are exactly where scheduling is most important.

Consider

```
   void f (int a[],int last) {
     for (p=&a[0];p!=&a[last];p++)
        (*p)++;
   }
```

The body of the loop might be:

```
   L: ld   [%g3],%g2
      nop
      add  %g2,1,%g2
      st   %g2,[%g3]
      add  %g3,4,%g3
      cmp  %g3,%g4
      bne  L
      nop
```

Scheduling this loop body in isolation is ineffective—each instruction depends upon its immediate predecessor.

So we have a loop body that takes 8 cycles to execute 6 “core” instructions.

We could unroll the loop body, but for how many iterations? What if the loop ends in the “middle” of an expanded loop body? Will extra registers be a problem?

In this case *software pipelining* offers a nice solution. We expand the loop body *symbolically*, intermixing instructions from several iterations. Instructions can overlap, increasing parallelism and forming a “tighter” loop body:

```
      ld   [%g3],%g2
      nop
      add  %g2,1,%g2
   L: st   %g2,[%g3]
      add  %g3,4,%g3
      ld   [%g3],%g2
      cmp  %g3,%g4
      bne  L
      add  %g2,1,%g2
```

Now the loop body is ideal—exactly 6 instructions. Also, no extra registers are needed!

But, we do “overshoot” the end of the loop a bit, loading one element past the exit point. (How serious is this?)

Key Insight of Software Pipelining

Software pipelining exploits the fact that a loop of the form **{A B C}**^{n}, where **A**, **B** and **C** are individual instructions, and **n** is the iteration count, is equivalent to **A {B C A}**$^{n-1}$ **B C** and is also equivalent to **A B {C A B}**$^{n-1}$ **C**.

Mixing instructions from several iterations may increase the effectiveness of code scheduling, and may perhaps allow for more parallel execution.

Software Pipelining is Hard

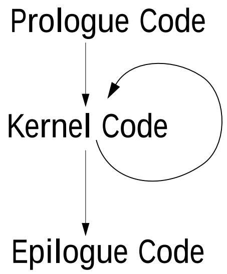
In fact, it is NP-complete:

> Hsu and Davidson, “Highly concurrent scalar processing,” 13th ISCA (1986).

The Iteration Interval

We seek to initiate the next iteration of a loop as soon as possible, squeezing each iteration of the loop body into as few machine cycles as possible.

The general form of a software pipelined loop is:



Prologue Code

Kernel Code

Epilogue Code

The prologue code “sets up” the main loop, and the epilogue code “cleans up” after loop termination. Neither the prolog nor the epilogue need be optimized, since they execute only once.

Optimizing the kernel is key in software pipelining. The kernel’s execution time (in cycles) is called the *initiation interval (II)*; it measures how quickly the next iteration of a loop can start.

We want the smallest possible initiation interval. Determining the smallest viable II is itself NP-complete. Because of parallel issue and execution in superscalar and multiple issue processors, very small II values are possible (even less than 1!)