Selecting Instructions to Issue

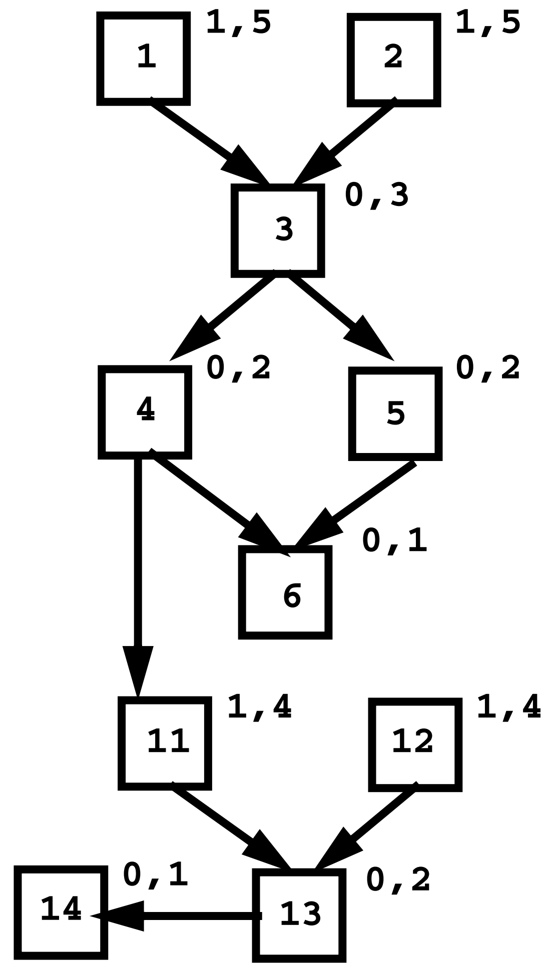
From the Ready Set (instructions with all dependencies satisfied, and which will not stall) use the following priority rules:

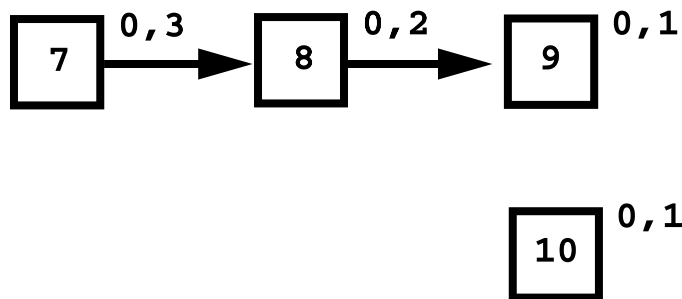
1. Instructions in block A and blocks equivalent to A have priority over other (speculative) blocks.
2. Instructions with the highest D values have priority.
3. Instructions with the highest CP values have priority.

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

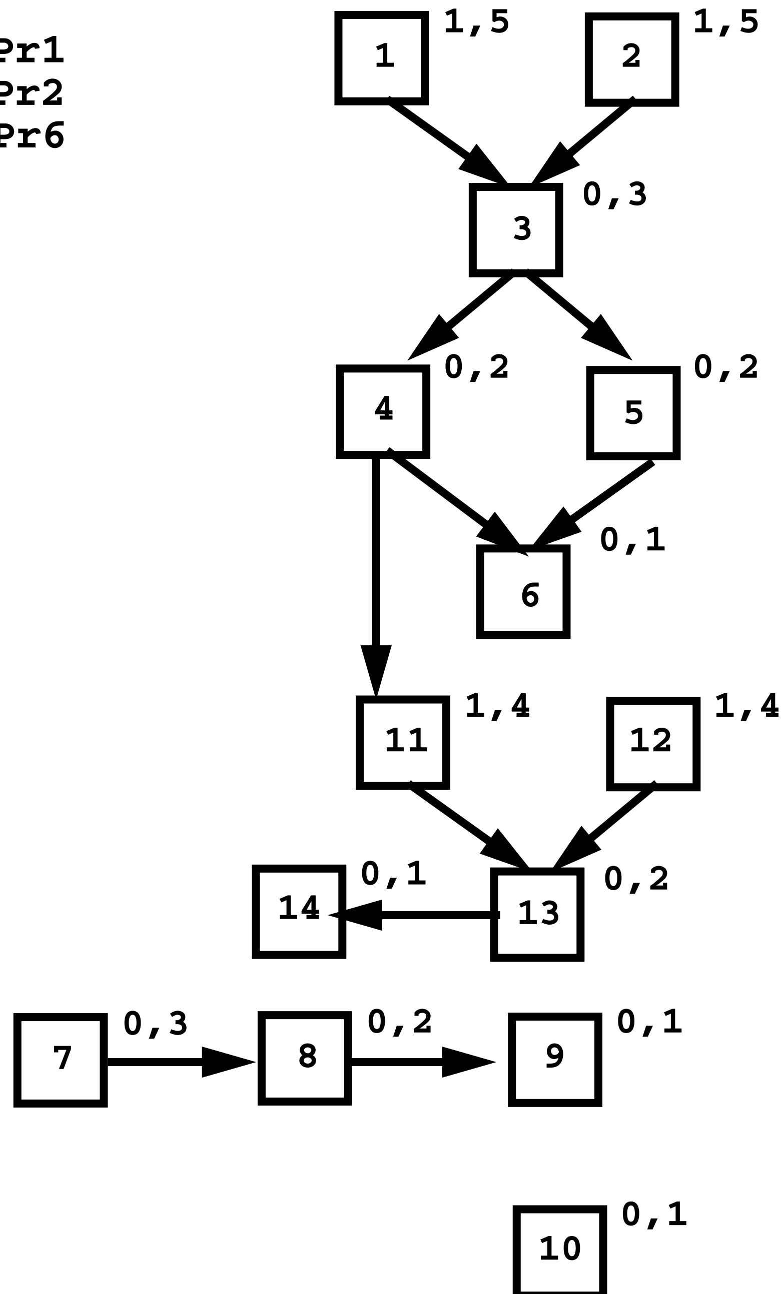
```
block1:
1.   ld   [a],Pr1
2.   ld   [b],Pr2
3.   add  Pr1,Pr2,Pr3
4.   st   Pr3,[d]
5.   cmp  Pr3,0
6.   be   block3
block2:
7.   mov  1,Pr4
8.   st   Pr4,[flag]
9.   b    block4
block3:
10.  st   0,[flag]
block4:
11.  ld   [d],Pr5
12.  ld   [g],Pr6
13.  sub  Pr5,Pr6,Pr7
14.  st   Pr7,[f]
```





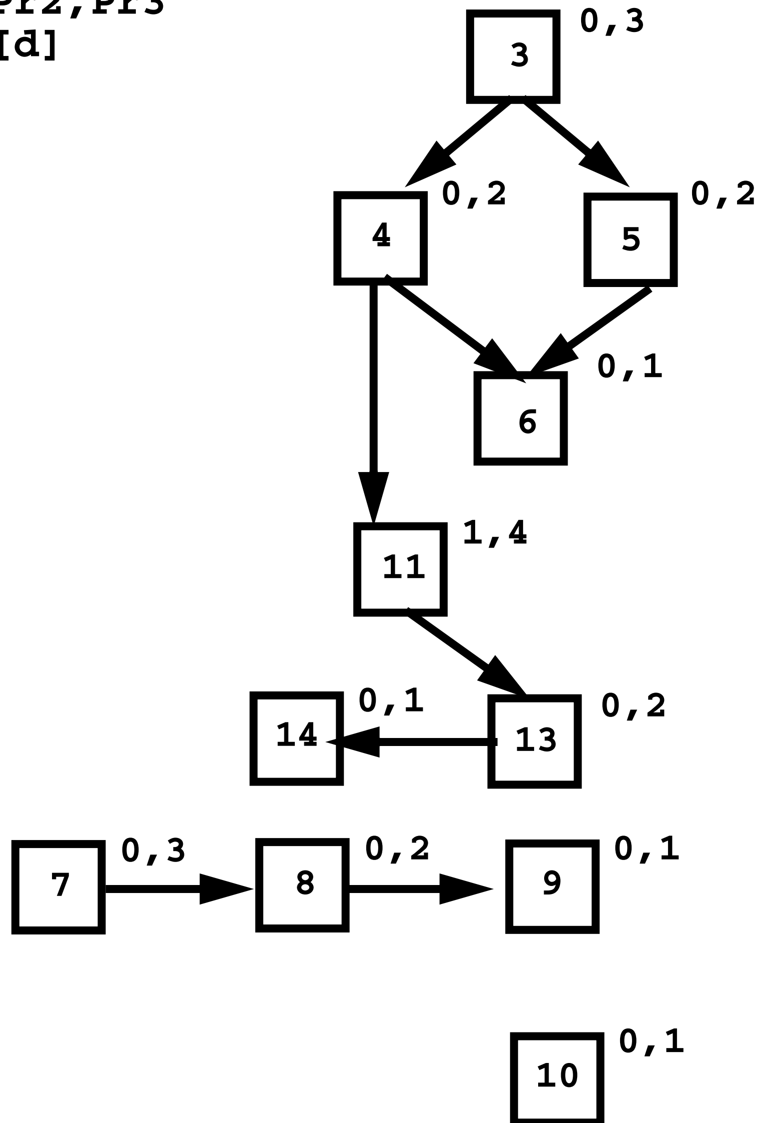
We'll schedule without speculation; highest D values first, then highest CP values.

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



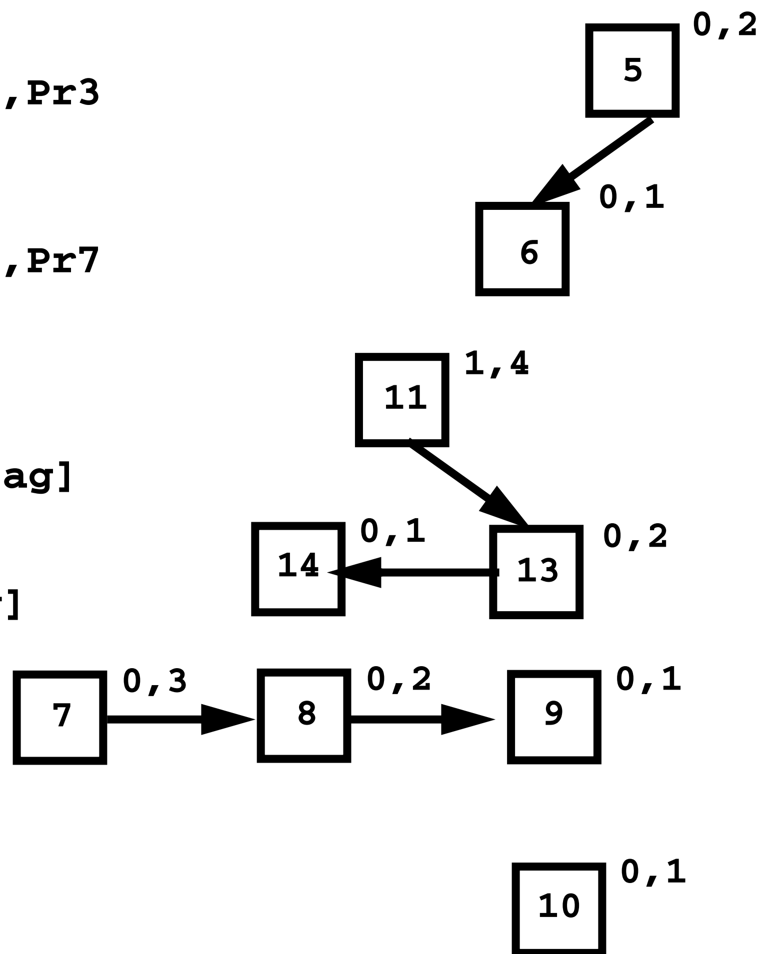
Next, come Instructions 3 and 4.

```
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 B2 and B3 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

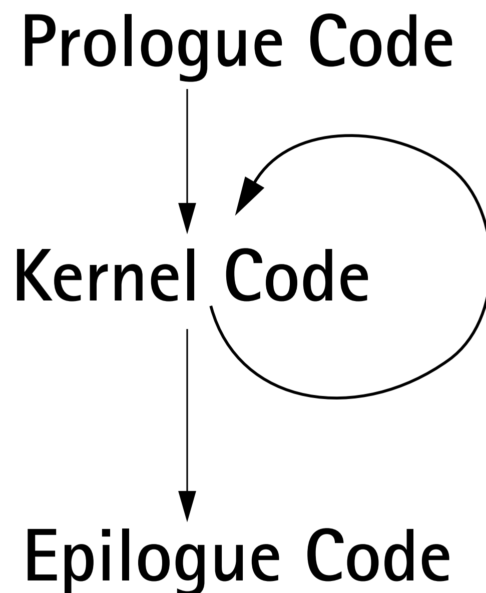
In fact, it is NP-complete:

> 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:



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!)

Factors that Limit the Size of the Initiation Interval

We want the initiation interval to be as small as possible. Two factors limit how small the II can become:

- Resource Constraints
- Dependency Constraints

Resource Constraints

A small II normally means that we are doing steps of several iterations simultaneously. The number of registers and functional units (that execute instructions) can become limiting factors of the size of II.

For example, if a loop body contains 4 floating point operations, and our processor can issue and execute no more than 2 floating point operations per cycle, then the loop's II can't be less than 2.

Dependency Constraints

A loop body can often contain a *loop-carried dependence*. This means one iteration of a loop depends on values computed in an earlier iteration. For example, in

```
void f (int a[]) {
  for (i=1;i<1000;i++)
     a[i]=(a[i-1]+a[i])/2;
}
```

there is a loop carried dependence from the use of `a[i-1]` to the computation of `a[i]` in the previous iteration. This means the computation of `a[i]` can't begin until the computation of `a[i-1]` is completed.

Let's look at the code that might be generated for this loop:

```
f:
    mov   %o0, %o2          !a in %o2
    mov   1, %o1            !i=1 in %o1
L:
    sll   %o1, 2, %o0       !i*4 in %o0
    add   %o0, %o2, %g2     !&a[i] in %g2
☛   ld    [%g2-4], %g2      !a[i-1] in %g2
    ld    [%o2+%o0], %g3    !a[i] in %g3
☛   add   %g2, %g3, %g2     !a[i-1]+a[i]
☛   srl   %g2, 31, %g3      !s=0 or 1=sign
☛   add   %g2, %g3, %g2     !a[i-1]+a[i]+s
☛   sra   %g2, 1, %g2       !a[i-1]+a[i]/2
    add   %o1, 1, %o1       !i++
    cmp   %o1, 999
    ble   L
☛   st    %g2, [%o2+%o0]    !store a[i]
    retl
    nop
```

The 6 marked instructions form a cyclic dependency chain from a use of `a[i-1]` to its computation (as `a[i]`) in the previous cycle. This cycle means that the loop's II can never be less than 6.

Modulo Scheduling

There are many approaches to software pipelining. One of the simplest, and best known, is *modulo scheduling*. Modulo scheduling builds upon the postpass basic block schedulers we've already studied.

First, we estimate the II of the loop we will create. How?

We can compute the minimum II based on resource considerations (II_{res}) and the minimum II based on cyclic loop-carried dependencies (II_{dep}). Then $\max(II_{res}, II_{dep})$ is a reasonable estimate of the best possible II. We'll try to build a loop with a kernel size of II. If this fails, we'll try II+1, II+2, etc.

In modulo scheduling we'll schedule instructions one by one, using the dependency dag and whatever heuristic we prefer to choose among multiple roots.

Now though, if we place an instruction at cycle c (many independent instructions may execute in the same cycle), then we'll place additional copies of the instruction at cycle c+II, c+2*II, etc.

Placement must respect dependency constraints and resource limits at all positions. We consider placements only until a kernel (of size II) forms. The kernel must begin before cycle s-1, where s is the size of the loop body (in instructions). The loop's conditional branch is placed *after* the kernel is formed.

If we can't form a kernel of size II (because of dependency or resource conflicts), we increase II by 1 and try again. At worst, we get a kernel equal in size to the original loop body, which guarantees that the modulo scheduler eventually terminates.

Depending on how many iterations are intermixed in the kernel, the loop termination condition may need to be adjusted (since the initial and final iterations may appear as part of the loop prologue and epilogue).

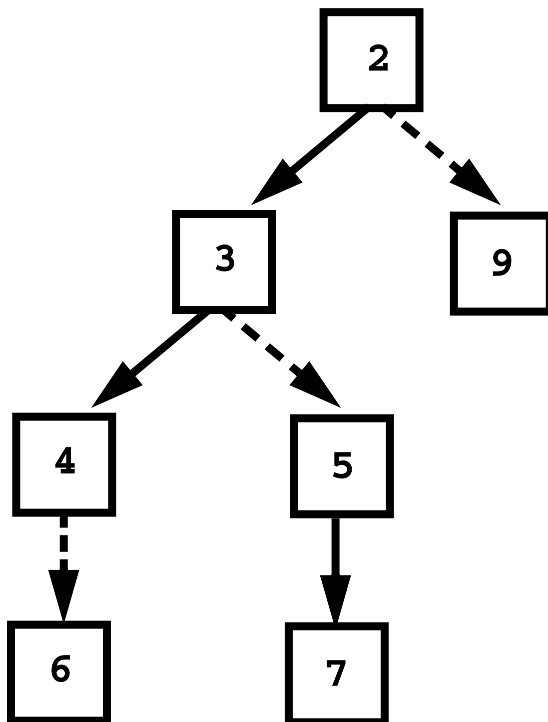
Example

Consider the following simple function which adds an array index to each element of an array and copies the results into a second array:

```
void f (int a[],int b[]) {
  t1 = &a[0];
  t2 = &b[0];
  for (i=0;i<1000;i++,t1++,t2++)
      *t1 = *t2 + i;
}
```

The code for `f` (compiled as a leaf procedure) is:

```
1.   f:   mov    0, %g3
2.   L:   ld     [%o1], %g2
3.        add    %g3, %g2, %g4
4.        st     %g4, [%o0]
5.        add    %g3, 1, %g3
6.        add    %o0, 4, %o0
7.        cmp    %g3, 999
8.        ble    L
9.        add    %o1, 4, %o1
10.       retl
11.       nop
```



Dashed arcs are anti dependencies.

We’ll software pipeline the loop body, excluding the conditional branch (which is placed after the loop kernel is formed).

This loop body contains 2 loads/ stores, 5 arithmetic and logical operations (including the compare) and one conditional branch.

Let’s assume the processor we are compiling for has 1 load/store unit, 3 arithmetic/logic units, and 1 branch unit. That means the processor can (ideally) issue and execute simultaneously 1 load or store, 3 arithmetic and logic instructions, and 1 branch. Thus its maximum issue width is 5. (Current superscalars have roughly this capability.)

Considering resource requirements, we will need at least two cycles to process the contents of the loop body. There are no loop-carried dependencies.

Thus we will estimate this loop's best possible Initiation Interval to be 2.

Since the only instruction that can stall is the root of the dependency dag, we'll schedule using estimated critical path length, which is just the node's height in the tree. Hence we'll schedule the nodes in the order: 2,3,4,5,6,7,9.

We'll schedule all instructions in a legal execution order (respecting dependencies), and we'll try to choose as many instructions as possible to execute in the same cycle.

Starting with the root, instruction 2, we schedule it at cycles 1, 3 (=1+II), 5 (=1+2*II):

```
cycle       instruction
1.           ld    [%o1], %g2
2.
3.           ld    [%o1], %g2
4.
5.           ld    [%o1], %g2
```

No conflicts so far, since each of the loads starts an independent iteration.

We'll schedule instruction 3 next. It must be placed at cycles 3, 5 and 7 since it uses the result of the load.

```
cycle      instruction
1.          ld    [%o1], %g2
2.
3.          add   %g3, %g2, %g4
3.          ld    [%o1], %g2
4.
5.          add   %g3, %g2, %g4
5.          ld    [%o1], %g2
6.
7.          add   %g3, %g2, %g4
```

Note that in cycles 3 and 5 we use the current value of **`%g2`** *and* initiate a load into **`%g2`**.

Instruction 4 is next. It uses the result of the add we just scheduled, so it is placed at cycles 4 and 6.

```
cycle      instruction
1.          ld    [%o1], %g2
2.
3.          add   %g3, %g2, %g4
3.          ld    [%o1], %g2
4.          st    %g4, [%o0]
5.          add   %g3, %g2, %g4
5.          ld    [%o1], %g2
6.          st    %g4, [%o0]
7.          add   %g3, %g2, %g4
```

Instruction 5 is next. It is anti dependent on instruction 3, so we can place it in the same cycles that 3 uses (3, 5 and 7).

```
cycle       instruction
1.           ld    [%o1], %g2
2.
3.           add   %g3, %g2, %g4
3.           ld    [%o1], %g2
3.           add   %g3, 1, %g3
4.           st    %g4, [%o0]
5.           add   %g3, %g2, %g4
5.           ld    [%o1], %g2
5.           add   %g3, 1, %g3
6.           st    %g4, [%o0]
7.           add   %g3, %g2, %g4
7.           add   %g3, 1, %g3
```

Instruction 6 is next. It is anti dependent on instruction 4, so we can place it in the same cycles that 4 uses (4 and 6).

```
cycle       instruction
1.           ld    [%o1], %g2
2.
3.           add   %g3, %g2, %g4
3.           ld    [%o1], %g2
3.           add   %g3, 1, %g3
4.           st    %g4, [%o0]
4.           add   %o0, 4, %o0
5.           add   %g3, %g2, %g4
5.           ld    [%o1], %g2
5.           add   %g3, 1, %g3
6.           st    %g4, [%o0]
6.           add   %o0, 4, %o0
7.           add   %g3, %g2, %g4
7.           add   %g3, 1, %g3
```

Next we place instruction 7. It uses the result of instruction 5 (`%g3`), so it is placed in the cycles following instruction 5 (4 and 6).

```
cycle      instruction
1.          ld     [%o1], %g2
2.
3.          add    %g3, %g2, %g4
3.          ld     [%o1], %g2
3.          add    %g3, 1, %g3
4.          st     %g4, [%o0]
4.          add    %o0, 4, %o0
4.          cmp    %g3, 999
5.          add    %g3, %g2, %g4
5.          ld     [%o1], %g2
5.          add    %g3, 1, %g3
6.          st     %g4, [%o0]
6.          add    %o0, 4, %o0
6.          cmp    %g3, 999
7.          add    %g3, %g2, %g4
7.          add    %g3, 1, %g3
```

Finally we place instruction 9. It is anti dependent on instruction 2 so it is placed in the same cycles as instruction 2 (1, 3 and 5).

```
cycle          instruction
1.              ld    [%o1], %g2
1.              add   %o1, 4, %o1
3.              add   %g3, %g2, %g4
3.              ld    [%o1], %g2
3.              add   %o1, 4, %o1
3.              add   %g3, 1, %g3
4.              st    %g4, [%o0]
4.              add   %o0, 4, %o0
4.              cmp   %g3, 999
5.              add   %g3, %g2, %g4
5.              ld    [%o1], %g2
5.              add   %o1, 4, %o1
5.              add   %g3, 1, %g3
6.              st    %g4, [%o0]
6.              add   %o0, 4, %o0
6.              cmp   %g3, 999
7.              add   %g3, %g2, %g4
7.              add   %g3, 1, %g3
```

We look for a 2 cycles kernel that contains all 7 instructions of the loop body that we have scheduled. We also want a kernel that sets the condition code (via the cmp) during its first cycle so that it can be tested during its second (and final) cycle. Cycles 4 and 5 meet these criteria, and will form our kernel.

We place the conditional branch just before the last instruction in cycle 5 (to give the conditional branch a useful instruction for its delay slot).

We now have:

```
cycle        instruction
1.              ld     [%o1], %g2
1.              add    %o1, 4, %o1
3.              add    %g3, %g2, %g4
3.              ld     [%o1], %g2
3.              add    %o1, 4, %o1
3.              add    %g3, 1, %g3
4.        L:    st     %g4, [%o0]
4.              add    %o0, 4, %o0
4.              cmp    %g3, 999
5.              add    %g3, %g2, %g4
5.              ld     [%o1], %g2
5.              add    %o1, 4, %o1
5.              ble    L
5.              add    %g3, 1, %g3
6.              st     %g4, [%o0]
6.              add    %o0, 4, %o0
6.              cmp    %g3, 999
7.              add    %g3, %g2, %g4
7.              add    %g3, 1, %g3
```

A couple of final issues must be dealt with:

- Does the iteration count need to be changed?
  In this case no, since the final valid value of `i`, 999, is used to compute `%g4` in cycle 5, before the loop exits.
- What instructions do we keep as the loop's epilogue?
  None! Instructions past the kernel aren't needed since they are part of future iterations (past `i==999`)which aren't needed or wanted.
- Note that `b[1000]` and `b[1001]` are "touched" even though they are never used. This is probably OK as long as arrays aren't placed at the very end of a page or segment.

Our final loop is:

```
cycle       instruction
1.          ld     [%o1], %g2        !N0
1.          add    %o1, 4, %o1       !N0
3.          add    %g3, %g2, %g4     !N0
3.          ld     [%o1], %g2        !N1
3.          add    %o1, 4, %o1       !N1
3.          add    %g3, 1, %g3       !N0
4.      L:  st     %g4, [%o0]        !N0
4.          add    %o0, 4, %o0       !N0
4.          cmp    %g3, 999          !N0
5.          add    %g3, %g2, %g4     !N1
5.          ld     [%o1], %g2        !N2
5.          add    %o1, 4, %o1       !N2
5.          ble    L                 !N0
5.          add    %g3, 1, %g3       !N1
```

This is very efficient code—we use the full parallelism of the processor, executing 5 instructions in cycle 5 and 8 instructions in just 2 cycles. All resource limitations are respected.