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 $\mathbf{\{A\ B\ C\}^n}$, where **A**, **B** and **C** are individual instructions, and **n** is the iteration count, is equivalent to $\mathbf{A\ \{B\ C\ A\}^{n-1}\ B\ C}$ and is also equivalent to $\mathbf{A\ B\ \{C\ A\ B\}^{n-1}\ C}$.

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

Software Pipelining is Hard

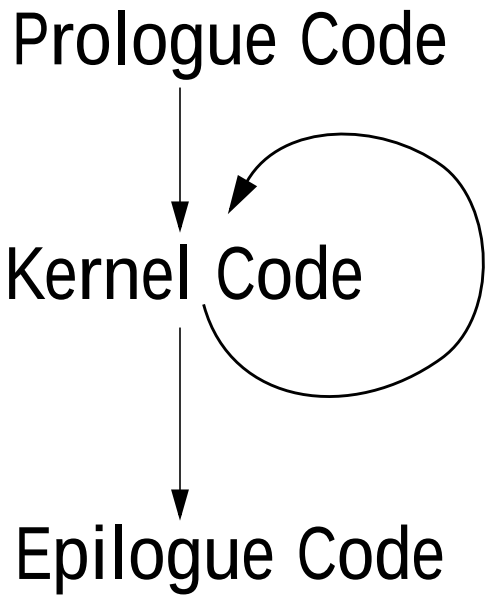
In fact, it is NP-complete:

> 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).

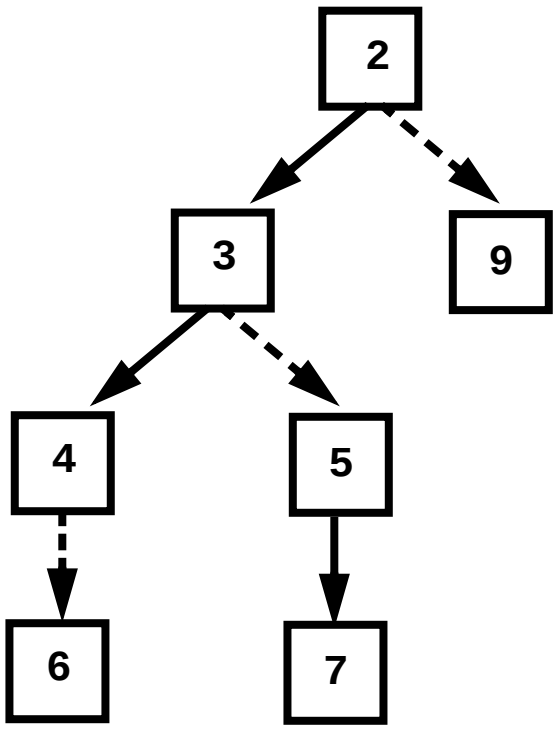
Example

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
7.          add   %g3, %g2, %g4
7.          add   %g3, 1, %g3
7.          cmp   %g3, 999
```

Finally we place instruction 9. It is anti dependent on instruction 2 so it is placed in the same cycles as instruction 3 (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
7.           add   %g3, %g2, %g4
7.           add   %g3, 1, %g3
7.           cmp   %g3, 999
```

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
7.            add   %g3, %g2, %g4
7.            add   %g3, 1, %g3
7.            cmp   %g3, 999
```

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.

False Dependencies & Loop Unrolling

A limiting factor in how "tightly" we can software pipeline a loop is reuse of registers and the false dependencies reuse induces.

Consider the following simple function that copies array elements:

```
void f (int a[],int b[], int lim) {
  for (i=0;i<lim;i++)
     a[i]=b[i];
}
```

The loop that is generated takes 3 cycles:

```
cycle        instruction
1.     L:     ld    [%g3+%o1], %g2
1.            addcc %o2, -1, %o2
3.            st    %g2, [%g3+%o0]
3.            bne   L
3.            add   %g3, 4, %g3
```

We'd like to tighten the iteration interval to 2 or less. One cycle is unlikely, since doing a load and a store in the same cycle is problematic (due to a possible dependence through memory).

If we try to use modulo scheduling, we can't put a second copy of the load in cycle 2 because it would overwrite the contents of the first load. A load in cycle 3 will clash with the store.

The solution is to unroll the loop into two copies, using different registers to hold the contents of the load and the current offset into the arrays.

The use of a "count down" register to test for loop termination is helpful,

since it allows an easy exit from the middle of the loop.

With the renaming of the registers used in the two expanded iterations, scheduling to “tighten” the loop is effective.

After expansion we have:

```
cycle        instruction
1.      L:    ld    [%g3+%o1], %g2
1.            addcc %o2, -1, %o2
3.            st    %g2, [%g3+%o0]
3.            beq   L2
3.            add   %g3, 4, %g4
4.            ld    [%g4+%o1], %g5
4.            addcc %o2, -1, %o2
6.            st    %g5, [%g4+%o0]
6.            bne   L
6.            add   %g4, 4, %g3
        L2:
```

We still have 3 cycles per iteration, because we haven’t scheduled yet.

Now we can move the increment of **%g3** (into **%g4**) above other uses of **%g3**. Moreover, we can move the load into **%g5** *above* the store from **%g2** (*if the load and store are independent*):

```
cycle       instruction
1.     L:    ld    [%g3+%o1], %g2
1.           addcc %o2, -1, %o2
1.           add   %g3, 4, %g4
2.           ld    [%g4+%o1], %g5
3.           st    %g2, [%g3+%o0]
3.           beq   L2
3.           addcc %o2, -1, %o2
4.           st    %g5, [%g4+%o0]
4.           bne   L
4.           add   %g4, 4, %g3
      L2:
```

We can normally test whether **%g4+%o1** and **%g3+%o0** can be equal at compile-time, by looking at the actual array parameters. (Can **&a[0] == &b[1]**?)