419 Compilers II

Tutorial exercise 2: instruction scheduling

IC Inside™ Computers Incorporated have employed you to work on the design of their new CPU, the 0898. There is essentially no limit to the number of transistors available for implementing their processor, so they are keen to investigate highly-parallel designs. Your job, as the head of the compiler team, is to evaluate how effectively these highly-parallel designs can be used.

The proposed machine will be able to issue N instructions per clock cycle. This is done by packing N instructions into a single, long instruction word. The machine will have many registers, and a given register r can be read by several instructions within a single instruction word. Only one instruction per word can write to r, and this happens after the reads.

The machine is constructed using fully-pipelined floating point units, which take 3 cycles to complete an add and 4 cycles to complete a multiply.

The main restriction is that only two double-precision floating point words can be read from memory per clock cycle, and only one can be written to memory. The memory read system is fully pipelined and takes 2 clock cycles to deliver a result to a register.

Example: add a constant to a vector

Source code:

```
for i = 0 to 99 do
  A[i] = B[i] + 1.0
enddo
```

After induction variable recognition and elimination of `i`:

```
double A[100], B[100];
double *pA = A;
double *pB = B;
while ( pA < &A[100] ) {
  *pA = *pB + 1.0;
  ++pA;
  ++pB;
}
```

Sensible assembler code for a single-issue machine (e.g. MIPS R4000):

```
  ld   r2, A       ! r2 holds pA
```

```
  ld   r3, B       ! r3 holds pB
  ld   r4, #A+800  ! loop bound
  ldd  f0, 1.0     ! load 1.0 into f0 (somehow)
L:
  ldd  f1, 0(r3)
!  <two cycle stall waiting for ldd>
  addd f0, f1, f2
  addi #8, r2      ! addi's fill some of addd delay
  addi #8, r3
!  <one cycle stall waiting for addd>
  sdd  f2, 0(r2)
  blt  r2,r4,L
```

Assuming no branch delays, this should run at 9 clock cycles per iteration.

Using software pipelining, the code for the IC Inside 0898 would look like this:

```
  ld   r2, A       ld   r3, B         ld   r4, #A+800    ldd  f0, 1.0
  ldd  f1, 0(r3)                      ! prologue code
  ldd  f1, 0(r3)   addd f0, f1, f2    !

L:
  sdd  f2, 0(r2)   addd f0, f1, f2    ldd  f1, 0(r3)     ! continued
    addi #8, r2      addi #8, r3        blt  r1,r4,L

  sdd  f2, 0(r2)   addd f0, f1, f2    ! epilogue code
  sdd  f2, 0(r2)
```

Note that the loop is now just one (long) instruction.

Unfortunately, although this code would work, it would perform badly because there would still be a stall between iterations of `L`, while waiting for the results from the `addd` (and `ldd`). It's better than the first version thanks to multiple issue – four cycles per iteration. However, we are missing three opportunities each iteration for injecting new operands into the floating point pipeline.

To overcome this, we need a slightly more sophisticated scheme (called *modulo variable expansion*:

```
  ld   r1, #0      ld   r2, A           ld   r3, B        ! continued
    ld   r4, #A+800  ldd  f0, 1.0
  ldd  f01, 0(r3)  addd f00, f01, f02   ! prologue code;
  ldd  f11, 0(r3)  addd f10, f11, f12   !  (some compaction possible)
  ldd  f21, 0(r3)  addd f20, f21, f22
  ldd  f31, 0(r3)  addd f30, f31, f32
```

```
L:
  sdd  f02, 0(r2)  addd f00, f01, f02   ldd  f01, 0(r3)
  sdd  f12, 8(r2)  addd f10, f11, f12   ldd  f11, 8(r3)
  sdd  f22, 16(r2) addd f20, f21, f22   ldd  f21, 16(r3)
  sdd  f23, 24(r2) addd f30, f31, f32   ldd  f31, 24(r3)  ! continued
    addi #32, r2     addi #32, r3         blt  r1,r4,L


  sdd  f02, 0(r2)  addd f00, f01, f02  ! epilogue code
  sdd  f02, 0(r2)
  sdd  f12, 0(r2)  addd f10, f11, f12  ! (some compaction possible)
  sdd  f12, 0(r2)
  sdd  f22, 0(r2)  addd f20, f21, f22
  sdd  f22, 0(r2)
  sdd  f32, 0(r2)  addd f30, f31, f32
  sdd  f32, 0(r2)
```

We have to use different registers for each instruction; I have used a systematic register numbering to clarify what's going on but any registers would do.

This version of the loop runs at one cycle per iteration.

Question 1

Consider the following loop:

```
for i = L to U do
   A[i] = A[i-k] + B[i]
```

Suppose this loop is to be implemented on the 0898 as in the example above. Explain,

1. what happens if k is fairly large, e.g. $k > 10$.
2. what happens if $k = 1$.
3. what happens if k is negative.
4. what is the smallest positive value for k for which the straightforward implementation would work properly?

Question 2

Consider the following loop:

```
for i = 0 to 99 do
   for j = 1 to 99 do
     A[i,j] = A[i,j-1] + B[i,j]
```

Estimate the performance (number of clock cycles per iteration) of this loop. Estimate the performance after interchanging the loop nest.

Up to now we have assumed an ideal memory system so that loads complete in two cycles. What memory access problems might arise? How might both the pipeline parallelism and the memory access performance be optimised simultaneously?

Question 3

1. How would you decide how many floating point units will be needed to gain maximum performance on realistic application programs?
2. How would you decide how many registers will be needed to gain maximum performance on realistic application programs?
3. How would you decide how many instructions per instruction word will be needed to gain maximum performance on realistic application programs?

Paul Kelly, Imperial College, February 1997.

419 Compilers II Tutorial 1

Sample Solutions

Question 1

Consider the following loop:

```
for i = L to U do
   A[i] = A[i-k] + B[i]
```

Suppose this loop is to be implemented on the 0898 as in the example above. Explain,

1. what happens if k is fairly large, e.g. $k > 10$.

   k is the loop's dependence distance; if $k > 0$ there is a loop-carried data dependence.

   Provided the correct value for `A[i-k]` is loaded, the code should be very similar to the example - something like this:

   ```
   L:
     sdd  f02, 0(r2)  addd f00, f01, f02   ldd  f01, 0(r3)   ldd  f00, -k*8(r2)
     sdd  f12, 8(r2)  addd f10, f11, f12   ldd  f11, 8(r3)   ldd  f10, -k*8+8(r2)
     sdd  f22, 16(r2) addd f20, f21, f22   ldd  f21, 16(r3)  ldd  f20, -k*8+16(r2)
     sdd  f23, 24(r2) addd f30, f31, f32   ldd  f31, 24(r3)  ldd  f30, -k*8+24(r2)   ! cont'd
       addi #32, r2     addi #32, r3         blt  r1,r4,L
   ```

   (with appropriate prologue/epilogue code).

2. what happens if $k = 1$.

   In this case the wrong value of `A[i-k]` will be loaded, since the store instruction which generates it will not yet have been executed.

   The problem is that the scheduling transformation to move the `ldd` instruction backwards over the `sdd` instruction of the previous iteration was invalid.

   The loop is fundamentally sequential, but performance can be improved slightly by using a register to carry the value of `A[i-k]` from iteration to iteration.

   For maximum performance, however, a different algorithm is needed.

3. what happens if k is negative.

   Although there would then be a loop-carried anti-dependence, it is not a problem. The reason is that no load has been scheduled to occur *later* than the store from its iteration.

Question 2

Consider the following loop:

```
for i = 0 to 99 do
   for j = 1 to 99 do
     A[i,j] = A[i,j-1] + B[i,j]
```

Estimate the performance (number of clock cycles per iteration) of this loop. Estimate the performance after interchanging the loop nest.

This will perform very badly because of the data dependence carried by the inner loop. At best we can expect 4 cycles/iteration because of the floating point add pipeline depth.

Interchanging the loop nest removes this dependence cycle, the loop should run at one cycle per iteration.

Up to now we have assumed an ideal memory system so that loads complete in two cycles. What memory access problems might arise? How might both the pipeline parallelism and the memory access performance be optimised simultaneously?

The problem is that the interchanged loop accesses the array with large stride. Suppose the machine has a 16-byte (two DP FP operands) cache line size. Before interchange, 50% of loads would be guaranteed to be cache hits because the previous load would have loaded a pair of adjacent words. After interchange, it is likely that the line will have been be replaced before the second element is accessed.

Replacement is unlikely if the iteration count of the inner loop is small. This is easily arranged by working in blocks. First strip mine:

```
for ii = 0 to 99  step BLKSIZE do
  for i = ii to ii+BLKSIZE-1 do
    for j = 1 to 99 do
      A[i,j] = A[i,j-1] + B[i,j]
```

Then interchange to increase parallelism:

```
for ii = 0 to 99  step BLKSIZE do
  for j = 1 to 99 do
    for i = ii to ii+BLKSIZE-1 do
      A[i,j] = A[i,j-1] + B[i,j]
```

Now select `BLKSIZE` so that the lines allocated when `A` and `B` are read are not replaced before the next iteration of the `j` loop.

Paul Kelly, Imperial College, February 1997.