Chapter 6

Performance Tuning for the R8000

This chapter outlines techniques for tuning the performance of your R8000 applications. It contains four sections:

- The first section presents the compiler optimization technique of software pipelining, which is crucial to getting optimal performance on the R8000. It shows you how to read your software pipelined code and how to understand what it does.
- The second section uses matrix multiplies as a case study on loop unrolling.
- The third section describes the phenomenon of bellows stalls on the R8000 architecture and gives tips on how to avoid them.
- The final section describes how the *IVDEP* directive can be used in Fortran to gain performance.

Software Pipelining

The purpose of this section is to give you an overview of software pipelining and to answer these questions:

- Why software pipelining delivers better performance
- How software pipelined code looks
- How to diagnose what happened when things go wrong

Why Software Pipelining Delivers Better Performance

To introduce the topic of software pipelining, let's consider the simple DAXPY loop (double precision *a* times *x* plus *y*) shown below.

```
DO i = 1, n

   v(i) = v(i) + X * w(i)
ENDDO
```

On the R8000 architecture, this can be coded as two load instructions followed by a *madd* instruction and a store. See Figure 6-1.

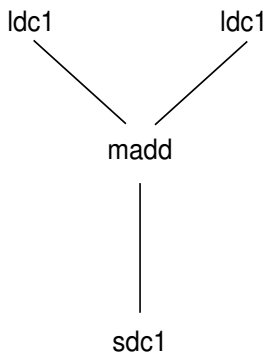

Figure 6-1 A Simple DAXPY Implementation

This simplest schedule achieves the desired result after five cycles. Since the R8000 architecture can allow up to two memory operations and two floating point operations in the same cycle, this simple example uses only 1/10th of the R8000's peak megaflops. There is also a delay of three cycles before the results of the *madd* can be stored.

```
0: ldc1 ldc1 madd
1:
2:
3:
4: sdc1
```

A loop unrolling by four schedule improves the performance to 1/4 th of the R8000's peak megaflops.

```
0: ldc1    ldc1    madd
1: ldc1    ldc1    madd
2: ldc1    ldc1    madd
3: ldc1    ldc1    madd
4: sdc1
5: sdc1
6: sdc1
7: sdc1
```

But this schedule does not take advantage of the R8000's ability to do two stores in one cycle.The best schedule that could be achieved would look like the following:

```
0: ldc1    ldc1
1: ldc1    ldc1    madd    madd
2: sdc1    sdc1
```

It uses 1/3 of the R8000's peak megaflops. But there still is a problem with the *madd* **sdc1** interlock delay. Software pipelining addresses this problem.

Software pipelining allows you to mix operations from different loop iterations in each iteration of the hardware loop. Thus the store instructions (which would interlock) in the above example could be used to store the results of different iterations. This can look like the following:

```
L1:
1: t1 = ldc1    t2 = ldc1   t3 = madd t1 X t2
2: t4 = ldc1    t5 = ldc1   t6 = madd t4 X t5
3: sdc1 t7      sdc1 t8     beq DONE

4: t1 = ldc1    t2 = ldc1   t7 = madd t1 X t2
5: t4 = ldc1    t5 = ldc1   t8 = madd t4 X t5
6: sdc1 t3      sdc1 t6     bne L1

DONE:
```

The stores in this loop are storing the *madd* results from previous iterations. But, in general, you could mix any operations from any number of different iterations. Also, note that every loop replication completes two loop iterations in 3 cycles.

In order to properly prepare for entry into such a loop, a windup section of code is added. The windup section sets up registers for the first stores in the main loop. In order to exit the loop properly, a winddown section is added. The winddown section performs the final stores. Any preparation of registers needed for the windown is done in the compensation section. The winddown section also prevents speculative operations.

```
windup:
1: t1 = ldc1    t2 = ldc1   t7 = madd t1 X t2
2: t4 = ldc1    t5 = ldc1   t8 = madd t4 X t5
L1:
1: t1 = ldc1    t2 = ldc1   t3 = madd t1 X t2
2: t4 = ldc1    t5 = ldc1   t6 = madd t4 X t5
3: sdc1 t7      sdc1 t8     beq compensation1
```

```
4: t1 = ldc1    t2 = ldc1   t7 = madd t1 X t2
5: t4 = ldc1    t5 = ldc1   t8 = madd t4 X t5
6: sdc1 t3      sdc1 t6     bne L1

winddown:
1: sdc1 t7      sdc1 t8     br ALLDONE

compensation1:
1: t7 = t3      t8 = t6
2: br winddown

ALLDONE:
```

Our example loop always does loads from at least 4 iterations, so we don't want to start it if we don't want speculative operations and if the trip count is less than 4. The following generalizes our example into a map of a software pipelined loop:

```
/* Precondition for unroll by 2 */
   do i = 1, n mod 2
      original loop
   enddo

   if ( n-i < 4 ) goto simple_loop
windup:
      ...
/* fp - fence post */
      fp = fp - peel_amount
      ...

swp replication 0:
      ...
      if ( i == fp ) goto compensation 0
      ...
swp replication n - 1:
      ...
      if ( i != fp ) goto swp replication 0

compensation n-1:

winddown:
   ...
   goto done

compensation 0:
/* Move registers to set up winddown */
```

```
   rx = ry
   ...
   goto winddown
   ...

compensation n - 2:
   ...
/* Move registers to set up winddown */
   rx = ry
   goto winddown

simple_loop:

   do i = i,n
      original_loop
   enddo

done:
```

In practice, software pipelining can make a huge difference. Sample benchmark performances see more than 100% improvement when compiled with software pipelining. This makes it clear that in order to get the best performance on the R8000 (**-mips4**), you should compile your application to use software pipelining (**-O3**).

Looking at the Code Produced by Software Pipelining

The proper way to look at the assembly code generated by software pipelining is to use the **-S** compiler switch. This is vastly superior to using the disassembler because the **-S** switch adds annotations to the assembly code which name out the sections described above. The annotations also provide useful statistics about the software pipelining process as well as reasons why certain code did not pipeline. To get a summary of these annotations do the following:

```
%f77 -64 -S -O3 -mips4 foo.f
```

This creates an annotated *.s* file

```
%grep '#<swp' foo.s
```

#<swpf is printed for loops that failed to software pipeline. #<*swps* is printed for statistics and other info about the loops that did software pipeline.

Another way to get a summary of the software pipelining annotations is to set the **-LIST:=ON** flag on the command line. For example:

```
%f77 -64 -O3 -mips4 -LIST:=ON foo.f
```

This creates a *.L* file which contains a summary of the flags used by the compiler (including default values) and the software pipelining annotations.

Example 1: Output from Using the -S Compiler Switch

```
%cat test.f
        program test
        real*8 a x(100000),y(100000)
        do i = 1, 2000
           call daxpy(3.7, x, y, 100000)
        enddo
        stop
        end


        subroutine daxpy(a, x, y, nn)
        real*8 a x(*),y(*)
        do i = 1, nn, 1
           y(i) = y(i) + a * x(i)
        enddo
        return
        end
%f77 -64 -mips4 -O3 -S test.f
%grep swps test.s
#<swps>
#<swps> Pipelined loop line 11 steady state
#<swps>
#<swps>     4 unrollings before pipelining
#<swps>     6 cycles per 4 iterations
#<swps>     8 flops       ( 33% of peak) (madds count as 2
```

```
#<swps>     4 flops       ( 33% of peak) (madds count as 1
#<swps>     4 madds       ( 33% of peak)
#<swps>     12 mem refs   ( 100% of peak)
#<swps>     2 integer ops ( 16% of peak)

#<swps>     18 instructions ( 75% of peak)
#<swps>     1 short trip threshold
#<swps>     7 ireg registers used
#<swps>     11 fgr registers used
#<swps>
```

This shows that the inner loop starting at line 11 was software pipelined. The loop was unrolled four times before pipelining. It used 6 cycles for every four loop iterations and calculated the statistics as follows:

- If each *madd* counts as two floating point operations, the R8000 can do four floating point operations per cycle (two *madds*), so its peak for this loop is 24. Eight floating point references are 8/24 or 33% of peak. The figure for *madds* is likewise calculated.
- If each *madd* counts as one floating point operation, the R8000 can do two floating point operations per cycle, so its peak for this loop is 12. Four floating point operations are 4/12 or 33% of peak.
- The R8000 can do two memory operations per cycle, so its peak for this loop is 12. Three memory references are 12/12 or 100% of peak.
- The R8000 can do two integer operations per cycle, so its peak for this loop is 12. Two integer operations are 2/12 or 16% of peak.
- The R8000 can do four instructions per cycle, so its peak for this loop is 24. Eighteen instructions are 18/24 or 75% of peak. The statistics also point out that loops of less than 1 iterations would not go through the software pipeline replication area, but would be executed in the **simple_loop** section shown above and that a total of seven integer and eleven floating point registers were used in generating the code.

How to Diagnose What Went Wrong

When you don't get much improvement in your application's performance after compiling with software pipelining, you should ask the following questions and consider some possible answers:

1. Did it software pipeline at all?

   Software pipelining works only on inner loops. What is more, inner loops with subroutine calls or complicated conditional branches do not software pipeline.

2. How well did it pipeline?

   Look at statistics in the *.s* or *.L* file.

3. What can go wrong in *SWP* code?

   Your generated code may not have the operations you expected.

4. Think about how you would hand code it.

   What operations did it need?

   Look at the loop in the *.s* file.

   Is it very different? Why?

   Sometimes this is human error. (Improper code, or typo.)

   Sometimes this is a compiler error.

Perhaps the compiler didn't schedule tightly. This can happen because there are unhandled recurrence divides (Divides in general, are a problem) and because there are register allocation problems (running out of registers).

KAP SWP Conflicts

KAP may fuse loops causing the resulting loop not to software pipeline. The resulting loop may be too long (requiring too many registers), or it may contain a subroutine call from one of the loops. If you hand code for the software pipeliner, you may need to turn off KAP. You can do this by adding:

```
-WK,-o=0,-ro=0,-so=0
```

to your compilation switches.

Matrix Multiply – A Tuning Case Study

Matrix multiplication illustrates some of the issues in compiling for the R8000. Consider the simple implementation.

```
do j = 1,n
   do i = 1,m
       do k = 1 , p
          c(i,j) = c(i,j) – a(i,k)*b(k,j)
```

The R8000 is capable of issuing two *madds* and two load/stores per cycle. This simple version of matrix multiply requires 2 loads in the inner loop and one *madd*. Thus at best, it can run at half of peak speed. Note, though, that the same locations are loaded on multiple iterations. By unrolling the loops, we can eliminate some of these redundant loads. Consider for example, unrolling the outer loop by 2.

```
do j = 1,n,2
   do i = 1,m
      do k = 1 , p
         c(i,j) = c(i,j) – a(i,k)*b(k,j)
         c(i,j+1) = c(i,j+1) – a(i,k)*b(k,j+1)
```

We now have 3 loads for two *madds*. Further unrolling can bring the ratio down further.

Choosing the best unrolling factor requires trade-offs. First, we'd like to get the ratios of loads to *madds* below 1. Having an equal number of loads and *madds* may not be sufficient. Recall that memory bank conflicts can occasionally increase the time required to do a load. Thus, it's desirable to eliminate as many redundant loads as possible. On the other hand, heavily unrolled loops require many registers. Unrolling too much can lead to register spills.

Below is a good compromise for the matrix multiply example. It gets very close to peak performance.

```
do j = 1,n–4,5
  do i = 1,m–3,4
    do k = 1 , p
      c(i,j) = c(i,j) – a(i,k)*b(k,j)
      c(i,j+1) = c(i,j+1) – a(i,k)*b(k,j+1)
      c(i,j+2) = c(i,j+2) – a(i,k)*b(k,j+2)
      c(i,j+3) = c(i,j+3) – a(i,k)*b(k,j+3)
      c(i,j+4) = c(i,j+4) – a(i,k)*b(k,j+4)
```

```
        c(i+1,j) = c(i+1,j) - a(i+1,k)*b(k,j)
        c(i+1,j+1) = c(i+1,j+1) - a(i+1,k)*b(k,j+1)
        c(i+1,j+2) = c(i+1,j+2) - a(i+1,k)*b(k,j+2)
        c(i+1,j+3) = c(i+1,j+3) - a(i+1,k)*b(k,j+3)
        c(i+1,j+4) = c(i+1,j+4) - a(i+1,k)*b(k,j+4)

        c(i+2,j) = c(i+2,j) - a(i+2,k)*b(k,j)
        c(i+2,j+1) = c(i+2,j+1) - a(i+2,k)*b(k,j+1)
        c(i+2,j+2) = c(i+2,j+2) - a(i+2,k)*b(k,j+2)
        c(i+2,j+3) = c(i+2,j+3) - a(i+2,k)*b(k,j+3)
        c(i+2,j+4) = c(i+2,j+4) - a(i+2,k)*b(k,j+4)

        c(i+3,j) = c(i+3,j) - a(i+3,k)*b(k,j)
        c(i+3,j+1) = c(i+3,j+1) - a(i+3,k)*b(k,j+1)
        c(i+3,j+2) = c(i+3,j+2) - a(i+3,k)*b(k,j+2)
        c(i+3,j+3) = c(i+3,j+3) - a(i+3,k)*b(k,j+3)
        c(i+3,j+4) = c(i+3,j+4) - a(i+3,k)*b(k,j+4)
      enddo
    enddo
  do i =i,m
    do k = 1,p
      c(i,j) = c(i,j) - a(i,k)*b(k,j)
    enddo
  enddo
enddo
do j = j,n
  do i = 1,m-3,4
    do k = 1 , p
      c(i,j) = c(i,j) - a(i,k)*b(k,j)
      c(i+1,j) = c(i+1,j) - a(i+1,k)*b(k,j)
      c(i+2,j) = c(i+2,j) - a(i+2,k)*b(k,j)
      c(i+3,j) = c(i+3,j) - a(i+3,k)*b(k,j)
    enddo
  enddo
  do i =i,m
    do k = 1,p
      c(i,j) = c(i,j) - a(i,k)*b(k,j)
    enddo
  enddo
enddo
return
end
```

Note that we have not unrolled the inner loop. The software pipelining phase of the compiler can effectively unroll inner loops automatically to eliminate redundant loads.

In addition, the KAP front end is able to automatically unroll outer loops. Given the initial matrix multiply, KAP generates the following code, which is almost as efficient as our hand coded version.

```
do 5 J=1,N-3,4
    do 3 I=1,M-3,4
      DD1 = C(I,J)
      DD2 = C(I,J+1)
      DD3 = C(I,J+2)
      DD4 = C(I,J+3)
      DD5 = C(I+1,J)
      DD6 = C(I+1,J+1)
      DD7 = C(I+1,J+2)
      DD8 = C(I+1,J+3)
      DD9 = C(I+2,J)
      DD10 = C(I+2,J+1)
      DD11 = C(I+2,J+2)
      DD12 = C(I+2,J+3)
      DD13 = C(I+3,J)
      DD14 = C(I+3,J+1)
      DD15 = C(I+3,J+2)
      DD16 = C(I+3,J+3)
      do 2 K=1,P
         DD1 = DD1 - A(I,K) * B(K,J)
         DD2 = DD2 - A(I,K) * B(K,J+1)
         DD3 = DD3 - A(I,K) * B(K,J+2)
         DD4 = DD4 - A(I,K) * B(K,J+3)
         DD5 = DD5 - A(I+1,K) * B(K,J)
         DD6 = DD6 - A(I+1,K) * B(K,J+1)
         DD7 = DD7 - A(I+1,K) * B(K,J+2)
         DD8 = DD8 - A(I+1,K) * B(K,J+3)
         DD9 = DD9 - A(I+2,K) * B(K,J)
         DD10 = DD10 - A(I+2,K) * B(K,J+1)
         DD11 = DD11 - A(I+2,K) * B(K,J+2)
         DD12 = DD12 - A(I+2,K) * B(K,J+3)
            DD13 = DD13 - A(I+3,K) * B(K,J)
            DD14 = DD14 - A(I+3,K) * B(K,J+1)
            DD15 = DD15 - A(I+3,K) * B(K,J+2)
            DD16 = DD16 - A(I+3,K) * B(K,J+3)
```

```
 2            continue
              C(I,J) = DD1
              C(I,J+1) = DD2
              C(I,J+2) = DD3
              C(I,J+3) = DD4
              C(I+1,J) = DD5
              C(I+1,J+1) = DD6
              C(I+1,J+2) = DD7
              C(I+1,J+3) = DD8
              C(I+2,J) = DD9
              C(I+2,J+1) = DD10
              C(I+2,J+2) = DD11
              C(I+2,J+3) = DD12
              C(I+3,J) = DD13
              C(I+3,J+1) = DD14
              C(I+3,J+2) = DD15
              C(I+3,J+3) = DD16
3           continue
            do 5 I=I,M,1
              DD17 = C(I,J)
              DD18 = C(I,J+1)
              DD19 = C(I,J+2)
              DD20 = C(I,J+3)
              do 4 K=1,P
                DD17 = DD17 - A(I,K) * B(K,J)
                DD18 = DD18 - A(I,K) * B(K,J+1)
                DD19 = DD19 - A(I,K) * B(K,J+2)
                DD20 = DD20 - A(I,K) * B(K,J+3)
4             continue
              C(I,J) = DD17
              C(I,J+1) = DD18
              C(I,J+2) = DD19
              C(I,J+3) = DD20
5           continue
            do 9 J=J,N,1
              do 7 I=1,M-3,4
                DD21 = C(I,J)
                DD22 = C(I+1,J)
                DD23 = C(I+2,J)
                DD24 = C(I+3,J)
                do 6 K=1,P
             DD21 = DD21 - A(I,K) * B(K,J)
             DD22 = DD22 - A(I+1,K) * B(K,J)
             DD23 = DD23 - A(I+2,K) * B(K,J)
             DD24 = DD24 - A(I+3,K) * B(K,J)
```

```
6         continue
          C(I,J) = DD21
          C(I+1,J) = DD22
          C(I+2,J) = DD23
          C(I+3,J) = DD24
7       continue
        do 9 I=I,M,1
          DD25 = C(I,J)
          do 8 K=1,P
            DD25 = DD25 - A(I,K) * B(K,J)
8         continue
          C(I,J) = DD25
9       continue
        return
      end
```

The Effect of Bellows Stalls

The R8000 microprocessor cache organization consists of a 16 kilobyte integer- only primary cache housed on the Integer Unit, and a 4 MByte second level streaming cache. The 4 MByte streaming cache acts as the second level cache for the IU and the first level cache for the FPU.

The R8000 microprocessor contains two Tag RAM's which support the two- way interleaved streaming cache and can perform two Tag RAM accesses per cycle. One bank contains even addresses (the cache is addressed in 64- bit words) and the other bank contains odd addresses.The R8000 microprocessor allows either two loads or one load and one store in the same cycle. In order to facilitate two accesses per cycle, one address must be even and the other odd. When there are two accesses that are both odd or both even, one of the accesses is delayed by one cycle in the address bellow register. Multiple conflicts can degrade performance. If all the accesses are even or all the accesses are odd, performance bottoms out at one load per cycle. The compiler tries but cannot always guarantee instruction schedules where one access is to an even address and the other to an odd address. So, even though the examples below may be actually scheduled correctly by the compiler to eliminate bellows stalls, they illustrate the basic principles that you can follow when the compiler can't eliminate bellows stalls.

Example 2: Single Precision Loops

SAXPY (Single precision A times X Plus Y)

The simple SAXPY code below can run at close to half the performance of theoretical maximum, depending on how the arrays *x* and *y* are aligned relative to each other and the order in which the arrays are accessed.

```
subroutine saxpy( a, x, y, nn)
real a, x(*), y(*)
do i = 1, nn
   y(i) =  y(i) + a*x(i)
end do

return
end
```

The alignment was controlled by putting the arrays in a common block in the calling main program (i.e.)

```
common x,y
```

The difficulty with this single-precision example is that the cache is interleaved at double word boundaries. As opposed to double precision accesses, consecutive single precision accesses do not go to alternate banks. Thus *y(i)* and *y(i+1)* are in one bank, while *y(i+2)* and *y(i+3)* are in the other bank.

A solution to this SAXPY problem is to unroll the single precision loop 4 times. It is possible (in theory) to schedule accesses such that they are grouped:

y(i), y(i+2) in one cycle

y(i+1), y(i+3) in a different cycle

This trick can be applied to any number of arrays and the normal bellows effect should take care of unknown base alignment of the arrays *y(i)*, *x(i)*:

y(i+2), x(i+2)

y(i+1), x(i+1)

...

Whether or not the alignment is good, interleaving gets you full performance.

The following code gets close to the theoretical maximum.

```
program sample
real a x(100000),y(100000)
   do i = 1,2000
      call saxpy(3.7,x,y,100000)
   enddo
   stop
   end
      subroutine saxpy( a, x, y, nn)
      real a, x(*), y(*)
      do i = 1, nn,4
            y(i) =  y(i) + a*x(i)
            y(i+2) =  y(i+2) + a*x(i+2)
            y(i+1) =  y(i+1) + a*x(i+1)
            y(i+3) =  y(i+3) + a*x(i+3)
   end do

   return
   end
```

Example 3: Double Precision Loops

Fortran Arrays are stored in column-major order. That means *y(1,1), y(2,1),y(3,1), y(4,1)* etc. are stored in contiguous memory locations. C on the other hand stores its arrays in row-major order; (*y[1,1], y[1,2], y[1,3]* etc. are stored contiguously.).

Fortran double precision loops can encounter bellows stalls if the column dimension is even. The simple loop below illustrates this:

```
        real*8 y(8,4)

        do 20 i = 1,8
           do 10 j = 1,4
              y(i,j) = y(i,j) + 1

10       continue
20    continue
```

This is because the array is laid out in the secondary cache as shown in Table 6-1:

Table 6-1 Fortran Array Layout in Secondary Cache

Bank 1	Bank 2
y(1,1)	y(2,1)
y(3,1)	y(4,1)
y(5,1)	y(6,1)
y(7,1)	y(8,1)
y(1,2)	y(2,2)
etc.	etc.

Note that *y(1,1)* and *y(1,2)* are in the same bank. Actually, all of the inner loop members are in the same bank. As a result there are bellows stalls.

Much better performance could have been obtained if the first dimension(column) was odd even if it didn't need the extra elements. This is because the secondary (G) cache would get laid out like this:

```
real*8 y(9,4)
```

The inner loop elements are in alternating banks now and the bellows stalls are avoided as shown in Table 6-2.

Table 6-2 Fortran Arrays in Alternating Banks

Bank 1	Bank 2
y(1,1)	y(2,1)
y(3,1)	y(4,1)
y(5,1)	y(6,1)
y(7,1)	y(8,1)
y(9,1)	y(1,2)
y(2,2)	y(2,3)
etc.	etc.

The software pipelining statistics in the *.s* or *.L* file can also tell you if you are having problems with bellows stalls. The following example illustrates this:

```
%cat daxpy2.f
      subroutine daxpy(a, x, y, nn)
      real *8 a, x(2,nn), y(2,nn)

      DO i = 1,nn
         y(1,i) = y(1,i) + a * x(1,i)
      END DO

      return
      end
%f77 -64 -mips4 -O3 -S daxpy2.f
%grep swps daxpy2.s
#<swps>
#<swps> Pipelined loop line 4 steady state
#<swps>
#<swps>     4 unrollings before pipelining
#<swps>     6 cycles per 4 iterations
#<swps>     8 flops       ( 33% of peak) (madds count as 2
#<swps>     4 flops       ( 33% of peak) (madds count as 1
#<swps>     4 madds       ( 33% of peak)
#<swps>     12 mem refs   ( 100% of peak)
#<swps>     2 integer ops ( 16% of peak)

#<swps>     18 instructions ( 75% of peak)
#<swps>     1 short trip threshold
#<swps>     9 ireg registers used
#<swps>     11 fgr registers used
#<swps>
```

```
#<swps>     6 possible stall cycles
#<swps>     6 min possible stall cycles
#<swps>
```

This shows that the inner loop starting at line 4 was software pipelined. In addition to the standard software pipelining statistics, the last three lines of the report show that at best, the compiler can schedule code which may have 6 bellows stall cycles for every four loop iterations and that the compiler did produce this schedule.

Use of the IVDEP Directive

The *IVDEP* directive is something that started in Cray Fortran. It stands for "Ignore Vector Dependencies".

It is a Fortran *pragma* that tells the compiler to be less strict when it is deciding whether it can get parallelism between loop iterations. By default, the compilers do the safe thing: they try to prove to themselves that there is no possible conflict between two memory references. If they can prove this, then it is safe for one of them to pass the other.

In particular, you need to be able to perform the load from iteration *i+1* before the store from iteration *i* if you want to be able to overlap the calculation from two consecutive iterations.

Now suppose you have a loop like:

```
do i = 1, n
   a(l(i)) = a(l(i)) + ...
enddo
```

The compiler has no way to know that

```
&a(l(i)) != &a(l(i+1))
```

without knowing something about the vector *l*. For example, if every element of *l* is 5, then

```
&a(l(i)) == &a(l(i+1))
```

for all values of *i*.

But you sometimes know something the compiler doesn't. Perhaps in the example above, *l* is a permutation vector and all its elements are unique. You'd like a way to tell the compiler to be less conservative. The *IVDEP* directive is a way to accomplish this.

Placed above a loop, the statement:

```
cdir$ ivdep
```

tells the compiler to assume there are no dependencies in the code.

Note: *IVDEP* IS DANGEROUS! If your code really isn't free of vector dependences, you may be telling the compiler to perform an illegal transformation which causes your program to get wrong answers.

But, *IVDEP* is also powerful and you may very well find yourself in a position where you need to use it. But you have to be very careful when you do this.

By using an *IVDEP*, it was possible to get a 5 (static) cycle inner loop for the following routine. This compares with 36 cycles without this directive.It is also possible to get good results by adding:

```
cdir$ ivdep
```

right above the start of the inner loop, then using:

-O3 -mips4

to compile.

For example

```
        subroutine m(a,nn,b,ldb,ix,kb,m,n,p,alpha,id1,id2,
     +          lda,aa)
c
        implicit double precision(a-h,o-z)
        real*8 alpha,beta
        dimension a(*),b(ldb,n),ix(nn),aa(lda,*)
        integer p
c
        double precision t
```

```
c
        do  100 j=1,(n-1),2
          idc = ix(id1+j) + id2
          idc1 = ix(id1+j+1) + id2
          do 50 k=1,(p-3),4
            t00 = b(k,j)
            t01 = b(k,j+1)
            t10 = b(k+1,j)
            t11 = b(k+1,j+1)
            t20 = b(k+2,j)
            t21 = b(k+2,j+1)
            t30 = b(k+3,j)
            t31 = b(k+3,j+1)
cdir$ ivdep
            do 30 i=1,m
              a(idc+i) = a(idc+i) + aa(i,k)*t00 +
&             aa(i,k+1)*t10 + aa(i,k+2)*t20 + aa(i,k+3)*t30
              a(idc1+i) = a(idc1+i) +
&             aa(i,k)*t01 +
&             aa(i,k+1)*t11 + aa(i,k+2)*t21 + aa(i,k+3)*t31
 30         continue
 50       continue
          do 80 k=k,p
            t00 = b(k,j)
            t01 = b(k,j+1)
            do i =1,m
              a(idc+i) = a(idc+i) + aa(i,k)*t00
              a(idc1+i) = a(idc1+i) + aa(i,k)*t01
            enddo
 80       continue
 100    continue
        do j=j,n
             idc = ix(id1+j) + id2
             do k=1,p
               t00 = b(k,j)
               do i=1,m
                 a(idc+i) = a(idc+i) + aa(i,k)*t00
               enddo
             enddo
           enddo
         return
         end
```

The inner loop now does 8 *madds* and 8 memory references in 5 cycles. In the limit it should be able to hit (static) full performance, but there is the nasty issue of incrementing the induction variables and testing for the end of the loop.

You can make it get a little closer to full performance if you force the loop to be unrolled. Right now this isn't happening because the loop contains too many operations before optimization when the decision of how much to unroll is made. The following option:

-OPT:unroll_size=1000

increases this limit from its default (320) to 1000 and allows the loop to be unrolled 2x. This yields 16 *madds* and 16 memory accesses in 9 cycles.

Vector Intrinsic Functions

The MIPSpro 64-bit compilers support both single and double precision versions of the following vector instrinsic functions: **asin()**, **acos()**, **atan()**, **cos()**, **exp()**, **log()**, **sin()**, **tan()**, **sqrt()**. In C they are declared as follows:

```
/* single precision vector routines */
extern  __vacosf( float *x, float *y, int count, int stridex, int stridey );
extern  __vasinf( float *x, float *y, int count, int stridex, int stridey );
extern  __vatanf( float *x, float *y, int count, int stridex, int stridey );
extern  __vcosf(  float *x, float *y, int count, int stridex, int stridey );
extern  __vexpf(  float *x, float *y, int count, int stridex, int stridey );
extern  __vlogf(  float *x, float *y, int count, int stridex, int stridey );
extern  __vsinf(  float *x, float *y, int count, int stridex, int stridey );
extern  __vtanf(  float *x, float *y, int count, int stridex, int stridey );
/* double precision vector routines */
extern  __vacos( double *x, double *y, int count, int stridex, int stridey );
extern  __vasin( double *x, double *y, int count, int stridex, int stridey );
extern  __vatan( double *x, double *y, int count, int stridex, int stridey );
extern  __vcos(  double *x, double *y, int count, int stridex, int stridey );
extern  __vexp(  double *x, double *y, int count, int stridex, int stridey );
extern  __vlog(  double *x, double *y, int count, int stridex, int stridey );
extern  __vsin(  double *x, double *y, int count, int stridex, int stridey );
extern  __vtan(  double *x, double *y, int count, int stridex, int stridey )
```

The variables x and y are assumed to be pointers to non-overlapping arrays. Each routine is functionally equivalent to the following pseudo-code fragment:

```
do i = 1, count-1
    y[0 + i*stridey] = func(x[0 + i*stridex])
enddo
```

where *func()* is the scalar version of the vector intrinsic.

Manual vs. Automatic Invocation

All of the vector intrinsics can be called explicitly from your program. In Fortran the following example invokes the vector intrinsic **vexpf()**.

```
real*4 x(N), y(N)
integer*8 N, i
    i = 1
    call vexpf$( %val(%loc(x(1))),
    &            %val(%loc(y(1))),
    &            %val(N),%val(i),%val(i))
    end
```

The compiler at optimization level **-O3** also recognizes the use of scalar versions of these intrinsics on array elements inside loops and turns them into calls to the vector versions. If you need to turn this feature off to improve the accuracy of your results, add **-OPT:vector_intrinsics=OFF** to your compilation command line or switch to a lower optimization level.

The compiler, however, is limited in its ability to recognize vector intrinsics. It does not generate calls to vector intrinsics in expressions that necessitate the generation of a temporary variable.

Performance and Accuracy

The vector intrinsics are optimized and software pipelined to take advantage of the R8000's performance features. Throughput is several times greater than that of repeatedly calling the corresponding scalar function though the result may not necessarily agree to the last bit. Also, the results from **vsin()**, **vcos()** and **vtan()** are valid only for argument values between 2^{28} and -2^{28}. Similarly, **Vlog**(0.0) and **vlog**(+*inf*) return *NaN*.