Compilation Issues

This chapter outlines the issues dealt with at compile and link times. It covers environment variable usage, compile-time switches, error messages and how to link correctly.

Environment Variables

The *SGI_ABI* environment variable can be used to specify whether you want a 32-bit or 64-bit compilation environment. It is overridden by use of the *-32* or *-64* option on the command line. The usage of other compiler environment variables is the same between the 32-bit and 64-bit compiler systems. Special care should be taken in using the following two environment variables.

_RLD_PATH	Used to override the default runtime 32-bit linker
_RLD64_PATH	Used to override the default runtime 64-bit linker

Using the wrong *_RLD*_PATH* can cause 64-bit code to try to use 32-bit *.so's*. For the most part, staying with the defaults is the best policy.

Command Line Switches

This section lists the differences between the supported switches of the 32-bit and 64-bit compilers.

If you use the -**32** switch on the compiler command line, the 32-bit (*ucode)* compiler is invoked. If you use -**64**, the 64-bit compiler is run.

The default switches on Power Challenge Systems are -**64 -mips4**.

Although they are two separate and different compiler systems, the 32-bit and 64-bit compilers have similar command line interfaces. The following list summarizes the differences between the compilers in the switches that they support:

Fortran Switch Differences

The 32-bit compiler supports the following switches which the 64-bit compiler does not:

-**32**	By definition.
-**mips1**	Generate code using the MIPS1 instruction set (the default for the 32-bit compiler).
-**mips2**	Generate code using the MIPS2 instruction set.
-**66**	Suppress extensions that enhance FORTRAN 66 compatibility.
-**w66**	Suppress only FORTRAN 66 compatibility warnings messages.
-**usefpidx**	Force the compiler to use the floating point DO loop variable as the loop counter.

The 64-bit compiler supports the following switches which the 32-bit compiler does not:

-**64**	By definition.
-**n32**	Generate code for the high performance 32-bit (N32) ABI.
-**mips3**	Generate code using the MIPS3 instruction set.
-**mips4**	Generate code using the MIPS4 instruction set (the default for the 64-bit compiler on Power Challenge).

C Switch Differences

The 32-bit compiler supports the following switches which the 64-bit compiler does not:

-32,-mipsl,-mips2
As in 32-bit Fortran above.

-kpicopt Asks *uopt* to perform the special treatment for global variables to optimize their program accesses regardless of the shared/non-shared compilation mode.

-hpath Use *path* rather than the directory where the name is normally found.

-Bstring Append *string* to all names specified by the **-t** option.

The 64-bit compiler supports the following switches which the 32-bit compiler does not:

-64, **-n32**,**-mips3 -mips4**
As in 64-bit Fortran above.

-A Add following *cpp* assertions.

-H Run *cpp* and print list of headers.

-help Print list of possible options.

-keep Keep intermediate files.

-pcakeep Run Power C Analyzer and keep listing and *pca* output.

-pcalist Run Power C Analyzer and keep listing.

-show1 Show phases but don't have phases print additional info.

-showt Show time taken by each phase.

-unsigned Characters are unsigned by default.

-woffall Turn off all warnings.

The following switches are accepted by both compilers but have different semantics:

-**v** On the 32-bit compiler, it's the same as -**show**, whereas it means verbose warnings on the 64-bit compiler.

-**woff** Turn off named warnings, but the warning numbers are different between the compilers.

-**Wc**,**arg1**,[,**arg2**...]
Designate to which pass, **c**, of the compiler the arguments are to be passed.

Optimization Switches of the 64-Bit Compilers

In addition to the switches listed above, both the 64-bit Fortran and 64-bit C compilers support many more switches. These switches are used to control the types of optimizations the compilers perform. This section outlines the various optimizations that the 64-bit compilers can perform, and lists the switches used to control them. To get a summary of the flags and their values 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 all the flags used in the compilation including the default values of flags not explicitly set on the command line.

The optimizations and switches fall into the following categories:

- General Optimizations
- Floating Point Optimizations
- Miscellaneous Optimizations
- Global Code Motion
- Software Pipelining
- Target Environment Issues

General Optimizations

-O0

Perform no optimization that may complicate debugging (this is the default). User data is kept in memory, and little or no attempt is made to eliminate redundant code.

-O1

Perform local optimizations which do not significantly affect compile-time performance. Most user data is still kept in memory, but much of the locally redundant code is removed.

-O2
-O

Perform extensive global optimization. The optimizations at this level are generally conservative in the sense that they:

- Provide code improvements commensurate with the compile time spent
- Are almost always beneficial
- Avoid changes which affect such things as floating point results.

-O3

Perform aggressive optimization. The additional optimization at this level is distinguished by a focus on maximizing code quality even if that requires extensive compile-time or relaxing language rules. It is more likely to use transformations that are usually beneficial but can hurt performance in isolated cases. This level may cause noticeable changes in floating point results due to relaxing expression evaluation rules. See the discussion of the *-OPT:roundoff* option below.

Floating Point Optimizations

Floating point numbers (Fortran's **REAL*n**, **DOUBLE PRECISION**, and **COMPLEX*n**, and C's **float**, **double**, and **long double**) are inexact representations of ideal real numbers. The operations performed on them are also necessarily inexact. However, the MIPS processors conform to the IEEE 754 floating point standard, producing results as precise as possible given the constraints of the IEEE 754 representations. The MIPSpro 64-bit compilers generally preserve this conformance. (Note, however, that 128-bit floating point, such as Fortran's **REAL*16** and C's **long double**, is not precisely IEEE-compliant.)

In addition, the source language standards imply rules about how expressions are to be evaluated. However, most code that has not been written with careful attention to floating point behavior does not require precise conformance to either the source language expression evaluation standards or to IEEE 754 arithmetic standards. Therefore, the MIPSpro 64-bit compilers provide a number of options which trade off source language expression evaluation rules and IEEE 754 conformance against better performance of generated code. These options allow transformations of calculations specified by the source code that may not produce precisely the same floating point result, although they involve a mathematically equivalent calculation.

Two of these options are the preferred controls. -**OPT:roundoff** deals with the extent to which language expression evaluation rules are observed, generally affecting the transformation of expressions involving multiple operations. -**OPT:IEEE_arithmetic** deals with the extent to which the generated code conforms to IEEE 754 standards for discrete IEEE-specified operations (for example, a divide or a square root). The remaining options in this class may be used to obtain finer control, but they may disappear or change in future compiler releases.

The first general option provides control over floating point accuracy and overflow/underflow exception behavior relative to the source language rules:

-OPT:roundoff=n

The roundoff option specifies the extent to which optimizations are allowed to affect floating point results, in terms of both accuracy and overflow/underflow behavior. The roundoff value, **n**, has a value in the range 0..3 with the following meaning:

- **roundoff=0**

  Do no transformations which could affect floating point results. This is the default for optimization levels -**O0** to -**O2**.

- **roundoff=1**

  Allow transformations with limited effects on floating point results. For roundoff, limited means that only the last bit or two of the mantissa are affected. For overflow (underflow), it means that intermediate results of the transformed calculation may overflow within a factor of two of where the original expression might have overflowed (underflowed). Note that limited effects may be less limited when compounded by multiple transformations.

- roundoff=2

  Allow transformations with more extensive effects on floating point results. Allow associative rearrangement, even across loop iterations, and distribution of multiplication over addition/subtraction. Disallow only transformations known to cause cumulative roundoff errors or overflow/underflow for operands in a large range of valid floating point values.

  Re-association can have a substantial effect on the performanceof software pipelined loops by breaking recurrences. This is therefore the default for optimization level -**O3**.

- roundoff=3

  Allow any mathematically valid transformation of floating point expressions. This allows floating point induction variables in loops, even when they are known to cause cumulative roundoff errors, and fast algorithms for complex absolute value and divide, which overflow (underflow) for operands beyond the square root of the representable extremes.

The second general option controls conformance to IEEE 754 arithmetic standards for discrete operators:

-OPT:IEEE_arithmetic=n

The roundoff option specifies the extent to which optimizations must preserve IEEE floating point arithmetic. The value *n* must be in the range 1..3, with the following meaning:

- **IEEE_arithmetic=1**

  No degradation: do no transformations which degrade floating point accuracy from IEEE requirements. The generated code may use instructions like **madds** which provide greater accuracy than required by IEEE 754. This is the default.

- **IEEE_arithmetic=2**

  Minor degradation: allow transformations with limited effects on floating point results, as long as exact results remain exact. This option allows use of the MIPS 4 **recip** and **rsqrt** operations.

- **IEEE_arithmetic=3**

  Conformance not required: allow any mathematically valid transformations. For instance, this allows implementation of *x/y* as **x*recip(y).** As an example, consider optimizing the Fortran code fragment:

  ```
  INTEGER i, n
  REAL sum, divisor, a(n)
  sum = 0.0
  DO i = 1,n
      sum = sum + a(i)/divisor
  END DO
  ```

  At **roundoff=0** and **IEEE_arithmetic=1**, the generated code must do the *n* loop iterations in order, with a **divide** and an **add** in each.

  Using **IEEE_arithmetic=3**, the **divide** can be treated like *a(i)*(1.0/divisor)*. On the MIPS R8000, the reciprocal can be done with a **recip** instruction. But more importantly, the reciprocal can be calculated once before the loop is entered, reducing the loop body to a much faster multiply and add per iteration, which can be a single *madd* instruction on the R8000.

Using **roundoff=2**, the loop may be reordered. The original loop takes at least 4 cycles per iteration on the R8000 (the latency of the *add* or *madd* instruction). Reordering allows the calculation of several partial sums in parallel, adding them together after loop exit. With software pipelining, a throughput of nearly 2 iterations per cycle is possible on the R8000, a factor of 8 improvement.

Consider another example:

```
INTEGER i,n
COMPLEX c(n)
REAL r
DO i = 1,n
   r = 0.1 * i
   c(i) = CABS ( CMPLX(r,r) )
END DO
```

Mathematically, **r** can be calculated by initializing it to 0.0 before entering the loop and adding 0.1 on each iteration. But doing so causes significant cumulative errors because the representation of 0.1 is not exact. The complex absolute value mathematically is equal to
SQRT(r*r + r*r). However, calculating it this way causes an overflow if **2*r*r** is greater than the maximum REAL value, even though a representable result can be calculated for a much wider range of values of *r* (at greater cost). Both of these transformations are forbidden for **roundoff=2**, but enabled for **roundoff=3**.

There are several other options which allow finer control of floating point behavior than is provided by **-OPT:roundoff**.

-OPT:div_split[=(ON|OFF)]

Enable/disable the calculation of **x/y** as **x*(1.0/y)**, normally enabled by **IEEE_arithmetic=2**. See **-OPT:recip**.

-OPT:fast_complex[=(ON|OFF)]

Enable/disable the fast algorithms for complex absolute value and division, normally enabled by **roundoff=3**.

-OPT:fast_exp[=(ON|OFF)]

Enable/disable the translation of exponentiation by integers or halves to sequences of multiplies and square roots. This can change **roundoff** and can make these functions produce minor discontinuities at the exponents where it applies. Normally enabled by **roundoff>0** for Fortran, or for C if the function **exp()** is labelled intrinsic in **<***math.h***>** (the default in -**xansi** and -**cckr** modes).

-OPT:fast_sqrt[=(ON|OFF)]

Enable/disable the calculation of square root as **x*rsqrt(x)** for MIPS4 and above. Note that this optimization produces a *NaN* instead of 0.0 for a 0 operand.

-OPT:fold_reassociate[=(ON|OFF)]

Enable/disable transformations which reassociate or distribute floating point expressions, normally enabled by **roundoff>1**.

-OPT:IEEE_comparisons[=ON]

Force comparisons to yield results conforming to the IEEE 754 standard for *NaN* and *Inf* operands, normally disabled. Setting this option disables certain optimizations like assuming that a comparison *x==x* is always TRUE (since it is FALSE if *x* is a *NaN*).

It also disables optimizations which reverse the sense of a comparison, for example, turning $x < y$ into **! (x >= y)**, since both **x<y** and **x>=y** may be FALSE if one of the operands is a *NaN*.

-OPT:recip[=(ON|OFF)]

Allow use of the MIPS4 reciprocal instruction for 1.0/*y*, normally enabled by **IEEE_arithmetic>/=1**. See **-OPT:div_split**.

-OPT:rsqrt[=(ON|OFF)]

Allow use of the MIPS4 reciprocal square root instruction for 1.0/**sqrt(y)**, normally enabled by **IEEE_arithmetic>/=1**.

-TARG:madd[=(ON|OFF)]

The MIPS 4 architecture supports fused multiply-add instructions, which add the product of two operands to a third, with a single roundoff step at the end. Because the product is not rounded separately, this can produce slightly different (but more accurate) results than a separate multiply and add pair of instructions. This is normally enabled for **-mips4**.

Miscellaneous Optimizations

The **-OPT** group allows control over a variety of other optimization choices.

-OPT:space

The MIPSpro 64-bit compilers normally make optimization decisions based strictly on the expected execution time effects. If code size is more important, use this option. One of its effects is to cause most subprogram exits to go though a single exit path, with a single copy of register restores, result loads, and so on.

-OPT:alias=name

The compilers must normally be very conservative in optimization of memory references involving pointers (especially in C), since aliases (different ways of accessing the same memory) may be very hard to detect. This option may be used to specify that the program being compiled avoids aliasing in various ways. The choices are:

- alias=any

  The compiler assumes that any pair of memory references may be aliased unless it can prove otherwise (the default).

- *alias=typed*

  The compiler assumes that any pair of memory references which reference distinct types in fact reference distinct data.

For example, consider the code:

```
void dbl ( int *i, float *f )
{
     *i = *i + *i;
     *f = *f + *f;
}
```

The compiler assumes that *i* and *f* point to different memory, and produces an overlapped schedule for the two calculations.

- **alias=unnamed**

  The compiler assumes that pointers never point to named objects. For example, consider the code:

```
floatg;
void dbl ( float *f )
{
    *g = g + g;
    *f = *f + *f;
}
```

  The compiler assumes that *f* cannot point to *g*, and produces an overlapped schedule for the two calculations. This option also implies the *alias=typed* assumption. Note that this is the default assumption for the pointers implicit in Fortran dummy arguments according to the ANSI standard.

- **alias=restrict**

  The compiler assumes a very restrictive model of aliasing, where no two pointers ever point to the same memory area. For example, consider the code:

```
void dbl ( int *i, int *j )
{
    *i = *i + *i;
    *j = *j + *j;
}
```

  The compiler assumes that *i* and *j* point to different memory, and produces an overlapped schedule for the two calculations.

  Although this is a very dangerous option to use in general, it may produce significantly better code when used for specific well-controlled cases where it is known to be valid.

Loop Unrolling Optimizations

The following options control loop unrolling in the MIPSpro 64-bit optimizer, that is, making multiple copies of a loop body to minimize the loop overhead or to expose more instruction parallelism. Unrolling is subject to a number of limits in the optimizer, intended to balance the runtime benefits against code expansion. These options allow the user to modify those limits when they can be improved. Note that loops expected to be software pipelined are subject to similar options in the -**SWP** group.

-OPT:unroll_times_max=n

The optimizer normally unrolls loops at most 2 times (-**mips4**) or 4 times (-**mips3**), unless it can unroll them completely. This option modifies the default limit.

-OPT:unroll_size=n

The optimizer normally unrolls loops only to the extent that the resulting unrolled loop body contains at most 320 instructions. This option modifies the default limit.

-OPT:unroll_bblimit=n

The optimizer normally unrolls loops containing up to 10 basic blocks. This option modifies the default limit. (A basic block is a sequence of code between branches and labels, that is, a sequece of code with a single entry at the top and a single exit at the bottom.)

Global Code Motion

An important optimization performed by the MIPSpro compilers is called Global Code Motion (GCM). It is intended to improve the overall execution time of programs by redistributing the instructions among the basic blocks along an execution path to improve instruction parallelism and make better use of the machine resources.

Traditional global optimizers avoid moving instructions in cases which might cause them to be executed along control flow paths where they would not have been in the original program. However, GCM performs such motion, called “speculative code motion” because the instructions moved are executed based on speculation that they will actually prove useful. By default, GCM is very conservative in its speculations. Most of the options in the -GCM: group control the sorts of speculation which are to be allowed.

Valid speculative code motion must normally avoid moving operations which may cause runtime traps. As a result, turning off certain traps at runtime enables more motion. See the target environment option -**TENV:X=n** for general control over the exception environment.

-GCM:aggressive_speculation[=(ON|OFF)]

GCM normally tries not to move instructions to basic blocks which are already using most of the instruction execution resources available, since doing so likely extends the execution time of the block. This option minimizes that bias, which often helps floating point intensive code.

-GCM:array_speculation[=(ON|OFF)]

A form of speculation which is often very effective is called “bottom loading.” It involves moving instructions from the top of a loop body to both the block before the loop (for the first iteration) and to the end of the loop body (which executes them at the end of one iteration so that they are ready early for the next iteration). Doing this, however, means that the instructions are executed one or more extra times in the last iteration(s) of the loop. If the instructions moved are loading elements of an array, this may cause extra accesses beyond the end of the array. This option enables such accesses a small distance beyond the endof the array. The compiler attempts to pad arrays to guarantee that such accesses won't cause memory faults, but it may not always be able to do so, so this option must be used with care.

-GCM:pointer_speculation[=(ON|OFF)]

This option allows speculative motion of loads of two kinds, both involving pointer usage. The first is to allow motion of loads through pointers which may be NULL. In order to prevent this from causing memory faults, the compiler causes page zero of the address space to be mapped at runtime. Since this may prevent some truly invalid references from causing faults, this option should be avoided during debugging.

The second form involves moving a reference like **(p+n)*, for a small integer *n*, to a block which already contains a reference to **p*. The assumption involved is that if *p* is a valid address, *p+n* is too.

Consider an example:

```
...
if ( p->next != NULL ) {
    sum += p->next->val;
}
...
```

If this option is set, the load of *p->next->val* can be moved before the *if* (it is through a potentially NULL pointer).

-GCM:static_load_speculation[=(ON|OFF)]

Allow the speculative motion of loads from static data areas.Since such areas are known to be allocated, such motion cannot cause new faults except page faults or cache misses. As a result, it is generally safe, although it may hurt performance if it causes unnecessary page faults or cache misses.

-GCM:prepass[=(ON|OFF)]
-GCM:postpass[=(ON|OFF)]

GCM is implemented in two passes, one before instruction scheduling and register allocation, and one after, currently enabled at optimization level 2 (*-O2*) and above. These options allow either or both of the passes to be enabled or disabled. The two passes are complementary in effect, often improving code more in combination than the sum of their individual improvements, so enabling just one is not recommended.

Software Pipelining

Software pipelining (SWP) is an important optimization for the inner loops of programs, which can cause dramatic improvement by rearranging the loops to overlap calculations from multiple iterations. This is an iterative process, searching for an effective schedule and then for a workable allocation of registers and then retrying if either step fails. Some of the options in the *-SWP* group control that process. Others control how the loop body is prepared for the attempt, for example, by unrolling.

Many important loop preparation transformations involve reassociation of floating point values. See the discussion of floating point optimization above, especially the *-OPT:roundoff* option.

SWP normally must be careful during the initial and final iterations of a loop to not perform extra operations which might cause runtime traps. It must be similarly careful if early exits from a loop (that is, before the initially calculated trip count is reached) are possible. Turning off certain traps at runtime can give it more flexibility, producing better schedules and/or simpler wind-up/wind-down code. See the target environment option *-TENV:X=n* for general control over the exception environment.

-SWP:=(ON|OFF)

Enable/disable SWP (normally enabled at *-O3*).

-SWP:back_substitution[=(ON|OFF)]

The iteration interval (II) of a pipelined loop (that is, the frequency at which new iterations are started), is constrained by circular data dependencies across iterations, called recurrences. This option, **ON** by default, allows transformations which make recurrences less severe by substituting the expression which defines a variable for the variable. For example, consider the code:

```
DO i=1,n
   a(i) = a(i-1) + 5.0
END DO
```

Without back-substitution, each iteration must wait for the previous iteration's add to complete, yielding a best case of 4 cycles per iteration on the R8000. Back-substitution can transform the loop to something like:

```
DO i=1,n
    a[i] = a[i-8] + 40.0
END DO
```

With appropriate initialization, this version can achieve an effective rate of nearly two iterations per cycle.

-SWP:backtracks_max=n

SWP often backtracks and tries again when it fails to find a workable schedule. This option controls the limit on how many times it does so. Increasing the limit improves its chances of success; descreasing it may reduce the compilation time.

-SWP:body_ins_count_max=n

SWP is not attempted for loop bodies containing more than *n* instructions (default 100, 0 for no limit). Larger loop bodies are less likely to be successfully pipelined, and take more compilation time in the attempt, so this is another tradeoff of (potential) code improvement vs. compile time.

Loop bodies are also normally unrolled in preparation for SWP. This also limits the unrolling, since loops are not unrolled to more than *n* instructions in the unrolled body. Unrolling is also constrained by the **unroll_times_max** option described below. (Unrolling of loop bodies not expected to be software pipelined is controlled separately by -**OPT:unroll_size** and -**OPT:unroll_times_max**.)

-SWP:fix_recurrences[=(ON|OFF)]

This option controls both of the transformations controlled by **back_substitution** and **interleave_reductions**. See their descriptions.

-SWP:if_conversion[=(ON|OFF)]

SWP generally works much better on loop bodies without internal branches caused by conditional execution. This option causes conditional branches to be removed when possible by using conditional move instructions (MIPS4) and equivalents. For example, consider the code:

```
DO i=1,n
   IF ( a(i) .LT. b(i) ) THEN
      c(i) = a(i)
   ELSE
      c(i) = b(i)
   END IF
END DO
```

The loop body can be compiled for MIPS4 as:

```
ldc1      $f0,a(i)
ldc1      $f1,b(i)
c.lt.s    $fcc1,$f0,$f1
movf.s    $f0,$f1,$fcc1
sdc1      $f0,c(i)
```

Note that there are no conditional branches in the code. This option is **ON** by default for MIPS4 targets only.

-SWP:interleave_reductions[=(ON|OFF)]

This option, **ON** by default, has the same motivation as back-substution. It allows transformations which make recurrences arising from reductions less severe by interleaving multiple threads of the reduction and then piecing them together at the end of the loop. For example, consider the code to sum an array:

```
DO i=1,n

     sum = sum + a(x)
END DO
```

Without interleaving, each iteration must wait for the previous iteration's add to complete, yielding a best case II of 4 cycles per iteration on the R8000. Interleaving can transform the loop to something equivalent to:

```
DO i=1,n,8
    sum1 = sum1 + a(i)
    sum2 = sum2 + a(i+1)
    sum3 = sum3 + a(i+2)
    sum4 = sum4 + a(i+3)
    sum5 = sum5 + a(i+4)
    sum6 = sum6 + a(i+5)
    sum7 = sum7 + a(i+6)
    sum8 = sum8 + a(i+7)
END DO
sum = sum + sum1 + sum2 + sum3 + sum4 + sum5 + sum6 + sum7 + sum8
```

This version can achieve an effective II of nearly 0.5 cycles.

These transformations generally require -**OPT:roundoff=2** or better.

-SWP:trip_count_min=n

SWP is not attempted for loops with trip counts known to be smaller than *n* (default 5). The limit is applied via a runtime test for cases where the trip count is not known at compile time. Sometimes, a longer loop body can be profitably pipelined even with a smaller trip count, enabled by this option. *-SWP:unroll_times_max=n.*

This option controls the maximum number of times inner loop bodies are unrolled before attempting pipelining. The default is 4 for MIPS4 and 1 for MIPS3. Unrolling is also constrainedby the *body_ins_count_max* option described above. (Unrolling of loop bodies not expected to be software pipelined is controlled separately by -**OPT:unroll_size** and -**OPT:unroll_times_max**.)

Target Environment Issues

Generated code is affected by a number of assumptions about the target software environment. The options in this group tell the compiler what assumptions it can make, and sometimes what assumptions it should enforce.

The first such options are concerned with the shared code model.

-TENV:large_GOT[=(ON|OFF)]
-TENV:small_GOT[=(ON|OFF)]

Shared code and dynamic shared objects (DSOs) require the use of a global offset table (GOT) containing addresses of static data and subprograms at runtime. A dedicated register (*$gp*) points to the GOT at runtime, and the code can load these addresses from the GOT without being dependent on its actual virtual address. If the GOT is less than 64KB in size, those loads can all be single instructions; otherwise they require adding a constructed offset to *$gp*. These options choose one of those cases (default **small_GOT**).

As mentioned in the GCM and SWP discussions above, being able to execute instructions speculatively can make a significant difference in the quality of generated code. What instructions can be executed speculatively depends on the exception state at runtime:

-TENV:X=n

Specify the level (0 to 5, default 1) of enabled traps that is assumed (and enforced) for purposes of performing speculative code motion. At level 0, no speculation is done. At level 1, only safe speculative motion may be done, assuming that the IEEE 754 underflow and inexact traps are disabled. At level 2, all IEEE 754 floating point traps are disabled except divide by zero. At level 3, divide by zero traps are disabled. At level 4, memory traps may be disabled or dismissed by the operating system. At level 5, any exceptions may be disabled or ignored. Non-default levels should be used with great care. Disabling traps eliminates a useful debugging tool, since the problems which cause traps are detected later (often much later) in the execution of the program. In addition, many memory traps can't be avoided outright, but must be dismissed by the operating system after they occur. As a result, level 4 or 5 speculation can actually slow a program down significantly if it causes frequent traps.

Disabling traps in one module requires disabling them for the entire program. Programs which make use of level 2 or above should not attempt explicit manipulation of the hardware trap enable flags.

Data Alignment Options

The last set of environment options are concerned with the alignment of data:

-align8 / -align16 / -align32 / -align64

The MIPS architectures perform memory references much more efficiently if the data referenced is naturally aligned, that is, if 4-byte objects are at 4-byte-aligned address, etc. By default, the compilers allocate well-aligned data, and that is a requirement of the ABI for C. However, code ported from other architectures without alignment constraints may require less restricted alignment.The ANSI Fortran standard essentially requires maximum alignment of 4 bytes (32 bits), although it is unusual for code to actually depend on this. These options specify a maximum alignment (in bits) to be forced in allocating data. The MIPSpro 64-bit compilers default to -**align64** for MIPS3 or MIPS4 Fortran, and to ABI alignment (up to 128 bits for long double) for C.

-TENV:align_aggregates=n

The ABI specifies that aggregates (that is, **structs** and **arrays**) be aligned according to the strictest requirements of their components (that is, fields or elements). Thus, an array of short ints (or a struct containing only short ints or chars) is normally 2-byte aligned. However, some non-ANSI-conforming code may reference such objects assuming greater alignment, and some code (for example, **struct** assignments) may be more efficient if the objects are better aligned). This option specifies that any aggregate of size at least *n* is at least n-byte aligned. It does not affect the alignment of aggregates which are themselves components of larger aggregates.

-TENV:misalignment=n

This option determines the model (**n**) the compiler uses in deciding whether or not to assume that the memory references it generates are well-aligned. If *n*=1, it assumes well-aligned references unless it can determine otherwise (except in circumstances like struct copies where the ABI alignment requirements wouldn't normally imply alignment). If *n*=2, it always analyzes alignment based on the actual expression used, and trusts casts in the source code to reflect actual alignment. For instance, consider the expression:

```
*(type_a *) &b
```

The declared type of *b* implies a certain minimum alignment, as does the cast to *type_a*; the compiler uses the maximum of the two. If *n*=3, the compiler also analyzes alignment, but does not trust casts. In the above example, it assumes the alignment of the declared type of *b*.

The current default (for the beta release) is model 3. This is expected to change to model 1 or 2 for the full release.

Compilation Messages

Because they are two separate and different compiler systems, the 32-bit and 64-bit MIPSpro compilers have different warning and error messages.

The 32-bit C compiler emits errors as follows: the phase of the compiler which detects an error or warning, identifies itself first, then it prints whether it found an error or warning. The next printed items are the file name, line number and description of the error.

Fortran error messages are similar except the compiler phase name is omitted.

Two examples illustrate this:

C:

```
%cc -32bit test.c
cfe: Error: test.c, line 4: Syntax Error
```

Fortran:

```
% f77 -32bit err.f
Error on line 6 of err.f: syntax error
```

Under the 64-bit compiler, back end warning messages start with the string “!!!”. Error messages start with the string “###”. This allows easier searches in log files. C error messages are similar to the 32-bit compiler’s error messages, although they are usually more descriptive.

Here the same files compiled with the 64-bit compiler result in the following error messages:

C:

```
%cat test.c
#include <stdio.h>
main ()
{
    printf (“hhelp”)
}
%cc -64bit test.c
“test.c”, line 4: error(1065): expected a “;”
}
^
```

Fortran:

```
%cat err.f
 program test
c
c
2 write(6,8)
8 formatr('This is a test')
 end

% f77 -64bit err.f
### 8 formatr('This is a test')
### in line 6 procedure TEST of file err.f ###
### error Statement unrecognizable as any known statement type. ###
### error Label “8” is not the label of a format statement. ###
2 errors in file /tmp/ctmi.AAAa000Y
```

Important Warning Messages

When porting a 32-bit application to 64-bits use the -**fullwarn** option to expose variable size related issues. Two warnings in particular emitted by the 64-bit C compiler are helpful in this respect. They allow you to locate faulty assumptions that integers and pointers are the same size.

warning(1411): destination type of cast is too small to hold all pointers: truncation possible

For example, the following code fragment generates this warning when compiled -**64**.

```
unsigned i, *p;
i = (unsigned) p;
```

In this example, the unsigned integer variable *i* is set to only the low order 32 bits of *p*. If *p* contains an address greater than 32 bits, that address is truncated in *i*.

warning(1412): source type of cast is too small to hold all pointers: sign extension possible

The following code fragment generates this warning when compiled -**64**.

```
int i, *p;
p = (int *) i;
```

In this example, if *i* is negative, the sign bit is extended throughout the high-order 32 bits of *p*. The result is that *p* contains an invalid address.

Linking Your Application

The *cc* driver automatically picks the correct compiler, linker, and library paths if you use the -**32** or -**64** switches on the command line. If you compile and link separately, you must use these switches on both the *cc* driver and *ld* command lines. If you link with libraries that are not included with the O/S or compiler, you must make sure that they are of the proper type. Remember, you can't link 32-bit applications with 64-bit *.so's* or 64-bit libraries and vice versa.

To create your own 64-bit or 32-bit libraries, you must supply the correct switch on the archiver command line.

Libraries

The 32-bit and 64-bit compilers include some different libraries. Whereas the 32-bit compiler includes *libF77.a*, *libI77.a*, *libU77.a* and *libisam.a* under */usr/lib/nonshared*, the 64-bit compiler has one library, *libftn.a*, under */usr/lib64/nonshared*.

The 64-bit compiler also introduces routines which do QUAD precision floating point calculations into existing libraries.

The 64-bit compiler provides a united runtime library for parallel C and parallel Fortran programs (*libmp*), as opposed to two (*libcmp*, *libmpc*) libraries under the 32 code compiler. This united library allows you to mix parallel C with parallel fortran programs.

The 64-bit compiler also does not include *libmld.a*, but provides *libdwarf.a*.