Chapter 5

Optimizing Program Performance

This chapter describes the compiler optimization facilities and their benefits, and explains the major optimizing techniques. Topics covered include:

- "Optimization Overview"
- "Optimization Options"
- "Improving Global Optimization"
- "Improving Other Optimization"
- "Register Allocation"

For information about writing code for 64-bit programs, see Chapter 6, "Coding for 64-Bit Programs." For information about porting code to N32 and 64-bit systems, see Chapter 7, "Porting Code to N32 and 64-Bit Silicon Graphics Systems."

Optimization Overview

This section explains optimization benefits and optimization and debugging.

Benefits of Optimization

The primary benefits of optimization are faster running programs and often smaller object code size. However, the optimizer can also speed up development time. For example, you can reduce coding time by leaving it up to the optimizer to relate programming details to execution time efficiency. You can focus on the more crucial global structure of your program. Moreover, programs often yield optimizable code sequences regardless of how well you write your source program.

Optimization and Debugging

Optimize your programs only when they are fully developed and debugged. The optimizer may move operations around so that the object code does not correspond in an obvious way to the source code. These changed sequences of code can create confusion when using a debugger.

Using the Optimization Options

This section describes the optimization options and includes:

- "Optimization Options"
- "Controlling Floating Point Optimization"
- "Controlling Miscellaneous Optimizations"
- "Controlling Global Code Motion"
- "Controlling Software Pipelining"
- "Controlling the Target Architecture"
- "Controlling the Target Environment"

Optimization Options

Invoke the optimizer by specifying a compiler, such as *cc*(1), with any of the options listed in Table 5-1.

Table 5-1 Optimization Options

Option	Result
–O0	Performs no optimization that may complicate debugging. No attempt is made to minimize memory access by keeping values in registers, and little or no attempt is made to eliminate redundant code. This is the default.
–O1	Performs as many local optimizations as possible without affecting compile-time performance. No attempt is made to minimize memory access by keeping values in registers, but much of the locally redundant code is removed.

Table 5-1 (continued) Optimization Options

Option	Result
–O2, –O	Performs extensive global optimization. The optimizations at this level are generally conservative in the sense that they: (1) provide code improvements commensurate with the compile time spent (2) are almost always beneficial (3) avoid changes that affect such things as floating point results
–O3	Performs aggressive optimization. The additional optimization at this level focuses on maximizing code quality even if that requires extensive compile time or relaxing language rules. **–O3** 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 floating point optimization and the *–OPT:roundoff* option below).

Refer to the applicable compiler reference page for details on the options.

Controlling Floating Point Optimization

Floating point numbers (the Fortran REAL*n, DOUBLE PRECISION, and COMPLEX*n, and the C 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, and the MIPSpro compilers generally preserve this conformance. Note, however, that 128-bit floating point (that is, the Fortran REAL*16 and the C long double) is not precisely IEEE-compliant. In addition, the source language standards imply rules about how expressions are evaluated.

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 compilers provide a number of options that 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).

These options are described below.

–OPT:roundoff=n

The **–***OPT:roundoff* option provides control over floating point accuracy and overflow/underflow exception behavior relative to the source language rules.

The *roundoff* option specifies the extent to that 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. Roundoff values are described below.

roundoff=0 Do no transformations that 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 is 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 may have overflowed (underflowed). Note that 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.

Reassociation can have a substantial effect on the performance of 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.

–*OPT:IEEE_arithmetic=n*

The **–OPT:IEEE_arithmetic** option controls conformance to IEEE 754 arithmetic standards for discrete operators.

The **–OPT:IEEE_arithmetic** option specifies the extent to which optimizations must preserve IEEE floating point arithmetic. The value *n* must be in the range 1...3. Values are described below.

–OPT:IEEE_arithmetic=1
No degradation: do no transformations that degrade floating point accuracy from IEEE requirements. The generated code may use instructions like **madd**s, which provide greater accuracy than required by IEEE 754. This is the default.

–OPT: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 IV **recip** and **rsqrt** operations.

–OPT:IEEE_arithmetic=3
Conformance not required: allow any mathematically valid transformations. For instance, this allows implementation of `x/y` as `x*recip(y)`, or `sqrt(x)` as `x*rsqrt(x)`.

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. (Refer to the description of **–SWP:interleave_reductions** below for another example.)

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 0.1is not exact. The complex absolute value is mathematically 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**.

Other Options to Control Floating Point Behavior

Other options exist that allow finer control of floating point behavior than is provided by *–OPT:roundoff*. The options may be used to obtain finer control, but they may disappear or change in future compiler releases.

–OPT:div_split[=(**ON**|**OFF**)]

Enable/disable the calculation of `x/y as x*(1.0/y)`, normally enabled by **IEEE_arithmetic=3**. 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_*io*[=(**ON | OFF**)]
Enable/disable inlining of **printf()**, **fprintf()**, **sprintf()**, **scanf()**, **fscanf()**, **sscanf()**, and **printw()** for more specialized lower-level subroutines. This option applies only if the candidates for inlining are marked as intrinsic (–D__INLINE_INTRINSICS) in the respective header files (**<***stdio.h***>** and **<***curses.h***>**); otherwise they are not inlined. Programs that use functions such as **printf()** or **scanf()** heavily generally have improved I/O performance when this switch is used. Since this option may cause substantial code expansion, it is OFF by default.

–OPT:fast_sqrt[=(**ON | OFF**)]
Enable/disable the calculation of square root as `x*rsqrt(x)` for MIPS IV and above, normally enabled by **IEEE_arithmetic=3**.

–OPT:fold_reassociate[=(**ON | OFF**)]
Enable/disable transformations that 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 that 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 MIPS IV reciprocal instruction for `1.0/y`, normally enabled by **IEEE_arithmetic>=2**. See **–OPT:div_split**.

–OPT:rsqrt[=(**ON | OFF**)]
Allow use of the MIPS IV reciprocal square root instruction for `1.0/sqrt(y)`, normally enabled by **IEEE_arithmetic>=2**.

–TARG:madd[=(**ON | OFF**)]
The MIPS IV 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 separately rounded, this can produce slightly different (but more accurate) results than a separate multiply and add pair of instructions. This is normally enabled for **–mips4**.

Debugging Floating-Point Problems

The options above can change the results of floating point calculations, causing less accuracy (especially **–OPT:IEEE_arithmetic**), different results due to expression rearrangement (**–OPT:roundoff**), or NaN/Inf results in new cases. Note that in some such cases, the new results may not be worse (that is, less accurate) than the old, they just may be different. For instance, doing a sum reduction by adding the terms in a different order is likely to produce a different result. Typically, that result is not less accurate, unless the original order was carefully chosen to minimize roundoff.

If you encounter such effects when using these options (including **–O3**, which enables **–OPT:roundoff=2** by default), first attempt to identify the cause by forcing the safe levels of the options: **–OPT:IEEE_arithmetic=1:roundoff=0**.

When you do this, *do not* have the following options explicitly enabled:

–OPT:div_split
–OPT:fast_complex
–OPT:fast_exp
–OPT:fast_sqrt
–OPT:fold_reassociate
–OPT:recip
–OPT:rsqrt

If using the safe levels works, you can either use the safe levels or, if you are dealing with performance-critical code, you can use the more specific options (for example, **div_split**, **fast_complex**, and so forth) to selectively disable optimizations. Then you can identify the source code that is sensitive and eliminate the problem. Or, you can avoid the problematic optimizations.

Controlling Miscellaneous Optimizations

The following **–OPT** options allow control over a variety of optimizations.

–OPT:space The MIPSpro 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 forth.

–OPT:alias=*name*

The compilers must normally be very conservative in optimization of memory references involving pointers (especially in C), since aliases (that is, 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. **alias** options are listed below.

–OPT:alias=any The compiler assumes that any pair of memory references may be aliased unless it can prove otherwise. This is the default.

–OPT:alias=typed

The compiler assumes that any pair of memory references that 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.

–OPT:alias=unnamed

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

```
float g;
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.

–OPT: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.

Controlling Loop Unrolling

The following options control loop unrolling in the MIPSpro 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 run-time benefits against code expansion. These options allow you 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; it has single entry at the top and a single exit at the bottom.)

Controlling Global Code Motion

An important optimization performed by the MIPSpro compilers is 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 that may 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 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 that are to be allowed.

Valid speculative code motion must normally avoid moving operations that may cause run-time traps. As a result, turning off certain traps at run time 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 that are already using most of the instruction execution resources available, since doing so will likely extend 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 that 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). Bottom loading, 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 end of 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 that 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 run time. Since this may prevent some truly invalid references from causing faults, avoid this option during debugging.

The second form involves moving a reference like `*(p+n)`, for a small integer `n`, to a block that 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;
} else {
     sum += p->final_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), as can the load of `p->final_val` (it is offset by a small amount from the `p->next` reference).

–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 **–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.

Controlling Software Pipelining

Software pipelining (SWP), an important optimization for the inner loops of programs, can cause dramatic improvement by rearranging the loops to overlap calculations from multiple iterations. This iterative process searches for an effective schedule and then for a workable allocation of registers, and then retries if either step fails. Some of the options in the **–SWP** group control that process. Other options 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=n" option.

SWP must normally be careful during the initial and final iterations of a loop to not perform extra operations that may cause run-time 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 run time 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 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 that make recurrences less severe by substituting the expression that 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 2 of 4 cycles per iteration on the R8000. Back-substitution can transform the loop to code equivalent to:

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

With appropriate initialization, this version can achieve an effective iteration interval of nearly 0.5 cycles.

–SWP:ii_backtracks_max=*n*

This option controls the limit on how many times SWP backtracks and searches for a schedule with *ii* (iteration interval) cycles before giving up and increasing *ii*. Increasing the limit improves its chances of success; decreasing it may improve the compilation time required.

–SWP:body_ins_count_max=*n*

SWP is not attempted for loop bodies containing more than *n* instructions (the default is 100, 0 for no limit). Larger loop bodies may be successfully pipelined, but take more compilation time in the attempt, so there may be a tradeoff of code improvement vs. compile time.

Loop bodies are 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**.)

For example, compile *tomcatv* (a spec benchmark) with:

```
 f77 -S -64 -O3 -mips4 -OPT:IEEE_arith=3:ro=3 -LIST:=ON
tomcatv.f
```

A listing file *tomcatv.L* is produced. The results for the loop at line 32 in look like this:

```
Compiling tomcatv.f (tomcatv.f)
Options:
  -O3   (Optimization level)
  -g0   (Debug level)
  -m1   (Report warnings)

  -TARG:        (Target group)
    abi=64                      (64-bit ABI)
    isa=mips4                   (Instruction Set Architecture)
    processor=R8000
    madd=ON             (Allow madd instructions)

  -TENV:        (Target environment group)
    PIC=ON                      (Shared code)
    small_GOT                   (Assume GOT < 64KB)
    no_page_offset=OFF          (Use page/offset addressing)
```

```
    short_data=8                (Size limit for short data objects)
    short_literals=8            (Size limit for short literals)
    misalignment=0              (Misaligned data model)
    align_aggregrates=8         (Forced aggregate alignment)
    use_fp=OFF                  (Force frame pointer use)
    varargs_prototypes=ON       (Require prototypes for varargs routines)
    X=1                         (Exception suppression model)

  -OPT:         (Optimization group)
    div_split=OFF               (Use a*(1/b) for a/b)
    fast_complex=OFF            (Use fast complex norm/sqrt)
    fast_exp=OFF                (Use fast exp algorithm)
    fast_sqrt=OFF               (Use 1/rsqrt(x) for sqrt(x))
    fast_io=OFF                 (Use fast I/O intrinsics)
    fold_aggressive=OFF         (Use aggressive expression folding)
    IEEE_arithmetic=3           (Level of IEEE-754 compliance)
    IEEE_comparisons=OFF        (Don't eliminate comparisons like x==x)
    roundoff=3                  (Level of roundoff errors allowed)
    space=OFF                   (Optimize code space over execution time)
    vector_intrinsics=OFF       (Use vector intrinsics)

  -LIST:        (Listing group)
    =ON                         (Produce listing file)
    file=tomcatv.L              (Listing file name)
    performance=ON              (List performance information)
    source=OFF                  (List source code)
    symbols=OFF                 (List symbol table)
...
#<swps> Pipelined loop line 32 steady state
#<swps>
#<swps>     Not unrolled before pipelining
#<swps>     4 cycles per iteration
#<swps>     1 flop         (  6% of peak) (madds count as 2)
#<swps>     1 flop         ( 12% of peak) (madds count as 1)
#<swps>     0 madds        (  0% of peak)
#<swps>     8 mem refs     (100% of peak)
#<swps>     3 integer ops  ( 37% of peak)
#<swps>    12 instructions ( 75% of peak)
#<swps>     4 short trip threshold
#<swps>
#<swps>     4 possible stall cycles
#<swps>     4 min possible stall cycles
#<swps>
#<swps>
```

If you add the switch **–SWP:body_ins=250**, the amount of unrolling increases. The loop at line 32 becomes:

```
#<swps> Pipelined loop line 32 steady state
#<swps>
#<swps>   2 unrollings before pipelining
#<swps>   7 cycles per 2 iterations
#<swps>   2 flops       (  7% of peak) (madds count as 2)
#<swps>   2 flops       ( 14% of peak) (madds count as 1)
#<swps>   0 madds       (  0% of peak)
#<swps>  14 mem refs    (100% of peak)
#<swps>   3 integer ops ( 21% of peak)
#<swps>  19 instructions( 67% of peak)
#<swps>   2 short trip threshold
#<swps>
#<swps>   6 possible stall cycles
#<swps>   4 min possible stall cycles
```

–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 (MIPS IV) 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 is compiled for MIPS IV as:

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

Note that no conditional branches occur in the code. This option is ON by default for MIPS IV targets only.

–SWP:interleave_reductions[=(**ON**|**OFF**)]

This option, ON by default, has the same motivation as back-substitution. It allows transformations that 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 iteration interval of 4 cycles/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 = sum1+sum2+sum3+sum4+sum5+sum6+sum7+sum8
```

This version can achieve an effective iteration interval 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* (the default is 5). The limit is applied via a run-time 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 2 for MIPS IV, and 1 for MIPS I-III. Unrolling is also constrained by 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**.)

The **–S** option to the compiler provides information about software pipelining, the loops it works on, and the results. See "General Options for Compiler Drivers" for more information.

Controlling the Target Architecture

Some options control the target architecture for which code is generated; most of the options are in the **–TARG**: option group. The options are described below.

–32 –n32 or **–64** This option determines the base ABI to be assumed, either the 32-bit ABI for MIPS I/II targets, or the new 32-bit (n32) and 64-bit ABI for MIPS III/IV targets.

–mips[**1234**] or **–TARG:isa=mips**[**1234**]
This option identifies the instruction set architecture (ISA) to be used for generated code. It defaults to MIPS I for 32-bit compilations, and to MIPS III for n32 and 64-bit compilations. However, you can specify higher ABIs for either, at the cost of producing an executable that will not run on the earlier architectures. The linker (*ld*) does not understand the **–TARG** group, so you must use the short form if linking is to be invoked as part of the compilation.

–TARG:madd[=(**ON**|**OFF**)]
This option enables/disables use of the multiply/add instructions for MIPS IV targets. These instructions multiply two floating point operands and then add (or subtract) a third with a single roundoff of the final result. They are therefore slightly more accurate than the usual discrete operations, and may cause results not to match baselines from other targets. This option may be used to determine whether observed differences are due to **madd**s. This option is ON by default for MIPS IV targets; otherwise, it is ignored.

–TARG:processor=(**r3000**|**r4000**|**r8000**)
This option identifies the probable execution target of the code to be generated; it will be scheduled for optimal execution on that target. This option does not affect the ABI and/or ISA selected by the options described above (that is, it has no effect on which target processors can correctly execute the program, but just on how well they do so).

The command below produces MIPS III code conforming to the 64-bit ABI (and therefore executable on any MIPS III or above processor), which is optimized to run on the R8000.

```
cc -64 -TARG:proc=r8000 ...
```

Controlling the Target Environment

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

The first **–TENV** 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 run time. A dedicated register (**$gp**) points to the GOT at run time, and the code can load these addresses from the GOT without being dependent on its actual virtual address. If the GOT is less than 64KB in size, those loads can all be single instructions; otherwise they require adding a constructed offset to **$gp**. These options choose one of these cases (the default is **small_GOT**).

–TENV:large_GOT is a synonym for the compiler option **–xgot**. For information on *gp* overflow, see *gp_overflow*(5).

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 trap enable state at run time.

–TENV:X=*n* Specify the level (0 to 5, the default is 1) of enabled traps that are 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 traps may be disabled or ignored.

Use nondefault levels with great care. Disabling traps eliminates a useful debugging tool, since the problems that cause them 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 down a program significantly if it causes frequent traps.

Disabling traps in one module requires disabling them for the entire program. Programs that make use of level 2 or above should not attempt explicit manipulation of the hardware trap enable flags. Furthermore, libraries (including DSOs) being built for incorporation in many client programs should generally avoid using this option, since using it makes debugging of the client programs difficult, and can prevent them from safely making use of the hardware trap enables.

–TENV:align_aggregates=*n*

The ABI specifies that aggregates (that is, **struct**s and **array**s) be aligned according to the strictest requirements of their components (fields or elements). Thus, an array of **short int**s (or a struct containing only **short int**s or **char**s) normally is 2-byte aligned. However, some code (non-ANSI conforming) 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 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 don'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 example, 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`.

Improving Global Optimization

This section describes the global optimizer and contains coding hints. Specifically this section includes:

- "Overview of the Global Optimizer"
- "Optimizing C and Fortran Programs"
- "Improving Other Optimization"
- "Register Allocation"

Overview of the Global Optimizer

The global optimizer is part of the compiler back end. It improves the performance of object programs by transforming existing code into more efficient coding sequences. The optimizer distinguishes between C, C++, and Fortran programs to take advantage of the various language semantics.

Optimizing C and Fortran Programs

The following suggestion applies to both C and Fortran programs.

Do not use indirect calls (that is, calls via function pointers, including those passed as subprogram arguments). Indirect calls may cause unknown side effects (for instance, changing global variables) reducing the amount of optimization possible.

Optimizing C Programs

The following suggestions apply to C programs only.

Return values. Use functions that return values instead of pointer parameters.

Do while. Use **do while** when possible, instead of **while** or **for**. For **do while**, the optimizer does not have to duplicate the loop condition in order to move code from within the loop to outside the loop.

Unions. Avoid unions that cause overlap between integer and floating point data types. The optimizer can not assign such fields to registers.

Local variables. Use local variables and void global variables. In C programs, declare any variable outside of a function as static, unless that variable is referenced by another source file. Minimizing the use of global variables increases optimization opportunities for the compiler.

Value parameters. Pass parameters by value instead of passing by reference (pointers) or using global variables. Reference parameters have the same performance-degrading effects as the use of pointers (see below).

Pointers and aliasing. *Aliases* occur when there are multiple ways to reference the same data object. For instance, when the address of a global variable is passed as a subprogram argument, it may be referenced either using its global name, or via the pointer. The compiler must be conservative when dealing with objects that may be aliased, for instance keeping them in memory instead of in registers, and carefully retaining the original source program order for possibly aliased references.

Pointers in particular tend to cause aliasing problems, since it is often impossible for the compiler to identify their target objects. Therefore, you can help the compiler avoid possible aliases by introducing local variables to store the values obtained from

dereferenced pointers. Indirect operations and calls affect dereferenced values, but do not affect local variables. Therefore, local variables can be kept in registers. The following example shows how the proper placement of pointers and the elimination of aliasing produces better code.

Example of Pointer Placement and Aliasing

In the example below, if `len>10` (for instance because it is changed in another function before calling zero), `*p++ = 0` will eventually modify len. Therefore, the compiler cannot place `len` in a register for optimal performance. Instead, the compiler must load it from memory on each pass through the loop.

Consider the following source code:

```
char a[10];
int len = 10;

void
zero()
{
   char *p;
   for (p= a; p != a + len; ) *p++ = 0;
}
```

The generated assembly code looks like this:

```
#8  for (p = a; p!= a + len; )  *p++ = 0;
       move   $2, $4
       lw     $3, len
       addu   $24, $4, $3
       beq    $24  $4  $33      # a + len != p
$32:
       sb     $0, 0($2)         # *p = 0
       addu   $2, $2, 1         # p++
       lw     $25, len
       addu   $8, $4, $25
       bne    $8, $2, $32       # a + len != p
$33:
```

You can increase the efficiency of this example by using subscripts instead of pointers, or by using local variables to store unchanging values.

Using subscripts instead of pointers. Using subscripts in the procedure `azero` (as shown below) eliminates aliasing. The compiler keeps the value of `len` in a register, saving two instructions. It still uses a pointer to access `a` efficiently, even though a pointer is not specified in the source code. For example, consider the following source code:

```
char a[10];
int len = 10;
void azero()
{
    int i;
    for ( i = 0; i != len; i++ )
    a[i] = 0;
}
```

The generated assembly code looks like this:

```
       for (i = 0; i != len; i++ ) a[i] = 0;
       move   $2, $0          # i = 0
       beq    $4, 0, $37      # len != a
       la     $5, a
$36:
       sb     $0, 0($5)       # *a = 0
       addu   $2, $2, 1       # i++
       addu   $5, $5, 1       # a++
       bne    $2, $4, $36     # i != len
$37:
```

Using local variables. Using local (automatic) variables or formal arguments instead of static or global prevents aliasing and allows the compiler to allocated them in registers.

For example, in the following code fragment, the variables `loc` and `form` are likely to be more efficient than `ext*` and `stat*`.

```
extern int ext1;
static int stat1;

void p ( int form )
{
   extern int ext2;
   static int stat2;
   int loc;
   ...
}
```

Write straightforward code. For example, do not use **++** and **- -** operators within an expression. Using these operators for their values, rather than for their side-effects, often produces bad code.

Addresses. Avoid taking and passing addresses (& values). Using addresses creates aliases, makes the optimizer store variables from registers to their home storage locations, and significantly reduces optimization opportunities that would otherwise be performed by the compiler.

VARARG/STDARG. Avoid functions that take a variable number of arguments. The optimizer saves all parameter registers on entry to VARARG or STDARG functions. If you must use these functions, use the ANSI standard facilities of *stdarg.h*. These produce simpler code than the older version of *varargs.h*

Improving Other Optimization

The global optimizer processes programs only when you specify the **–O2** or **–O3** option at compilation. However, the code generator phase of the compiler always perform certain optimizations.

This section contains coding hints that increase optimization for the other passes of the compiler.

C and Fortran Programs

The following suggestions apply to both C and Fortran programs:

- Use tables rather than **if-then-else** or **switch** statements. For example:

  ```
  typedef enum { BLUE, GREEN, RED, NCOLORS } COLOR;
  ```

  Instead of:

  ```
  switch ( c ) {
     case CASE0: x = 5; break;
     case CASE1: x = 10; break;
     case CASE2: x = 1; break;
  }
  ```

  Use:

  ```
  static int Mapping[NCOLORS] = { 5, 10, 1 };
  ...
  x = Mapping[c];
  ```

- As an optimizing technique, the compiler puts the first eight parameters of a parameter list into registers where they may remain during execution of the called routine. Therefore, always declare, as the first eight parameters, those variables that are most frequently manipulated in the called routine.
- Use word-size scalar variables instead of smaller ones. This practice can take more data space, but produces more efficient code.

C Programs Only

The following suggestions apply to C programs only:

- Rely on *libc.so* functions (for example, **strcpy**, **strlen**, **strcmp**, **bcopy**, **bzero**, **memset**, and **memcpy**). These functions are carefully coded for efficiency.
- Use a signed data type, or cast to a signed data type, for any variable that does not require the full unsigned range and must be converted to floating-point. For example:

```
double d;
unsigned int u;
int i;
/* fast */ d = i;
/* fast */ d = (int)u;
/* slow */ d = u;
```

  Converting an unsigned type to floating-point takes significantly longer than converting signed types to floating-point; additional software support must be generated in the instruction stream for the former case.

- Use signed **int**s in 64-bit code if they may appear in mixed type expressions with **long int**s. Since the hardware automatically sign-extends the results of most 32-bit operations, this may avoid explicit zero-extension code. For example:

```
unsigned int ui;
signed int i;
long int li;
/* fast */ li += i;
/* fast */ li += (int)ui;
/* slow */ li += ui;
```

Register Allocation

The MIPS architecture emphasizes the use of registers. Therefore, register usage has a significant impact on program performance.

The optimizer allocates registers for the most suitable data items, taking into account their frequency of use and their locations in the program structure. In addition, the optimizer assigns values to registers in such a way as to minimize shifting around within loops and during procedure invocations.