Chapter 4

Optimizing Program Performance

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

- "Optimization Overview"
- "Using the Optimization Options"
- "Performance Tuning with Interprocedural Analysis"
- "Controlling Loop Nest Optimizations"
- "Controlling Floating Point Optimization"
- "The Code Generator"
- "Controlling the Target Architecture"
- "Controlling the Target Environment"
- "Improving Global Optimization"

Note: Please see the *Release Notes* and reference page for your compiler for a complete list of options that you can use to optimize and tune your program.

You can find additional information about optimization in *MIPSpro 64-Bit Porting and Transition Guide*, Chapter 6, "Performance Tuning." For information about writing code for 64-bit programs, see Chapter 5, "Coding for 64-Bit Programs." For information about porting code to N32 and 64-bit systems, see Chapter 6, "Porting Code to N32 and 64-Bit Silicon Graphics Systems."

Optimization Overview

This section covers optimization benefits 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. For information on the debugger, see *dbx User's Guide*.

Using the Optimization Options

This section lists and briefly describes the optimization options, **–O0** through **–O3**.

Invoke the optimizer by specifying a compiler, such as *cc*(1), with any of the options listed in Table 4-1.

Table 4-1 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.
-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=2** option below).

Refer to your compiler's reference page and *Release Notes* for details on the optimization options and all other options.

Performance Tuning with Interprocedural Analysis

Interprocedural Analysis (IPA) performs program optimizations that can only be done in the presence of the whole program. Some of the optimizations it performs also allow downstream phases to perform better code transformations.

Currently IPA optimizes code by performing:

- procedure inlining
- interprocedural constant propagation of formal parameters
- dead function elimination
- identifi cation of global constants
- dead variable elimination
- pic optimization
- automatic selection of candidates for the gp-relative area (*autognum*)
- dead call elimination
- automatic internal padding of COMMON arrays in Fortran
- interprocedural alias analysis

Figure 4-1 shows interprocedural analysis and interprocedural optimization phase of the compilation process.

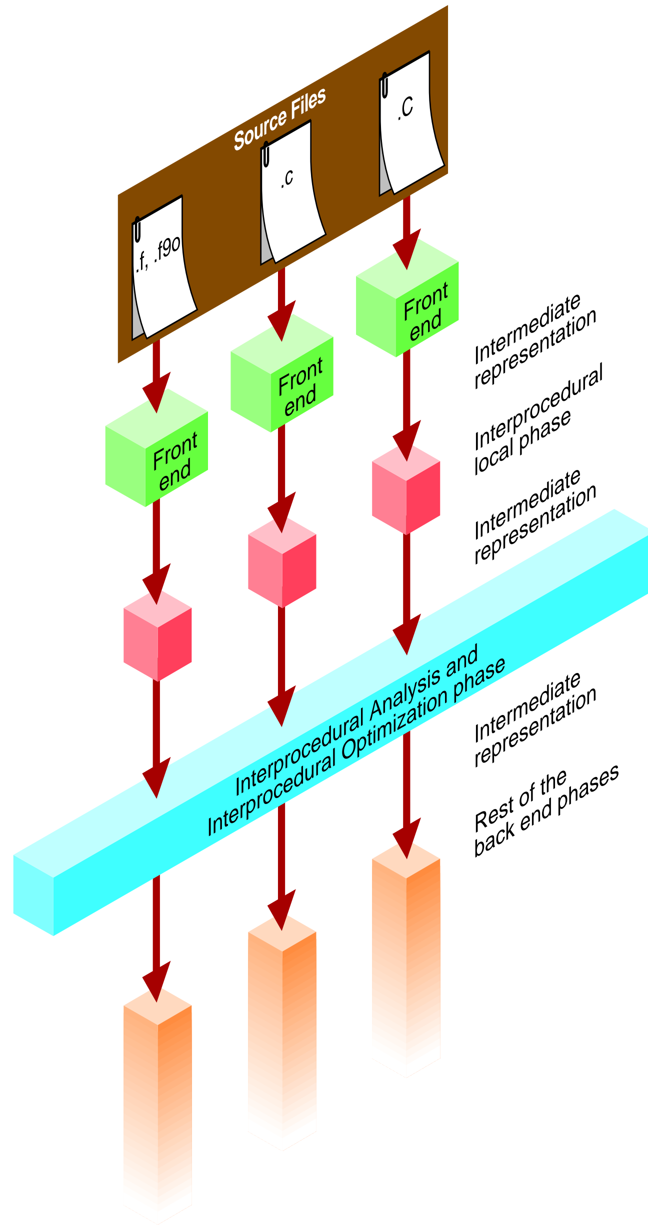

Figure 4-1 Compilation Process Showing Interprocedural Analysis

Typically, you invoke IPA with the **–IPA:** option group to *f77*, *cc*, *CC*, and *ld*. Its inlining decisions are also controlled by the **–INLINE:** option group. Up-to-date information on IPA and its options is in the ipa(5) reference page.

This section covers some IPA options including:

- "Inlining"
- "Common Block Padding"
- "Alias and Address Taken Analysis"

Inlining

IPA performs across and within file inlining. A default inlining heuristic determines which calls to inline. This section covers the following information:

- "Benefits of Inlining"
- "Inlining Options for Routines"
- "Options To Control Inlining Heuristics"

Benefits of Inlining

Your code may benefit from inlining for the following reasons:

- Inlining exposes a larger context to the scalar and loop-nest optimizers, thereby allowing more optimizations to occur.
- Inlining eliminates overhead resulting from the call (for example, register save and restore, the call and return instructions, and so forth). Instances occur, however, when inlining may hurt run-time performance due to increased demand for registers, or compile-time performance due to code expansion. Hence extensive inlining is not always useful. You must select callsites for inlining based on certain criteria such as frequency of execution and size of the called procedure. Often it is not possible to get an accurate determination of frequency information based on compile-time analysis. As a result, inlining decisions may benefit from generating feedback and providing the feedback file to IPA. The inlining heuristic will perform better since it is able to take advantage of the available frequency information in its inlining decision.

Inlining Options for Routines

You may wish to select certain procedures to be inlined or not to be inlined by using any of the options listed in Table 4-2.

Table 4-2 Inlining Options for Routines

Inline Option	Description
-INLINE:=OFF	Suppresses inlining. **-IPA:inline=OFF** also suppresses inlining.
-INLINE:must= ... **-INLINE:never=** ...	Allows you to specify the desired action for specific routines.
-INLINE:none **-INLINE:all**	Inlines all (or none) of the routines not covered by the above options.
-INLINE:file=<filename>	Provides cross-file inlining.

These options are covered in more detail in the subsections below.

Note: You can use the inline keyword and pragmas in C++ or C to specifically identify routines to callsites to inline. The inliner's heuristics decides whether or not to inline any cases not covered by the **-INLINE** options in the preceding table.

In all cases, once a call is selected for inlining, a number of tests are applied to verify its suitability. These tests may prevent its inlining regardless of user specification, for instance if the callee is a C varargs routine, or parameter types don't match.

The -INLINE:none and -INLINE:all Options

At call sites not marked by inlining pragmas, the compiler does not attempt to inline routines not specified with the **must** option or a routine pragma requesting inlining; it observes site inlining pragmas.

At call sites not marked by inlining pragmas, the compiler attempts to inline all routines not specified with the **never** option or a routine pragma requesting no inlining; it observes site inlining pragmas.

If you specify both **all** and **none**, **none** is ignored with a warning.

The -INLINE:must and -INLINE:never Options

If **-INLINE:must=** *routine_name*<,*routine_name*>* is specified, the compiler attempts to inline the associated routines at call sites not marked by inlining pragmas, but does not inline if varargs or similar complications prevent it. It observes site inlining pragmas.

Equivalently, you can mark a routine definition with a pragma requesting inlining.

If **-INLINE:never=** *routine_name*<,*routine_name*>* is specified, the compiler does not inline the associated routines at call sites not marked by inlining pragmas; it observes site inlining pragmas.

Note: For C++, you must provide mangled routine names.

The -INLINE:*file=<filename>* Option

This option invokes the standalone inliner, which provides cross-file inlining. The option **-INLINE:*file=<filename>*** searches for routines provided via the **-INLINE:must** list option in the file specified by the **-INLINE:file** option. The file provided in this option must be generated using the **-IPA -c** options. The file generated contains information used to perform the cross file inlining.

For example, suppose two files exist: *foo.f* and *bar.f*.

The file, *foo.f*, looks like this:

```
program main
   ...
   call bar()
end
```

The file, *bar.f*, looks like this:

```
subroutine bar()
...
end
```

To inline **bar** into **main**, using the standalone inliner, compile with **-IPA** and **-c** options:

```
f77 -n32 -IPA -c bar.f
```

This produces the file, *bar.o*. To inline **bar** into *foo.f*, enter:

```
f77 -n32 foo.f -INLINE:must=bar:file=bar.o
```

Options To Control Inlining Heuristics

Group options control the inlining heuristics used by IPA are listed in Table 4-3.

Table 4-3 Options to Control Inlining Heuristics

Option	Description
-IPA:maxdepth=*n*	Inline nodes at a depth less than or equal to *n* in the call graph. Leaf nodes are at depth 0. Inlining is still subject to space limit (see space and Olimit below).
-IPA:forcedepth=*n*	Inline nodes at a depth less than or equal to in the call graph regardless of the size of the procedures and total program size. Leaf nodes are at depth 0. You may use this option to force the inlining of, for example, leaf routines.
-IPA:space=*n*	Inline until the program expands by a factor of *n*% is reached. For example, *n*=**20** causes inlining to stop once the program has grown in size by 20%. You may use this option to limit the growth in program size.
-IPA:plimit=*n*	Inline calls into a procedure until the procedure has grown to a size of *n*, where *n* is a measure of the size of the procedure. This may be used to control the size of each program unit. The current default procedure limit is 2000.
-OPT:Olimit=*n*	Controls the size of procedures that the global optimizer will process, measured as for plimit. IPA will avoid inlining that makes a procedure larger than this limit as well. Unlike plimit, a value of *n*=**0** specifies unlimited.

Common Block Padding

Padding global arrays (in Fortran) reduces cache confl icts and can signifi cantly improve performance. Several current restrictions exist which limits IPA padding of common arrays. The current restrictions are as follows:

1. The shape of the common block to which the global array belongs must be consistent across procedures. That is, the declaration of the common block must be the same in every subroutine that declares it.

   In the example below, IPA can not pad any of the arrays in the common block because the shape is not consistent.

   ```
   program main
       common /a/ x(1024,1024), y(1024, 1024), z(1024,1024)
       ....
       ....
   end

   subroutine foo
       common /a/ xx(100,100), yy(1000,1000), zz(1000,1000)
       ....
       ....
   end
   ```

2. The common block variables must not initialize data associated with them. In this example, IPA can not pad any of the arrays in common block `/a/`:

   ```
   block data inidata
       common /a/ x(1024,1024), y(1024,1024), z(1024,1024), b(2)
        DATA b /0.0, 0.0/
   end

   program main
       common /a/ x(1024,1024), y(1024,1024), z(1024,1024), b(2)
       ....
       ....
   end
   ```

3. The array to be padded may be passed as a parameter to a routine only if it declared as a one dimensional array, since passing multi-dimensional arrays that may be padded can cause the array to be re-shaped in the callee.

4. Restricted types of equivalences to arrays that may be padded are allowed. Equivalences that do not intersect with any column of the array are allowed. This implies an equivalencing that will not cause the equivalenced array to access

invalid locations. In the example below, the arrays in `common /a/` will not be padded since `z` is equivalenced to `x(2,1)`, and hence `z(1024)` is equivalenced to `x(1,2)`.

```
program main
    real z(1024)
    common /a/ x(1024,1024), y(1024,1024) equivalence (z, x(2,1))
    ....
    ....
end
```

5. The common block symbol must have an INTERNAL or HIDDEN attribute, which implies that the symbol may not be referenced within a DSO that has been linked with this program.
6. The common block symbol can not be referenced by regular object files that have been linked with the program.

Alias and Address Taken Analysis

The optimizations that are performed later in the compiler are often constrained by the possibility that two variable references may be "aliased." That is, they may be aliased to the same address. This possibility is increased by calls to procedures that aren't visible to the optimizer, and by taking the addresses of variables and saving them for possible use later (for example, in pointers). Furthermore, the compiler must normally assume that a global (extern) datum may have its address taken in another file, or may be referenced or modified by any procedure call. The IPA alias and address-taken analyses are designed to identify the actual global variable addressing and reference behavior so that such worst-case assumptions are not necessary.

The options (described below) that control these analyses are:

- "The –IPA:alias=ON Option"
- "The –IPA:addressing=ON Option"
- "The –IPA:opt_alias=ON Option"

The –IPA:alias=ON Option

This option performs IPA alias analysis. That is, it determines which global variables and formal parameters are referenced or modified by each call, and which global variables are passed to reference formal parameters. This analysis is used for other IPA analyses,

including constant propagation and address-taken analysis. This option is OFF by default.

The –IPA:addressing=ON Option

This option performs IPA address-taken analysis. That is, it determines which global variables and formal parameters have their addresses taken in ways that may produce aliases. This analysis is used for other IPA analyses, especially constant propagation. Its effectiveness is very limited without **–IPA:alias=ON**. This option is OFF by default.

The –IPA:opt_alias=ON Option

This options performs IPA alias analysis (implying **–IPA:alias=ON**), and passes the results to the global optimizer. This option is OFF by default.

Controlling Loop Nest Optimizations

Numerical programs often spend most of their time executing loops. The loop nest optimizer (LNO) performs high-level loop optimizations that can greatly improve program performance by better exploiting caches and instruction-level parallelism.

This section covers the following topics:

- "Running LNO"
- "LNO Optimizations"
- "Compiler Options for LNO"
- "Pragmas and Directives for LNO"

Running LNO

LNO is run by default when you use the **–O3** option for all Fortran, C, and C++ programs. LNO is an integrated part of the compiler back end and is not a preprocessor. Therefore, the same optimizations (with the same control options) apply to Fortran, C, and C++ programs. Note that this does not imply that LNO will optimize numeric C++ programs as well as Fortran programs. C and C++ programs often include features that make them inherently harder to optimize than Fortran programs.

After LNO performs high-level transformations, it may be desirable to view the transformed code in the original source language. Two translators that are integrated into the back end translate the compiler internal code representation back into the original source language after the LNO transformation (and IPA inlining). You can invoke either one of these translators by using the *f77* option **-FLIST:=on** or the *cc* option **-CLIST:=on**. For example, `f77 -O3 -FLIST:=on x.f` creates an *a.out* as well as a Fortran file *x.w2f.f*. The *.w2f.f* file is a readable and compilable Silicon Graphics Fortran representation of the original program after the LNO phase (see Figure 4-2). LNO is not a preprocessor, which means that recompiling the *.w2f.f* file directly may result in an executable that is different from the original compilation of the *.f* file.

The **-CLIST:=on** option may be applied to CC to translate compiler internal code to C. No translator exists to translate compiler internal code to C++. When the original source language is C++, the generated C code may not be compilable.

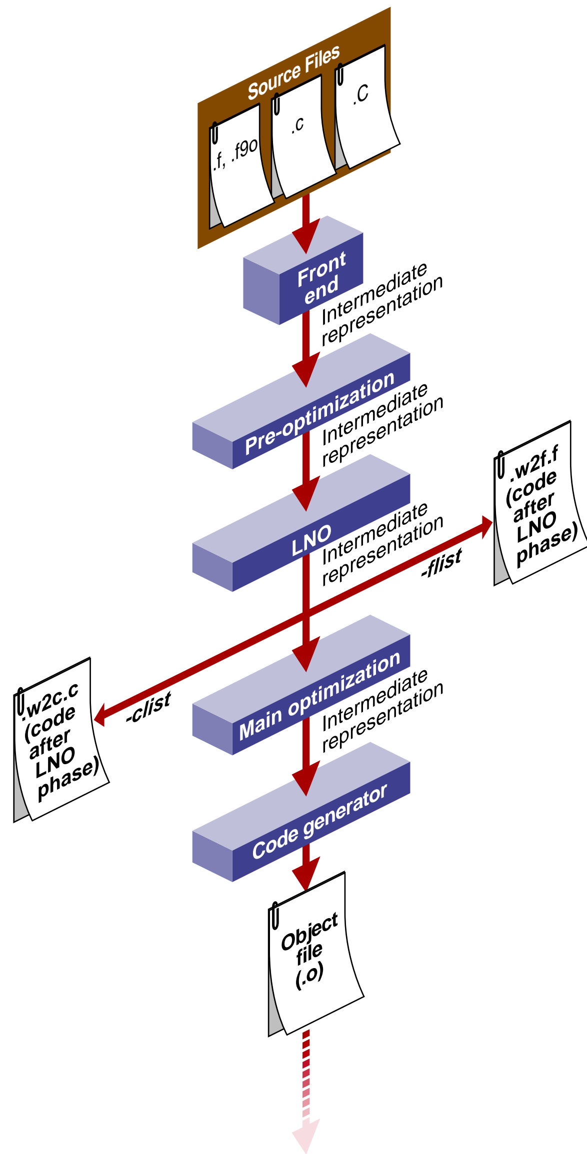

Figure 4-2 Compilation Process Showing LNO Transformations

LNO Optimizations

This section describes some important optimizations performed by LNO. For a complete listing, see your compiler's reference page. Optimizations include:

- "Loop Interchange"
- "Blocking and Outer Loop Unrolling"
- "Loop Fusion"
- "Loop Fission/Distribution"
- "Prefetching"
- "Gather-Scatter Optimization"

Loop Interchange

The order of loops in a nest can affect the number of cache misses, the number of instructions in the inner loop, and the ability to schedule an inner loop. Consider the following loop nest example.

```
do i
  do j
    do k
      a(j,k) = a(j,k) + b(i,k)
```

As written, the loop suffers from several possible performance problems. First, each iteration of the `k` loop requires two loads and one store. Second, if the loop bounds are sufficiently large, every memory reference will result in a cache miss.

Interchanging the loops improves performance.

```
do k
   do j
     do i
       a(j,k) = a(j,k) + b(i,k)
```

Since `a(j,k)` is loop invariant, only one load is needed in every iteration. Also, `b(i,k)` is "stride-1," successive loads of `b(i,k)` come from successive memory locations. Since each cache miss brings in a contiguous cache line of data, typically 4-16 elements, stride-1 references incur a cache miss every 4-16 iterations. In contrast, the references in the original loop are not in stride-1 order. Each iteration of the inner loop causes two cache misses; one for `a(j,k)` and one for `b(i,k)`.

In a real loop, different factors may affect different loop ordering. For example, choosing i for the inner loop may improve cache behavior while choosing j may eliminate a recurrence. LNO uses a performance model that considers these factors. It then orders the loops to minimize the overall execution time estimate.

Blocking and Outer Loop Unrolling

Cache blocking and outer loop unrolling are two closely related optimizations used to improve cache reuse, register reuse, and minimize recurrences. Consider matrix multiplication in the following example.

```
do i=1,10000
   do j=1,10000
    do k=1,10000
      c(i,j) = c(i,j) + a(i,k)*b(k,j)
```

Given the original loop ordering, each iteration of the inner loop requires two loads. The compiler uses loop unrolling, that is, register blocking, to minimize the number of loads.

```
do i=1,10000
   do j=1,10000,2
    do k=1,10000
      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)
```

Storing the value of a(i,k) in a register avoids the second load of a(i,k). Now the inner loop only requires three loads for two iterations. Unrolling the j loop even further, or unrolling the i loop as well, further decrease the amount of loads required. How much is the ideal amount to unroll? Unrolling more decreases the amount of loads but not the amount of floating point operations. At some point, the execution time of each iteration is limited by the floating point operations. There is no point in unrolling further. LNO uses its performance model to choose a set of unrolling factors that minimizes the overall execution time estimate.

Given the original matrix multiply loop, each iteration of the i loop reuses the entire b matrix. However, with sufficiently large loop limits, the matrix b will not remain in the cache across iterations of the i loop. Thus in each iteration, you have to bring the entire matrix into the cache. You can "cache block" the loop to improve cache behavior.

```
do tilej=1,10000,Bj
  do tilek=1,10000,Bk
   do i=1,10000
    do j=tilej,MIN(tilej+Bj-1,10000)
      do k=tilek,MIN(tilek+Bk-1,10000)
        c(i,j) = c(i,j) + a(i,k)*b(k,j)
```

By appropriately choosing `Bi` and `Bk`, `b` remains in the cache across iterations of `i`, and the total number of cache misses is greatly reduced.

LNO automatically caches tile loops with block sizes appropriate for the target machine. When compiling for a Silicon Graphics R8000, LNO uses a single level of blocking. When compiling for a Silicon Graphics systems (such as R4000, R5000, or R10000) that contain multi-level caches, LNO uses multiple levels of blocking where appropriate.

Loop Fusion

LNO attempts to fuse multiple loop nests to improve cache behavior, to lower the number of memory references, and to enable other optimizations. Consider the following example.

```
do i=1,n
  do j=1,n
    a(i,j) = b(i,j) + b(i,j-1) + b(i,j+1)

do i=1,n
  do j=1,n
    b(i,j) = a(i,j) + a(i,j-1) + a(i,j+1)
```

In each loop, you need to do one store and one load in every iteration (the remaining loads are eliminated by the software pipeliner). If *n* is suffi ciently lage, in each loop you need to bring the entire `a` and `b` matrices into the cache.

LNO fuses the two nests and creates the following single nest:

```
do i=1,n
  a(i,1) = b(i,0) + b(i,1) + b(i,2)
  do j=2,n
    a(i,j) = b(i,j) + b(i,j-1) + b(i,j+1)
    b(i,j-1) = a(i,j-2) + a(i,j-1) + a(i,j)
  end do
  b(i,n) = a(i,n-1) + a(i,n) + a(i,n+1)
end do
```

Fusing the loops eliminates half of the cache misses and half of the memory references. Fusion can also enable other optimizations. Consider the following example:

```
do i
   do j1
     S1
   end do
   do j2
     S2
   end do
 end do
```

By fusing the two inner loops, other transformations are enabled such as loop interchange and cache blocking.

```
do j
   do i
     S1
     S2
   end do
end do
```

As an enabling transformation, LNO always tries to use loop fusion (or fi ssion, discussed below) to create larger perfectly nested loops. In other cases, LNO decides whether or not to fuse two loops by using a heuristic based on loop sizes and the number of variables common to both loops.

Loop Fission/Distribution

The opposite of fusing loops is distributing loops into multiple pieces, or loop fi ssion. As with fusion, fi ssion is useful as an enabling transformation. Consider this example again:

```
do i
   do j1
     S1
   end do
   do j2
     S2
   end do
end do
```

Using loop fi ssion, as shown belowalso enables loop interchange and blocking.

```
do i1
   do j1
```

```
      S1
   end do
end do
do i2
   do j2
      S2
   end do
end do
```

Loop fission is also useful to reduce register pressure in large inner loops. LNO uses a model to estimate whether or not an inner loop is suffering from register pressure. If it decides that register pressure is a problem, fission is attempted. LNO uses a heuristic to decide on how to divide the statements among the resultant loops.

Loop fission can potentially lead to the introduction of temporary arrays. Consider the following loop.

```
do i=1,n
   s = ..
   .. = s
end do
```

If you want to split the loop so that the two statements are in different loops, you need to scalar expand `s`.

```
do i=1,n
   tmp_s(i) = ..
end do
do i=1,n
   .. = tmp_s(i)
end do
```

Space for `tmp_s` is allocated on the stack to minimize allocation time. If *n* is very large, scalar expansion can lead to increased memory usage, so the compiler blocks scalar expanded loops. Consider the following example:

```
do se_tile=1,n,b
   do i=se_tile,MIN(se_tile+b-1,n)
      tmp_s(i) = ..
   end do
   do i=se_tile,MIN(se_tile+b-1,n)
      .. = tmp_s(i)
   end do
end do
```

Related to loop fission is vectorization of intrinsics. The Silicon Graphics math libraries support vector versions of many intrinsic functions that are faster than the regular versions. That is, it is faster, per element, to compute *n* cosines than to compute a single cosine. LNO attempts to split vectorizable intrinsics into their own loops. If successful, each such loop is collapsed into a single call to the corresponding vector intrinsic.

Prefetching

The MIPS IV instruction set supports a data prefetch instruction that initiates a fetch of the specified data item into the cache. By prefetching a likely cache miss sufficiently ahead of the actual reference, you can increase the tolerance for cache misses. In programs limited by memory latency, prefetching can change the bottleneck from hardware latency time to the hardware bandwidth. By default, prefetching is enabled at **-O3** for the R10000.

LNO runs a pass that estimates which references will be cache misses and inserts prefetches for those misses. Based on the miss latency, the code generator later schedules the prefetches ahead of their corresponding references.

By default, for misses in the primary cache, the code generator moves loads early in the schedule ahead of their use, exploiting the out-of-order execution feature of the R10000 to hide the latency of the primary miss. For misses in the secondary cache, explicit prefetch instructions are generated.

Prefetching is limited to array references in well behaved loops. As loop bounds are frequently unknown at compile time, it is usually not possible to know for certain whether a reference will miss. The algorithm therefore uses heuristics to guess.

Gather-Scatter Optimization

Software pipelining attempts to improve performance by executing statements from multiple iterations of a loop in parallel. This is difficult when loops contain conditional statements. Consider the following example:

```
do i = 1,n
   if (t(i) .gt. 0.0) then
     a(i) = 2.0*b(i-1)
   end do
end do
```

Ignoring the IF statement, software pipelining may move up the load of `b(i-1)`, effectively executing it in parallel with earlier iterations of the multiply. Given the conditional, this is not strictly possible. The code generator will often IF convert such loops, essentially executing the body of the IF on every iteration. IF conversion does not work well when the 'if' is frequently not taken. An alternative is to gather-scatter the loop, so the loop is divided as follows:

```
inc = 0 do i = 1,n
   tmp(inc) = i
   if (t(i) .gt. 0.0) then
     inc = inc + 1
   end do
end do

do i = 1,inc
   a(tmp(i)) = 2.0*b((tmp(i)-1)
end do
```

The code generator will IF convert the first loop; however, no need exists to IF convert the second one. The second loop can be effectively software pipelined without having to execute unnecessary multiplies.

Compiler Options for LNO

The next sections describe the compiler options for LNO. Specifically topics include:

- "Controlling LNO Optimization Levels"
- "Controlling Fission and Fusion"
- "Controlling Gather-Scatter"
- "Controlling Cache Parameters"
- "Controlling Blocking and Permutation Transformations"
- "Controlling Prefetch"

All of the LNO optimizations are on by default when you use the **-O3** compiler option. To turn off LNO at **-O3**, use **-LNO:opt=0**. If you want direct control, you can specify options and pragmas to turn on and off optimizations that you require.

Controlling LNO Optimization Levels

Table 4-4 lists LNO options that control optimization levels.

Table 4-4 LNO Options to Control Optimization Levels

Option	Description
-LNO:opt={0,1}	Provides general control over the LNO optimization level. **0** Computes dependence graph to be used by later passes. Removes inexecutable loops and IF statements. Guards DO loops so that every DO loop is guaranteed to have at least one iteration. **1** Provides full LNO transformations.
-LNO:ignore_pragmas	By default, pragmas within a file override the command-line options. This option allows command-line options to override the pragmas in the file.

Controlling Fission and Fusion

Table 4-5 lists LNO options that control fission and fusion.

Table 4-5 LNO Options to Control Fission and Fusion

Option	Description
-LNO:fission={0,1,2}	**0** Performs no fission. **1** Uses normal heuristics when deciding on loop fission (the default). **2** Tries fission before fusion when trying to create perfect nests, and fissions inner loops as much as possible.
-LNO:fusion={0,1,2}	**0** Performs no fusion. **1** Uses normal heuristics when deciding on loop fusion (the default). **2** Fuses outer loops even if fusing destroys perfect nests; tries fusion before fission when trying to create perfect nests.

Table 4-5 (continued) LNO Options to Control Fission and Fusion

Option	Description
-LNO:fusion_peeling_limit=n	Sets the limit (n>=0) for number of iterations allowed to be peeled in fusion. The default is 5.
-LNO:fission_inner_register_limit=n	Sets the limit (n>=0) for estimated register usage of loop bodies after inner loop fission. The default is processor specific.

Note: If both **-LNO:fission** and **-LNO:fusion** are set to 1 or 2, fusion is preferred.

Controlling Gather-Scatter

Table 4-6 lists the LNO option that controls gather-scatter.

Table 4-6 LNO Option to Control Gather-Scatter

Option	Description
-LNO:gather_scatter={0,1,2}	Controls gather-scatter. **0** Does not perform the gather-scatter optimization. **1** Gather-scatters non-nested IF statements. The default is 1. **2** Performs multi-level gather-scatter.

Controlling Cache Parameters

The options **-r5000** , **-r8000** , **-r10000** set a series of default cache characteristics. To override a default setting, use one or more of the options below.

To define a cache entirely, you must specify all options immediately following the **-LNO:cache_size** option. For example, if the processor is an R4000 (**r4k**), which has no secondary cache, then specifying **-LNO:cache_size2=4m** is not valid unless you supply the options necessary to specify the other characteristics of the cache. (Setting **-LNO:cache_size2=0** is adequate to turn off the second level cache; you don’t have to specify other second-level parameters.) Options are available for third and fourth level caches. Currently none of the Silicon Graphics machines have such caches. However, you can also use those options to block other levels of the memory hierarchy.

For example, on a machine with 128Mb of main memory, you can block for it by using the parameters below, for example, **-LNO:cs3=128M:ls3=...** . In this case, **assoc3** is

ignored and doesn't have to be specifi ed. Instead, you must specify**is_mem3..**, since virtual memory is fully associative.

Table 4-7 lists LNO options that control cache parameters.

Table 4-7 LNO Options to Control Cache Parameters

Option	Description
-LNO:{cache_size1,cs1}=n -LNO:{cache_size2,cs2}=n -LNO:{cache_size3,cs3}=n -LNO:{cache_size4,cs4}=n	The size of the cache. The value *n* can either be 0, or it must be a positive integer followed by only one of the letters **k**, **K**, **m**, or **M**. This specifi es the cache size in kilobytes or megabytes. A value of zero indicates that no cache exists at that level.
-LNO:{line_size1,ls1}=n -LNO:{line_size2,ls2}=n -LNO:{line_size3,ls3}=n -LNO:{line_size4,ls4}=n	The line size in bytes. This is the number of bytes that are moved from the memory hierarchy level further out to this level on a miss. A value of zero indicates that no cache exists at that level.
-LNO:{associativity1,assoc1}=n -LNO:{associativity2,assoc2}=n -LNO:{associativity3,assoc3}=n -LNO:{associativity4,assoc4}=n	The cache set associativity. Large values are equivalent. For example, when blocking for main memory, it's adequate to set **assoc3=128**. A value of zero indicates that no cache exists at that level.
-LNO:{miss_penalty1,mp1}=n -LNO:{miss_penalty2,mp2}=n -LNO:{miss_penalty3,mp3}=n -LNO:{miss_penalty4,mp4}=n	In processor cycles, the time for a miss to the next outer level of the memory hierarchy. This number is approximate, since it depends on a clean or dirty line, read or write miss, etc. A value of zero indicates that no cache exists at that level.
-LNO:{is_memory_leve1,is_mem1}={on,of f} -LNO:{is_memory_leve2,is_mem2}={on,of f} -LNO:{is_memory_leve3,is_mem3}={on,of f} -LNO:{is_memory_leve4,is_mem4}={on,of f}	Optional; the default is **off**. If specifi ed, the corresponding associativity is ignored and needn't be specifi ed. Model this memory hierarchy level as a memory, not a cache. This means that blocking may be attempted for this memory hierarchy level, and that blocking appropriate for a memory rather than cache will be applied (for example, no prefetching, and no concern about confl ict misses).

Controlling Blocking and Permutation Transformations

Table 4-8 lists an option that aides in modeling. The default depends on the target processor.

Table 4-8 LNO Option to Control Modeling

Option	Description
-LNO:{non_blocking_loads,nbl}=n1	Optional; specify FALSE if the processor blocks on loads. If not set, takes the default of the current processor (not associated with a cache level).

Table 4-9 lists options that control transformations.

Table 4-9 LNO Options to Control Transformations

Option	Description
-LNO:inter change={ON,OFF}	Specify OFF to disable the interchange transformation. The default is ON.
-LNO:blocking={ON,OFF}	Specify OFF to disable the cache blocking transformation. Note that loop interchange to improve cache performance can still be applied. The default is ON.
-LNO:blocking_size=[n1][,n2]	Specifi es a blocksize that the compiler must use when performing any blocking. No default exists.
-LNO:outer_unr oll=n -LNO:ou=n -LNO:outer_unr oll_max=n[no default] -LNO:ou_max=n -LNO:outer_unr oll_prod_max=n	Specifi es how far to unroll outer loops. **outer_unroll** (**ou**) indicates that every outer loop for which unrolling is valid should be unrolled by exactly *n*. The compiler either unrolls by this amount or not at all. No default exists. If you specify **outer_unroll**, neither outer_unroll_max nor outer_unroll_prod_max can be specifi ed. **outer_unroll_max** tells the compiler to unroll as many as *n* per loop, but no more. **outer_unroll_prod_max** indicates that the product of unrolling of the various outer loops in a given loop nest is not to exceed outer_unroll_prod_max. The default is 16.

Table 4-10 lists the LNO option that indicates how hard to try to optimize for the cache.

Table 4-10 LNO Option to Control Cache Optimization

Option	Description
–LNO:optimize_cache=n	**0** Does not model any cache. **1** Models square blocks (fast). **2** Models rectangular blocks (the default).

Table 4-11 lists the LNO option to control illegal transformation.

Table 4-11 LNO Option to Control Illegal Transformation

Option	Description
−LNO:apply_illegal_transformation_directives ={on,off}	Issues a warning if the compiler sees a directive to perform a transformation that it considers illegal. **on** May attempt to perform the transformation. **off** Does not attempt to perform the transformation.

Controlling Prefetch

Table 4-12 lists LNO options that control prefetch operations.

Table 4-12 LNO Options to Control Prefetch

Option	Description
–LNO:prefetch=[0,1,2]	Enables or disables prefetching. **0** Disables prefetch. **1** Enables prefetch but is conservative. **2** Enables prefetch and is aggressive. The default is enabled and conservative for the R5000/R10000, and disabled for all previous processors.
–LNO:prefetch_ahead=[*n*]	Prefetches the specifi ed number of cache lines ahead of the reference. The default is **2**.

Table 4-12 (continued) LNO Options to Control Prefetch

Option	Description
–LNO:prefetch_leveln=[on,off] –LNO:pfn=[on,off]	Selectively enables/disables prefetching for cache level *n* where *n* ranges from [1..4].
–LNO:prefetch_manual=[on,off]	Ignores or respects manual prefetches (through pragmas). **on** Respects manual prefetches (the default for R10000). **off** Ignores manual prefetches (the default for all processors except R10000).

Dependence Analysis

Table 4-13 lists options that control dependence analysis.

Table 4-13 Options to Control Dependence Analysis

Option	Description
–OPT:cray_ivdep={false/true}	Interprets any **ivdep** pragma using Cray semantics. The default is **false**. See "Pragmas and Directives for LNO" for a definition.
–OPT:liberal_ivdep={false/true}	Interprets any **ivdep** pragma using liberal semantics. The default is **false**. See "Pragmas and Directives for LNO" for a definition.

Pragmas and Directives for LNO

Fortran *directives* and C and C++ *pragmas* enable, disable, or modify a feature of the compiler. This section uses the term *pragma* when describing either a pragma or a directive.

Pragmas within a procedure apply only to that particular procedure, and revert to the default values upon the end of the procedure. Pragmas that occur outside of a procedure alter the default value, and therefore apply to the rest of the file from that point on, until overridden by a subsequent pragma.

By default, pragmas within a file override the command-line options. Use the **–LNO:ignore_pragmas** option to allow command-line options to override the pragmas in the file.

This section covers:

- " Fission/Fusion"
- " Blocking and Permutation Transformations"
- " Prefetch"
- " Dependence Analysis"

Fission/Fusion

The following pragmas/directives control fission and fusion.

C*$* AGGRESSIVE INNER LOOP FISSION

#pragma aggressive inner loop fission
: Fission inner loops into as many loops as possible. It can only be followed by a inner loop and has no effect if that loop is not inner any more after loop interchange.

C*$* FISSION [(n)]

#pragma fission [(n)]

C*$* FISSIONABLE

#pragma fissionable
: Fission the enclosing *n* level of loops after this pragma. The default is 1. Performs validity test unless a FISSIONABLE pragma is also specified. Does not reorder statements.

C*$* FUSE [(n,level)]

#pragma fuse [(n,level)]

C*$* FUSABLE

#pragma fusable
: Fuse the following *n* loops, which must be immediately adjacent. The default is **2,level**. Fusion is attempted on each pair of adjacent loops and the level, by default, is the determined by the maximal perfectly nested loop levels of the fused loops although partial fusion is allowed. Iterations may be peeled as needed during fusion and the limit of this

peeling is 5 or the number specified by the **-LNO:fusion_peeling_limit** option. No fusion is done for non-adjacent outer loops. When the FUSABLE pragma is present, no validity test is done and the fusion is done up to the maximal common levels.

C*$* NO FISSION

#pragma no fission

The loop following this pragma should not be fissioned. Its innermost loops, however, are allowed to be fissioned.

C*$* NO FUSION

#pragma no fusion

The loop following this pragma should not be fused with other loops.

Blocking and Permutation Transformations

The following pragmas/directives control blocking and permutation transformations.

Note: The parallelizing preprocessor may decide to do some transformation for parallelism that violates some of these pragmas.

C*$* INTERCHANGE (I, J, K)

#pragma interchange (i,j,k)

Loops I, J and K (in any order) must directly follow this pragma and be perfectly nested one inside the other. If they are not perfectly nested, the compiler may choose to perform loop distribution to make them so, or may choose to ignore the annotation, or even apply imperfect interchange. Attempts to reorder loops so that I is outermost, then J, then K. The compiler may choose to ignore this pragma.

C*$* NO INTERCHANGE

#pragma no interchange

Prevent the compiler from involving the loop directly following this pragma in an interchange, or any loop nested within this loop.

C*$* BLOCKING SIZE [(n1,n2)]

#pragma blocking size (n1,n2)

The loop specifi ed, if it is involved in a blocking for the primary (secondary) cache, will have a blocksize of n1 {n2}. The compiler tries to include this loop within such a block. If a 0 blocking size is specifi ed, then the loop is not stripped, but the entire loop is inside the block.

For example:

```
      subroutine amat(x,y,z,n,m,mm)
      real*8 x(100,100), y(100,100), z(100,100)
      do k = 1, n
    C*$* BLOCKING SIZE 20
         do j = 1, m
    C*$* BLOCKING SIZE 20
           do i = 1, mm
             z(i,k) = z(i,k) + x(i,j)*y(j,k)
           enddo
         enddo
       enddo
       end
```

In this example, the compiler makes 20x20 blocks when blocking. However, the compiler can block the loop nest such that loop `k` is not included in the tile. If it didn't, add the following pragma just before the `k` loop.

```
C*$* BLOCKING SIZE (0)
```

This pragma suggests that the compiler generates a nest like:

```
subroutine amat(x,y,z,n,m,mm)
 real*8 x(100,100), y(100,100), z(100,100)
 do jj = 1, m, 20
   do ii = 1, mm, 20
     do k = 1, n
       do j = jj, MIN(m, jj+19)
         do i = ii, MIN(mm, ii+19)
           z(i,k) = z(i,k) + x(i,j)*y(j,k)
         enddo
      enddo
     enddo
   enddo
 enddo
 end
```

Finally, you can apply a INTERCHANGE pragma to the same nest as a BLOCKING SIZE pragma. The BLOCKING SIZE applies to the loop it directly precedes only, and moves with that loop when an interchange is applied.

C*$* NO BLOCKING

#pragma no blocking

Prevent the compiler from involving this loop in cache blocking.

C*$* UNROLL (n)

#pragma unroll (n)

This pragma suggests to the compiler that *n*-1 copies of the loop body be added to the inner loop. If the loop that this pragma directly precedes is an inner loop, then it indicates standard unrolling. If the loop that this pragma directly precedes is not innermost, then outer loop unrolling (unroll and jam) is performed. The value of *n* must be at least 1. If it is exactly 1, then no unrolling is performed.

For example, the following code:

```
C*$* UNROLL (2)
   DO i = 1, 10
    DO j = 1, 10
     a(i,j) = a(i,j) + b(i,j)
    END DO
   END DO
```

becomes:

```
DO i = 1, 10, 2
  DO j = 1, 10
   a(i,j) = a(i,j) + b(i,j)
   a(i+1,j) = a(i+1,j) + b(i+1,j)
  END DO
 END DO
```

and not:

```
DO i = 1, 10, 2
  DO j = 1, 10
   a(i,j) = a(i,j) + b(i,j)
  END DO
```

```
  DO j = 1, 10
   a(i+1,j) = a(i+1,j) + b(i+1,j)
  END DO
 END DO
```

The UNROLL pragma again is attached to the given loop, so that if an INTERCHANGE is performed, the corresponding loop is still unrolled. That is, the example above is equivalent to:

```
C*$* INTERCHANGE i,j
   DO j = 1, 10
C*$* UNROLL 2
    DO i = 1, 10
     a(i,j) = a(i,j) + b(i,j)
    END DO
   END DO
```

C*$* BLOCKABLE(I,J,K)

#pragma blockable (i,j,k)

The loops I, J and K must be adjacent and nested within each other, although not necessarily perfectly nested. This pragma informs the compiler that these loops may validly be involved in a blocking with each other, even if the compiler considers such a transformation invalid. The loops are also interchangeable and unrollable. This pragma does not tell the compiler which of these transformations to apply.

Prefetch

The following pragmas/directives control prefetch operations.

C*$* PREFETCH [(n1,n2)]

#pragma prefetch [(n1,n2)]

Specify prefetching for each level of the cache. Scope: entire function containing the pragma.

0 prefetching off (default for all processors except R10000)
1 prefetching on, but conservative (default at **-O3** when prefetch is on)
2 prefetching on, and aggressive

C*$* PREFETCH_MANUAL[(n)]

#pragma prefetch_manual[(n)]

Specify whether manual prefetches (through pragmas) should be respected or ignored. Scope: entire function containing the pragma.
0 ignore manual prefetches (default for all processors except R10000)
1 respect manual prefetches (default at **-O3** for R10000 and beyond)

C*$* PREFETCH_REF=[array-ref], stride=[str][,str], level=[lev][,lev],
& kind=[rd/wr], size=[sz]

#pragma prefetch_ref =[array-ref], stride=[str][,str], level=[lev][,lev],
& kind=[rd/wr], size=[sz]

array-ref specifies the reference itself, for example, A(i, j). Must be specified.

stride prefetches every stride iterations of this loop. Optional, the default is 1.

level specifies the level in memory hierarchy to prefetch. Optional, the default is 2.
lev=1: prefetch from L2 to L1 cache.
lev=2: prefetch from memory to L1 cache.

kind specifies read/write. Optional, the default is **write**.

size specifies the size (in Kbytes) of this array referenced in this loop.
Must be a constant. Optional.

The effect of this pragma is:

- generate a prefetch and connect to the specified reference (if possible).
- search for array references that match the supplied reference in the current loop-nest. If such a reference is found, then that reference is connected to this prefetch node with the specified latency. If no such reference is found, then this prefetch node stays free-floating and is scheduled "loosely."
- ignore all references to this array in this loop-nest by the automatic prefetcher (if enabled).
- if the size is specified, then the auto-prefetcher (if enabled) uses that number in its volume analysis for this array.

No scope, just generate a prefetch.

C*$* PREFETCH_REF_DISABLE=A, size=[num]

#pragma prefetch_ref_disable=A,size=[num]

size specifi es the size (in Kbytes) of this array referenced in this loop (optional). The *size* must be a constant. This explicitly disables prefetching all references to array A in the current loop nest. The auto-prefetcher runs (if enabled) ignoring array A. The *size* is used for volume analysis.
Scope: entire function containing the pragma.

Dependence Analysis

The following pragmas/directives control dependence analysis.

CDIR$ IVDEP

#pragma ivdep

Liberalize dependence analysis. This applies only to inner loops. Given two memory references, where at least one is loop variant, ignore any loop-carried dependences between the two references.

For example:

```
 CDIR$ IVDEP
   do i = 1,n
      b(k) = b(k) + a(i)
   enddo
```

ivdep does not break the dependence since `b(k)` is not loop variant.

```
CDIR$ IVDEP
   do i=1,n
      a(i) = a(i-1) + 3.0
   enddo
```

ivdep does break the dependence, but the compiler warns the user that it's breaking an obvious dependence.

```
CDIR$ IVDEP
   do i=1,n
      a(b(i)) = a(b(i)) + 3.0
   enddo
```

ivdep does break the dependence.

```
CDIR$ IVDEP
  do i = 1,n
     a(i) = b(i)
     c(i) = a(i) + 3.0
  enddo
```

ivdep does not break the dependence on `a(i)` since it is within an iteration.

If **-OPT:cray_ivdep=TRUE**, use Cray semantics. Break all lexically backwards dependences. For example:

```
CDIR$ IVDEP
  do i=1,n
     a(i) = a(i-1) + 3.0
  enddo
```

ivdep does break the dependence but the compiler warns the user that it's breaking an obvious dependence.

```
CDIR$ IVDEP
  do i=1,n
     a(i) = a(i+1) + 3.0
  enddo
```

ivdep does not break the dependence since the dependence is from the load to the store, and the load comes lexically before the store.

To break all dependencies, specify **-OPT:liberal_ivdep=TRUE**.

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 (described below) are the preferred controls:

- "–OPT:roundoff=n" deals with the extent to which language expression evaluation rules are observed, generally affecting the transformation of expressions involving multiple operations.
- "–OPT:IEEE_arithmetic=n" 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).

–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 mips4 **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.

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 **–mips4**, 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 mips4 reciprocal instruction for `1.0/y`, normally enabled by **IEEE_arithmetic>=2**. 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>=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 With the –OPT Option

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

- "Using the –OPT:space Option"
- "Using the –OPT:Olimit=n Option"
- "Using the –OPT:alias Option"
- "Simplifying Code With the –OPT Option"

Using the –OPT:space Option

–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.

Using the –OPT:Olimit=n Option

–OPT:Olimit=*n*

This option controls the size of procedures to be optimized. Procedures above the cut-off limit are not optimized. A value of 0 means "infinite Olimit," and causes all procedures to be optimized. If you compile at **–O2** or above, and a routine is so large that the compile speed may be slow, then the compiler prints a message telling you the **Olimit** value needed to optimize your program.

Using the –OPT:alias Option

–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. The **–OPT: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.

Simplifying Code With the –OPT Option

The following **–OPT** options perform algebraic simplifications of expressions, such as turning `x + 0` into `x`.

–OPT:div_split

Simplifies expressions by determining if `(A/B)` should be turned into `(1/B)*A`. This can be useful if `B` is a loop-invariant, as it replaces the divide with a multiply. For example, `X = A/B` becomes `X = A*(1/B)`.

–OPT:fold_reassociate

Determines if optimization involving reassociation of floating point quantities is allowed. This option is on at **–O3**, or if **roundoff >= 2**. For example, `X + 1. X` can be turned into `X - X + 1.0`, which will then simplify to `1`. This can cause problems is `X` is large compared to `1`, so that `X+1` is `X` due to roundoff.

–OPT:fold_unsafe_relops

Controls folding of relational operators in the presence of possible integer overflow On by default. For example, `X + Y < 0` may turn into `X < Y`. If `X + Y` overflows, it is possible to get different answers.

–OPT:fold_unsigned_relops

Determines if simplifications are performed of unsigned relational operations that may result in wrong answers in the event of integer overflow Off by default. The example is the same as above, only for unsigned integers.

-OPT:recip Allows the generation of the mips4 **recip** instruction. On at **-O3** or **IEEE_Arithmetic >= 2**. For example, `X = 1./Y` generates the **recip** instruction instead of a divide instruction. This may change the results slightly.

-OPT:rsqrt Allows the generation of the mips4 **rsqrt** instruction. On at **-O3** or **IEEE_Arithmetic >= 2**. For example, `X = 1./SQRT(Y)` generates the **rsqrt** instruction instead of a divide and a square root. This may change the results slightly.

The Code Generator

This section describes the code generator and covers the following topics:

- "Overview of the Code Generator"
- "Code Generator and Optimization Levels -02 and -03"
- "Modifying Code Generator Defaults"
- "Miscellaneous Code Generator Performance Topics"

Overview of the Code Generator

The Code Generator (CG) processes an input program unit (PU) in intermediate representation form to produce an output object file (*.o*) and/or assembly file (*.s*).

Program units are partitioned into basic blocks (BBs). A new BB is started at each branch target. BBs are also ended by CALLs or branches. Large BBs are arbitrarily ended after a certain number of OPs, because some algorithms in CG work on one BB at a time ("local" algorithms) and have a complexity that is nonlinear in the number of operations in the BB.

This section covers the following topics:

- "Code Generator and Optimization Levels"
- "An Example of Local Optimization for Fortran"

Code Generator and Optimization Levels

At optimization levels **-O0** and **-O1**, the CG only uses local algorithms that operate individually on each BB. At **-O0**, no optimization is done. References to global objects are spilled and restored from memory at BB boundaries. At **-O1**, CG performs standard local optimizations on each BB (for example, copy propagation, dead code elimination) as well as some elimination of useless memory operations.

At optimization levels **-O2** and **-O3**, CG includes global register allocation and a large number of special optimizations for innermost loops, including software pipelining at **-O3**.

An Example of Local Optimization for Fortran

Consider the Fortran statement, `a(i) = b(i)`. At **-O0**, the value of `i` is kept in memory and is loaded before each use. This statement uses two loads of `i`. The CG local optimizer replaces the second load of `i` with a copy of the first load, and then it uses copy-propagation and dead code removal to eliminate the copy. Comparing *.s* files for the **-O0** and **-O1** versions shows:

The *.s* file for **-O0**:

```
lw $3,20($sp)                  #  load address of i
lw $3,0($3)                    #  load i
addiu $3,$3,-1                 #  i - 1
sll $3,$3,3                    #  8 * (i-1)
lw $4,12($sp)                  #  load base address for b
addu $3,$3,$4                  #  address for b(i)
ldc1 $f0,0($3)                 #  load b
lw $1,20($sp)                  #  load address of i
lw $1,0($1)                    #  load i
addiu $1,$1,-1                 #  i - 1
sll $1,$1,3                    #  8 * (i-1)
lw $2,4($sp)                   #  load base address for a
addu $1,$1,$2                  #  address for a(i)
sdc1 $f0,0($1)                 #  store a
```

The *.s* file for **-O1**:

```
lw $1,0($6)                    #  load i
lw $4,12($sp)                  #  load base address for b
addiu $3,$1,-1                 #  i - 1
sll $3,$3,3                    #  8 * (i-1)
```

```
lw $2,4($sp)                    #  load base address for a
addu $3,$3,$4                   #  address for b(i)
addiu $1,$1,-1                  #  i - 1
ldc1 $f0,0($3)                  #  load b
sll $1,$1,3                     #  8 * (i-1)
addu $1,$1,$2                   #  address for a(i)
sdc1 $f0,0($1)                  #  store a
```

The *.s* file for **–O2** (using OPT to perform scalar optimization) produces optimized code:

```
lw $1,0($6)                     # load i
sll $1,$1,3                     # 8 * i
addu $2,$1,$5                   # address of b(i+1)
ldc1 $f0,-8($2)                 # load b(i)
addu $1,$1,$4                   # address of a(i+1)
sdc1 $f0,-8($1)                 # store a(i)
```

Code Generator and Optimization Levels –02 and –03

This section provides additional information about the **–O2** and **–O3** optimization levels. Topics include:

- "If Conversion"
- "Cross-Iteration Optimizations"
- "Loop Unrolling"
- "Recurrence Breaking"
- "Software Pipelining"
- "Steps Performed By the Code Generator at Levels –O2 and –O3"

If Conversion

If conversion is a transformation that converts control-flow into conditional assignments. For example, consider the following code before *if* conversion. Note that `expr1` and `expr2` are arbitrary expressions without calls.

```
if ( cond )
    a = expr1;
else
    a = expr2;
```

After *if* conversion, the code looks like this:

```
tmp1 = expr1;
tmp2 = expr2;
a = (cond) ? tmp1 : tmp2;
```

Benefits of *if* Conversion

Performing *if* conversion:

- **Exposes more instruction-level parallelism.** This is almost always valuable on hardware platforms such as R10000.
- **Eliminates branches.** Some platforms (for example, the R10000) have a penalty for taken branches. There can be substantial costs associated with branches that are not correctly predicted by branch prediction hardware. For example, a mispredicted branch on R10000 has an average cost of about 8 cycles.
- **Enables other compiler optimizations.** Currently, cross-iteration optimizations and software pipelining both require single basic block loops. *If* conversion changes multiple BB innermost loops into single BB innermost loops.

Interaction of *if* Conversion With Floating Point and Memory Exceptions

In the code above that was *if converted*, the expressions, `expr1` and `expr2`, are UNCONDITIONALLY evaluated. This can conceivably result in generation of exceptions that do not occur without *if conversion*. An operation that is conditionalized in the source, but is unconditionally executed in the object, is called a speculated operation. Even if the **–TENV:X** level prohibits speculating an operation, it may be possible to *if convert*. For information about the **–TENV** option, see "Controlling the Target Environment" and the appropriate compiler reference page.

For example, suppose `expr1 = x + y;` is a floating point add, and `X=1`. Speculating fl ops is not allowed (to avoid false overflow exceptions). Define `x_safe` and `y_safe` by `x_safe = (cond)? x : 1.0; y_safe = (cond) ? y : 1.0;`. Then unconditionally evaluating `tmp1 = x_safe + y_safe;` cannot generate any spurious exception. Similarly, if `X < 4`, and `expr1` contains a load (for example, `expr1 = *p`), it is illegal to speculate the dereference of `p`. But, define `p_safe = (cond) ? p : known_safe_address;` and then `tmp1 = *p_safe;` cannot generate a spurious memory exception.

Notice that with **–TENV:X=2**, it is legal to speculate fl ops, but not legal to speculate memory references. So `expr1 = *p + y;` can be speculated to `tmp1 = *p_safe + y;`. If `*known_safe_address` is uninitialized, there can be spurious floating point exceptions

associated with this code. In particular, on some MIPS platforms (for example, R10000) if the input to a fl op is a denormalized number, then a trap will occur. Therefore, by default, CG initializes ***known_safe_address** to 1.0.

Cross-Iteration Optimizations

Four main types of cross-iteration optimizations include:

- " Read-Read Elimination"
- " Read-Write Elimination"
- " Write-Write Elimination"
- " Common Sub-expression Elimination"

Read-Read Elimination

Consider the example below:

```
DO i = 1,n
   B(i) = A(i+1) - A(i)
END DO
```

The load of `A(i+1)` in iteration `i` can be reused (in the role of `A(i)`) in iteration `i+1`. This reduces the memory requirements of the loop from 3 references per iteration to 2 references per iteration.

Read-Write Elimination

Sometimes a write in one iteration is read in a later iteration. For example:

```
DO i = 1,n
   B(i+1) = A(i+1) - A(i)
   C(i) = B(i)
END DO
```

In this example, the load of `B(i)` can be eliminated by reusing the value that was stored to `B` in the previous iteration.

Write-Write Elimination

Consider the example below:

```
DO i = 1,n
   B(i+1) = A(i+1) - A(i)
   B(i) = C(i) - B(i)
END DO
```

Each element of `B` is written twice. Only the second write is required, assuming read-write elimination is done.

Common Sub-expression Elimination

Consider the example below:

```
DO i = 1,n
   B(i) = A(i+1) - A(i)
   C(i) = A(i+2) - A(i+1)
END DO
```

The value computed for `C` in iteration `i` may be used for `B` in iteration `i+1`. Thus only one subtract per iteration is required.

Loop Unrolling

In this example, unrolling 4 times converts this code:

```
for( i = 0; i < n; i++ ) {
   a[i] = b[i];
}
```

After unrolling, the code looks like this:

```
for ( i = 0; i < (n % 4); i++ ) {
   a[i] = b[i];
}
for ( j = 0; j < (n / 4); j++ ) {
   a[i+0] = b[i+0];
   a[i+1] = b[i+1];
   a[i+2] = b[i+2];
   a[i+3] = b[i+3];
   i += 4;
}
```

Loop unrolling:

- Exposes more instruction-level parallelism. This may be valuable even on execution platforms such as R10000.
- Eliminates branches.
- Amortizes loop overhead. For example, unrolling replaces four increments `i+=1` with one increment `i+=4`.

Recurrence Breaking

There are two types of recurrence breaking:

- **Reduction interleaving.** For example, interleaving by 4 transforms this code:

  ```
  sum = 0
  DO i = 1,n
     sum = sum + A(i)
  END DO
  ```

  After reduction interleaving, the code looks like this (omitting the cleanup code):

  ```
  sum1 = 0
  sum2 = 0
  sum3 = 0
  sum4 = 0
  DO i = 1,n,4
     sum1 = sum1 + A(i+0)
     sum2 = sum2 + A(i+1)
     sum3 = sum3 + A(i+2)
     sum4 = sum4 + A(i+3)
  END DO
  sum = sum1 + sum2 + sum3 + sum4
  ```

- **Back substitution.** For example:

  ```
  DO i = 1,n
     B(i+1) = B(i) + k
  END DO
  ```

  The code is converted to:

  ```
  k2 = k + k
  B(2) = B(1) + k
  DO i = 2,n
     B(i+1) = B(i-1) + k2
  END DO
  ```

Software Pipelining

Software pipelining (SWP) schedules innermost loops to keep the hardware pipeline full. For information about software pipelining, see *MIPSpro 64-Bit Porting and Transition Guide*, Chapter 6, "Performance Tuning." Also, for information on instruction level parallelism, refer to B.R.Rau and J.A.Fisher, "Instruction Level Parallelism," Kluwer Academic Publishers, 1993 (reprinted from the Journal of Supercomputing, Volume 7, Number 1/2).

Steps Performed By the Code Generator at Levels -O2 and -O3

The steps performed by the code generator at **-O2** and **-O3** include:

1. Non-loop *if* conversion.
2. Find innermost loop candidates for further optimization.
   Loops are rejected for any of the following reasons:
   - marked UNIMPORTANT (for example, LNO cleanup loop)
   - strange control flow (for example, branch into the middle)
3. *If* convert. This transforms a multi-BB loop into a single "BB" containing OPs with "guards." If a loop has multiple BBs and is not if converted it is not a candidate for any further optimization) If conversion can fail for any of the following reasons:
   - loop contains a CALL
   - loop has too many BBs
   - loop has too many instructions
4. Perform cross-iteration optimizations (these optimizations are only attempted for trip-count loops).
5. Unroll (only done for trip count loops; may unroll fully so that the loop is no longer a loop)
6. Fix recurrences.
7. If still a loop, and there is a trip count, and **-O3**, invoke software pipelining (SWP).
8. If not software pipelined, reverse *if* convert.
9. Reorder BBs to minimize (dynamically) the number of taken branches. Also eliminate branches to branches when possible, and remove unreachable BBs. This step also happens at **-O1**.

At several points in this process local optimizations are performed, since many of the transformations performed can expose additional opportunities. It is also important to note that many transformations require legality checks that depend on alias information. There are three sources of alias information:

- At **-O3** , the loop nest optimizer, LNO, provides a dependence graph for each innermost loop.
- The scalar optimizer provides information on aliasing at the level of symbols. That is, it can tell whether arrays A and B are independent, but it does not have information about the relationship of different references to a single array.
- CG can sometimes tell that two memory references are identical or distinct. For example, if two references use the same address, and there are no definitions of that address between the two references, then the two references are identical.

Modifying Code Generator Defaults

CG makes many choices, for example, what conditional constructs to *if* convert, or how much to unroll a loop. In most cases, the compiler makes reasonable decisions. Occasionally, however, you can improve performance by modifying the default behavior.

You can control the code generator by:

- Increasing or decreasing the unroll amount.

  A heuristic is controlled by **-OPT :unroll_analysis** (on by default), which generally tries to minimize unrolling. Less unrolling leads to smaller code size and faster compilation. You can change the upper bound for the amount of unrolling with **-OPT :unroll_times** (default is 8) or **-OPT :unroll_size** (the number of instructions in the unrolled body, current default is 80).
- Disabling software pipelining with **-OPT :swp=off**.

  As far as CG is concerned, **-O3 -OPT :swp=off** is the same as **-O2** . Since OPT does not run at **-O2** , however, the input to CG can be very different, and the available aliasing information can be very different.

Miscellaneous Code Generator Performance Topics

This section explains a few miscellaneous topics including:

- "Prefetch and Load Latency"
- "Frequency and Feedback"

Prefetch and Load Latency

At **-O3**, with **-r10000**, LNO generates prefetches for memory references that are likely to miss either the L1 or the L2 cache. By default, CG generates prefetch operations for L2 prefetches, but discards L1 prefetches. It also makes sure that loads that had associated L1 prefetches are issued at least 8 cycles before their results are used.

You can adjust all these default behaviors, but in most cases such fine control is not necessary. In general, two common situations exist:

- Prefetch is a good thing, and the default settings of the detailed controls are reasonable (the most common situation).
- Performance improves when all prefetches are disabled. The best way to do this is with **-LNO:prefetch=0**.

There is a possibly confusing interaction between prefetching and SWP notes in the *.s* file. Typically several replications of a software pipelined loop that differ only in the registers used for corresponding values. (This is necessary because values may have to survive in registers across multiple iterations of the loop.)

It is often possible to reduce prefetch overhead by eliminating some of the corresponding prefetches from different replications. For example, suppose a prefetch is only required on every 4th iteration of a loop, because 4 consecutive iterations will load from the same cache line. If the loop is replicated 4 times by SWP, then there is no need for a prefetch in each replication, so 3 of the 4 corresponding prefetches are pruned away.

The original SWP schedule has room for a prefetch in each replication, and the number of cycles for this schedule is what is described in the SWP notes as "cycles per iteration." The number of memory references listed in the SWP notes ("mem refs") is the number of memory references including prefetches in Replication 0. If some of the prefetches have been pruned away from replication 0, the notes will overstate the number of cycles per iteration while understating the number of memory references per iteration.

Frequency and Feedback

Some choices that CG makes are decided based on information about the frequency with which different BBs are executed. By default, CG makes guesses about these frequencies based on the program structure. This information is available in the *.s* file. The frequency printed for each block is the predicted number of times that block will be executed each time the PU is entered.

If compilation uses a feedback file, the frequency guesses are replaced with the measured frequencies derived from the feedback data. Currently feedback information guides control flow optimizations, global spill and restore placement, instruction alignment, and delay slot filling. Average loop trip-counts can be derived from feedback information. Trip count estimates are used to guide decisions about how much to unroll and whether or not to software pipelining.

Controlling the Target Architecture

Some options control the target architecture for which code is generated. The options are described below.

–32, **–n32**, or **–64** This option determines the base ABI to be assumed, either the 32-bit ABI for mips1/2 targets, or the new 32-bit (n32) and 64-bit ABI for mips3/4 targets.

–mips [1 2 3 4]
: This option identifies the instruction set architecture (ISA) to be used for generated code. It defaults to **mips2** for 32-bit compilations, **mips3** for n32 compilations, and **mips4** for 64-bit compilations.

–TARG:madd[=(**ON**|**OFF**)]
: This option enables/disables use of the multiply/add instructions for mips4 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 mips4 targets; otherwise, it is ignored.

-r[5000 8000 10000]

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 -mips3 -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.

Executing 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, from 0 to 5, 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, traps may be disabled or ignored. The default is 1 at **-O0** through **-O2** and 2 at **-O3** .

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.

Improving Global Optimization

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

- "Overview of the Global Optimizer"
- "Optimizing C, 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, C++, and Fortran Programs

The following suggestion applies to C, 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 and C++ Programs

The following suggestions apply to C and C++ programs.

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

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 avoid global variables. In C and 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.

Const parameters. Declare pointer parameters as **const** in prototypes whenever possible, that is, when there is no path through the routine that modifies the pointee. This allows the compiler to avoid some of the negative assumptions normally required for pointer and reference parameters (see below).

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. For example:

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

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

You can increase the effi ciency 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` effi cientlyeven though a pointer is not specifi ed in the source code. For example:

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

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 (and 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, C++, and Fortran Programs

The following suggestions apply to both C, 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 32-bit or 64-bit scalar variables instead of smaller ones. This practice can take more data space, but produces more efficient code.

C and C++ Programs

The following suggestions apply to C and C++ programs:

- 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 (or with **long long int**s in either 32-bit or 64-bit code). 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;
```

C++ Programs Only

The following suggestions apply to C++ programs:

- Inline whenever possible. Inline calls inside loops since functions calls that are not inlined prevent loop-nest optimizations and software pipelining.
- Use pre-increment (for example, increment `i` as `++i`) instead of post-increment (`i++`) to avoid generating copy.
- Avoid using virtual functions. The penalty is in method lookup and the inability to inline them.
- Pass by value rather than const ref (to ameliorate aliasing problems).
- Avoid the creation of temporaries, that is, `Aa = 1` is better than `Aa = A(1)`.
- If your code does not use exception handing, use **-LANG:exceptions=off** when compiling.

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.

Using SpeedShop

SpeedShop is an integrated package of performance tools that you can use to gather performance data and generate reports. To record the experiments, use the ssrun(1) command, which runs the experiment and captures data from an executable (or instrumented version). You can examine the data using prof. Speedshop also lets you start a process and attach a debugger to it.

For detailed information about SpeedShop, *ssrun*, *prof*, and *pixie*, see the *SpeedShop User's Guide*. In particular, refer to:

- “Setting Up and Running Experiments: ssrun”
- “Analyzing Experiment Results: prof”