MIPSpro™ 64–Bit
Porting and Transition Guide

Chapter 1
System and Compiler Overview

This chapter gives a brief overview of the IRIX™ 6 operating system and describes the MIPSpro™ 32–bit and 64–bit compilers. It contains four sections:

- The first section lists the new features of IRIX 6 and describes its compatibility with IRIX 5.*x* (see Figure 1–1).
- The second section describes the various components of both the 64–bit compiler system and the 32–bit compiler.
- The third section describes the Kuck and Associates preprocessor (KAP), a major component of the 64–bit Fortran front end.
- The last section outlines the multiprocessing support provided by the MIPSpro compilers.

IRIX 6 Overview

The IRIX 6 release introduces support for both 64–bit virtual addressing and the MIPS R8000 processor. The R8000 processor has the following capabilities:

- Executes MIPS1 user code, compatible with the R3000.
- Executes MIPS2 instruction set extensions introduced in the R4000.
- Executes MIPS3 64–bit addressing and instructions introduced in the R4400.
- Integrates features for floating point performance improvements and integer improvements, which are called MIPS4 instructions.

The MIPS3 and MIPS4 64–bit capabilities provide both 64–bit virtual addressing and instructions which manipulate 64–bit integer data. Processor registers are 64 bits in size.

For more information about the R8000, please consult the *MIPS R8000 Microprocessor User's Manual*.

IRIX 6 Objectives

The objectives for the IRIX 6 release are:

- Enable intensive floating point programs to compile optimized R8000 code, usually Fortran, and to link to the libraries that they need.
- Supply optimized versions of our own intensive floating point codes, to optimize R8000 performance. Note that a utility program has no need to be ported to a 64–bit version, unless it needs to exploit floating point performance.
- Provide 64–bit user processes with a virtual address space of up to one terabyte (240) in size, provided that the size of physical memory combined with swap space is larger.
- Provide 32–bit user processes with compatibility to IRIX 5.*x*, and supply 32–bit binaries from a *partnered* IRIX 5.*x* release (the first being 5.2). The goal for the 32–bit libraries and utilities is to be compatible, in most cases identical, with the partnered IRIX 5.*x* release.

What is Not in the IRIX 6 Release

- The filesystem has no new features since IRIX 5.2. Maximum file size is 2 Gbytes and maximum filesystem size is 8 Gbytes.
- There is no check–point/restart support in IRIX 6.

Compatibility

With the IRIX 6 release on an R8000 system, you can execute any of the following:

- A program built under IRIX 4.0.5 (32–bit MIPS1 instructions)
- A program built under IRIX 5.2 or earlier (32–bit MIPS1 or MIPS2 instructions).

- A program built under IRIX 6 release (any of 32–bit MIPS1 or MIPS2 instructions, or 64–bit MIPS3 or 64–bit MIPS4 instructions).

Figure 1–1 illustrates in more detail how IRIX 6 is compatible with IRIX 5.*x*.

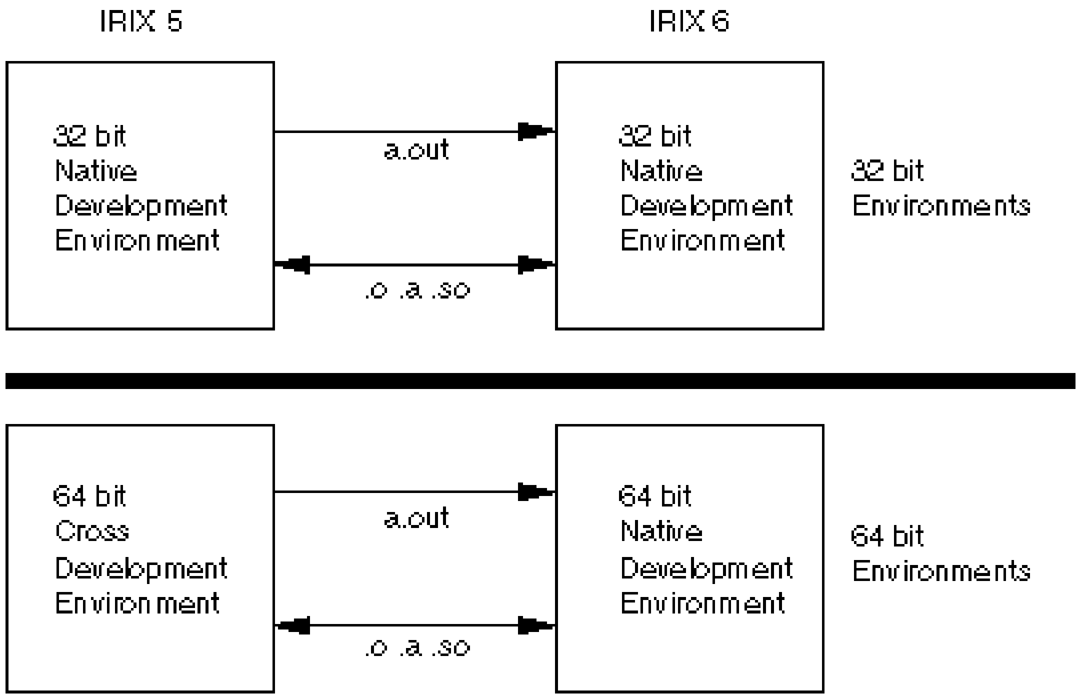

Figure 1–1 IRIX 5 and IRIX 6 Compatibility

- 32–bit IRIX 5.*x* binaries and Dynamic Shared Objects (DSOs) execute under IRIX 6.
- IRIX 6 has a set of compilers (32–bit Native Development Environment) that generate 32–bit code. You can mix objects and link with objects produced on IRIX 5.*x*. (We do not, however, guarantee that this code runs on current IRIX 5.*x* systems.)

  For the first release, this set of compilers is, in fact, the same *ucode* compilers shipped on current IRIX 5.*x* systems.
- IRIX 6 also has a set of compilers (64–bit Native Development Environment) that generates 64–bit code. This code **cannot** run on current IRIX 5.x systems.
- You can specify which compiler you want to run by using the *–64* or *–32* flags on the compiler command line.

  The compiler driver then executes the appropriate compiler binaries and links with the correct libraries. This also applies to the assembler, linker, and archiver. If these switches are not present, the driver checks an environment variable, *SGI_ABI*, for these values.

  The default under IRIX 6 is 64–bit operation.
- All of the compiler–related tools (*dbx*, *nm*, *dis*, *pixie*, *prof*) except *pixstats* can work with either 32–bit or 64–bit binaries. *pixstats* functionality for 64–bit programs is integrated into *prof*.
- You **cannot** mix objects and DSOs produced by the 32–bit compilers with objects and DSOs produced by the 64–bit compilers. In Figure 1–1, this is illustrated by the heavy line separating the 32–bit and 64–bit development environments.

  This means:
  - You **cannot** link 32–bit objects with 64–bit objects
  - You **cannot** link 64–bit objects with 32–bit objects
  - Your 32–bit program **cannot** use 64–bit shared libraries
  - Your 64–bit program **cannot** use 32–bit shared libraries
- The */usr/lib* directory on IRIX 6 contains the 32–bit libraries and *.so*s. The 64–bit *.so*s are located in */usr/lib64*. The complete layout looks like this:

32−bit: This is the IRIX 5.x */usr/lib*, including compiler components:

```
/usr/lib/
*.so
mips2/
*.so
/usr/lib/
cfe
fcom
ugen
uopt
as
```

64−bit: These are the 64−bit−specific components:

```
/usr/lib64/
*.so
mips3/
*.so
mips4/
*.so
/usr/lib64/cmplrs
be
fec
fef77
fef90
as
```

- A CROSS environment is provided on IRIX 5.*x* systems which allows you to develop code for IRIX 6 systems. This environment is a stripped−down version of IRIX 6 installed completely under */usr/cross64*. Tools in the cross development world behave just as though they were rooted to */*. On an IRIX 6 system, */usr/cross64* is a link to */*.

  Note: IRIX 6 device driver development is not supported by the CROSS environment.

- In cases where there are significant differences between the 32−bit and 64−bit interfaces, both reference pages (man pages) are supplied.

  For example, when you type *man cc*, both the 64−bit and 32−bit reference pages for *cc(1)* are shown, one right after the other.

Known Compatibility Issues

The following issues are known to cause troubles for 32−bit programs running on IRIX 6:

- Any access to kernel data structures, for example, through */dev/kmem*. Many of these structures have changed in size. Programs making these kinds of accesses must be ported to 64−bit. 32−bit programs cannot access all of kernel memory, hence probably must also be ported to 64−bit.
- Use of *nlist()* does not work on any 64−bit *.o* or *a.out*. A new *nlist64()* is supplied for 64−bit ELF.
- Any assumption that the page size is 4Kbytes (for example, using *mmap()* and specifying the address). The page size is no longer 4Kbytes. Programs must use *getpagesize()*.
- Ada programs which catch floating point exceptions do not work.
- Any program using */proc* must have some interfaces changed.

It is possible for a program to determine whether it is running on a 64−bit capable kernel in order to work around the isses listed above. Use *sysconf(_SC_KERN_POINTERS)*, which returns 32 or 64.

Source Level System Interface

The IRIX APIs, as defined in SVID and POSIX, are unchanged. Source code requires no changes to be built *−64*. The compiled bit sizes of arguments and fields may indeed be different between the 32−bit and 64−bit definitions, but source code should not have to be changed. For example, *off_t* and *size_t* are now 64 bits when compiled *−64* and 32 bits when compiled *−32*.

The source code interface to IRIX therefore remains the same, with a few exceptions. This interface is defined by the set of header files in */usr/include*. Most of these header files have not changed. The kernel maintains compatibility by converting all system calls from a 32−bit binary into 64−bit internal kernel format. The kernel maintains all states in a superset form (usually 64 bits), converting data structures upon entry to and exit from the kernel. As a result, the kernel continues to keep traditional IRIX form and state information of processes, files, and other structures.

Graphics Environment Support

Table 1−1 summarizes the graphics environments supported under IRIX 6. Only IRIX GL is not supported for 64−bit applications. All of the other graphics enivironments are provided for both 32−bit and 64−bit applications.

Table 1−1 IRIX 6 Graphics Environment Support

Package	32−bit Support	64−bit Support
IRIX GL (libgl)	Yes	No
OpenGL	Yes	Yes
X11R6	Yes	Yes
Motif1.2.4	Yes	Yes

Inter−Process Communication

Interfaces across a network connection usually allow 32−bit and 64−bit programs to communicate (for example, NFS works fine). Programs using RPC and XDR should have no problems. Of course, exchanging binary data with size assumptions which do not agree can fail.

Usually, there are no problems with having 32−bit and 64−bit programs interact via system calls. For instance, sending data through a pipe, or sharing file access, or connecting through the network, or shared memory usage, all work. In all cases, however, binary data structure sizes must agree.

Arenasare an exception (see *usinit(3)*). They are not compatible between 32−bit and 64−bit programs. Arenas can be used within a set of 32−bit programs, and within a set of 64−bit programs, but not across program types. See *test_and_set(3)* and *abilock(3)* for locks compatible with 32−bit and 64−bit programs.

Device Driver Interface

The IRIX 6 kernel requires MIPS3 or MIPS4 code. MIPS1 drivers cannot be linked. Additionally, the kernel is now linked in ELF format, so the *−coff* option to link 5.*x* drivers is no longer used.

The kernel/driver interface (DDI/DKI) has a few changes and some additions. Usually kernel drivers need a lot of rework due to internal assumptions of pointer sizes, page sizes, and I/O mapping sizes.

Socket *ioctl()* calls to the kernel by 32−bit programs are converted for socket *ioctls* automatically. Hence, whether a 32−bit or 64−bit program is doing the call makes no difference to the kernel.

Block and character drivers have to be responsible for their own *ioctls()*. The approach which IRIX drivers follow is to have a single header file for user code compilation, so the driver must convert fields according to differences between the 32−bit and 64−bit compilation models.

For streams, the driver will not know the user process type, so *ioctls* must be fixed binary data structures. They cannot vary between the 32−bit or 64−bit compilation models. You must fix your header files such that this is true.

Routines have been added to the device kernel interface (DKI) to aid *ioctl* conversions. Please refer to the *IRIX Device Driver Writer's Guide*.

Compiler System Components

As explained earlier, the MIPSpro compiler system on IRIX 6 consists of two independent compiler systems. This section describes and compares them.

Fortran

The MIPSpro Fortran 77 compilers support both 32−bit and 64−bit compiler modes. The components of the two systems are described in the section that follows.

Fortran 32−Bit System

The 32−bit (*ucode)* Fortran compiler systems contain the following components:

f77	Fortran driver: Executes the appropriate components below.
cpp	C preprocessor: Handles *#include* statements and other *cpp* constructs such as *#define*, *#ifdef*, and so on, in the source file.
fopt	Special scalar optimizer: Performs scalar optimization on the Fortran source.
pfa	Power Fortran Accelerator™: Automatically parallelizes Fortran source code and performs scalar optimizations. Available with the Power Fortran product.
fcom	Fortran front end: Parses the source file into intermediate code (*ucode*).
uopt	Optimizer: Performs optimizations on the intermediate file.
ugen	Code generator: Generates binary assembly code from the intermediate file.
as1	Binary assembler: Assembles the *binasm* code into an object file.
ld	Linker: Links the object file(s) with any libraries.

When you run simple examples through the *ucode* Fortran compilers, the following components are executed by the compiler driver:

```
%f77 −32 foo.f
      cpp --> fcom --> ugen --> as1 --> ld
```

The command

```
%f77 −32 −O foo.f
      cpp --> fcom --> uopt --> ugen --> as1 --> ld
```

also invokes the *ucode* optimizer, *uopt*. The command

```
%f77 −32 −sopt foo.f
      cpp --> fopt --> fcom --> ugen --> as1 --> ld
```

invokes the scalar optimizer but does not invoke the *ucode* optimizer.

The *−mp* option signals *fcom* to recognize inserted parallelization directives:

```
%f77 −32 −mp foo.f
      cpp --> fcom --> ugen --> as1 --> ld
```

The Power Fortran option, *−pfa*:

```
%f77 −32 −pfa foo.f
      cpp --> pfa --> fcom --> ugen --> as1 --> ld
```

invokes the Power Fortran Accelerator to convert programs to parallel code automatically.

Fortran 64−Bit System

The 64−bit Fortran compiler consists of the following components:

f77	Fortran driver: Executes the appropriate components below.
cpp	K&R C preprocessor: Handles *#include* statements in the source file.
fef77	Fortran 77 front end: Parses the source file into an intermediate representation. It also performs scalar optimization and automatic parallelization. A special version of *fef77*, *fef77p*, is available with the Power Fortran product.
fef77p	Power Fortran front end: Parses the source file into an intermediate representation. It also performs scalar optimization and automatic parallelization.
be	Back end: Generates code and assembles it into an object file. It also performs a variety of optimizations which are described in Chapter Four of this book, "Compilation Issues".
ld64	Linker: Links the object file(s) with any libraries. When you run the examples from this section on the 64−bit compiler, the following components are executed:

```
%f77 -64 foo.f
%f77 -64 -O foo.f
%f77 -64 -sopt foo.f
%f77 -64 -mp foo.f
         cpp --> fef77 --> be --> ld64
%f77 -64 -pfa foo.f
         cpp --> fef77p --> be --> ld64
```

Under the MIPSpro 64-bit compiler, scalar optimizations are performed in *fef77*. Further optimizations are performed in the back end.

Note that when *-O* is used under the 64-bit compiler, as opposed to *ucode*, **both** scalar and back end optimizations are performed.

Note also that *-O3* is available with *-c*. Unlike the *ucode* compilers, *-O3* does not result in interprocedural optimizations being performed.

The *-sopt* switch is supported on the 64-bit compiler and causes *fef77* to perform the desired scalar optimizations.

The *-mp* switch is supported on the 64-bit compiler and causes *fef77* to recognize inserted parallelization directives.

The *-pfa* switch is supported on the 64-bit compiler and causes *fef77p* to perform automatic parallelization.

C

For C, the respective compiler systems are similar to their Fortran counterparts. The front ends, of course, are different in each system. Also, *cpp* functionality was included in the C front end (*cfe*).

C 32-Bit System

The 32-bit (*ucode*) C compiler systems contain the following components:

cc	C driver: Executes the appropriate components below.
acpp	ANSI C preprocessor: Handles *#include* statements and other *cpp* constructs such as *#define*, *#ifdef*, and so on, in the source file.
cfe	C front end: Preprocesses the C file, and then parses the source file into intermediate code (*ucode*).
ccom_mp	C front end for Power C: Interprets parallel directives (K & R version).
accom_mp	C front end for Power C: Interprets parallel directives (ANSI version).
copt	C scalar optimizer: Performs scalar optimization.
pca	Power C analyzer: Automatically parallelizes C source code and performs scalar optimizations. Available with the Power C product.
uopt	Optimizer: Performs optimizations on the intermediate file.
ugen	Code Generator: Generates binary assembly code from the intermediate file.
as1	Binary assembler: Assembles the *binasm* code into an object file.
ld	Linker: Links the object file(s) with any libraries.

When you run simple examples through the *ucode* C compiler, the following components are executed by the compiler driver:

```
%cc -32 foo.c
      cfe --> ugen --> as1 --> ld
```

Note: *cfe* has a built-in C preprocessor.

The command

```
%cc -32 -O foo.c
      cfe --> uopt --> ugen --> as1 --> ld
```

also invokes the *ucode* optimizer, *uopt*.

The command

```
%cc -32 -sopt foo.c
      acpp --> copt --> cfe --> ugen --> as1 --> ld
```

invokes the scalar optimizer but does not invoke the *ucode* optimizer.

The C preprocessor has to be run before *copt* can do its source−to−source translation:

```
%cc -32 -mp foo.c
      acpp --> accom_mp --> ugen --> as1 --> ld
```

−mp signals *accom_mp* to recognize inserted parallelization directives.

The command

```
%cc -32 -pca foo.c
      acpp --> pca --> accom_mp --> ugen --> as1 --> ld
```

invokes the Power C accelerator to convert programs to parallel code automatically.

C 64−Bit System

The MIPSpro (64−bit) C compiler systems contain the following components:

cc	C driver: Executes the appropriate components below.
acpp	ANSI C preprocessor: Handles *#include* statements and other *cpp* constructs such as *#define*, *#ifdef*, and so on, in the source file.
copt	C scalar optimizer: Performs scalar optimization.
pca	Power C analyzer: Automatically parallelizes C source code and performs scalar optimizations. Available with the Power C product.
mpc	Interpreter: Interprets parallel directives.
fec	C front end: Preprocesses the C file, and then parses the source file into an intermediate representation.
be	Back end: Generates code and assembles it into an object file.
ld	Linker: Links the object file(s) with any libraries.

When you run simple examples through the 64−bit C compilers, the following components are executed by the compiler driver:

```
%cc -64 foo.c
%cc -64 -O foo.c
          fec --> be --> ld64
%cc -64 -sopt foo.c
          acpp --> copt --> fec --> be --> ld64
%cc -64 -mp foo.c
          acpp --> mpc --> fec --> be --> ld64
%cc -64 -pca foo.f
          acpp --> pca --> mpc --> fec --> be --> ld64
```

Fortran and C 64−Bit Components

Table 1−2 summarizes the major components of both the Fortran and C compilers of the new MIPSpro (64−bit) compiler system.

Table 1−2 Major Components of 64−Bit Fortran and C Compilers

Fortran		C
f77	Driver	*cc*
cpp	Preprocessor	*fec*
fef77	Front end	*fec*
fef77	Scalar optimizer	*copt*
fef77	MP directive interpreter	*mpc*
fef77p	Automatic parallelizer	*pca*
be	Back end	*be*

ld	Linker	*ld*

Kuck and Associates Preprocessor (KAP)

KAP (Kuck and Associates Preprocessor) is a Fortran optimizer that analyzes data dependence to guide traditional serial optimizations and automatic parallelization. It is built into *fef77* (rather, *fef77* consists largely of KAP).

This is an example of a transformation that KAP performs. The following routine:

```
subroutine bar(a,b,c,d,j)
real*4 a(1024),b(1024),c(1024)
real*4 d,e,f
sum = 0
do m= 1,j
do i=1,1024
b(i) = b(i) * d
enddo
enddo
call foo(a,b,c)
end
```

is turned into this by the KAP:

```
# 1 "vmuls.f"
subroutine BAR (A, B, C, D, J )
real*4 A(1024), B(1024), C(1024)
real*4 D, E, F
do 2 M=1,J-3,4
do 2 I=1,1024
B(I) = B(I) * D
B(I) = B(I) * D
B(I) = B(I) * D
B(I) = B(I) * D
2        continue
do 3 M=M,J,1
do 3 I=1,1024
B(I) = B(I) * D
3        continue
call FOO (A,B,C)
end
```

As you can see, some outer loop unrolling has occurred.

Passing Options to KAP

You can pass options to KAP using *–WK,option*, for example:

-WK,-cmp=file.cmp

passes *–cmp=file.cmp* to KAP (which puts the equivalent transformed Fortran into *file.cmp*).

KAP's Optimization Flags (–o, –r, –so)

Kap accepts three important optimization flags, *–o*, *–r*, and *–so*. They are described in this section.

The –o Flag

–o=integer
Long name: *–optimize=integer*

The *−optimize* option sets the optimization level. Each optimization level is cumulative (that is, each level performs everything up to and including its level). The meaning of each of the different optimization levels is:

0	No optimization is done.
1	Only simple optimizations are done. Induction variable recognition is enabled.
2	Performs lifetime analysis to determine when last−value assignment of scalars is necessary.
3	Triangular loops are recognized and loop interchanging is attempted to improve memory referencing. Special case data dependence tests are used. Special index sets, called wraparound variables, are also recognized.
4	Two versions of a loop are generated, if necessary, to break a data dependence arc.
5	Array expansion and loop fusion are enabled.

The −r Flag

−r=integer

Long name: *−roundoff=integer*

The *−roundoff* option allows you to specify the tolerable change from serial roundoff error. If an arithmetic reduction is accumulated in a different order than in the scalar program, the roundoff error is accumulated differently and the final result may differ from that of the original program's output. Although the difference is usually insignificant, certain restructuring transformations performed by the KAP must be disabled to obtain exactly the same answers as those of the scalar program.

KAP classifies its transformations by the amount of difference in roundoff error that can accumulate, so you can decide what level of roundoff error difference is allowable.

The meanings of the switch values are as follows:

0	Allow no roundoff−changing transformations.
1	Enable expression simplification that might generate overflow or underflow errors differently. Enable simplification of expressions with operands between binary and unary operators. Perform expression simplification that is exposed due to forward substitution. Enable code floating if the *−scalaropt* option is greater than or equal to 1.
2	Allow loop interchanging around arithmetic reductions to be recognized.
3	Recognize real (float) induction variables. Enable sum reductions. Enable memory management optimizations if *scalaropt=3*.

The −so Flag

−so=integer

Long name: *−scalaropt=integer*

The *−scalaropt* option sets the level of dusty−deck and other serial transformations performed.

The following are possible values for *integer*:

0	No scalar optimizations are performed.
1	Only simple scalar optimizations are performed ð - dead code elimination, global forward substitution and dusty−deck IF transformations.
2	Full range of scalar optimizations are performed - invariant IF's are floated out of loops, loop rerolling/unrolling, array expansion, loop fusion, loop peeling and induction variable recognition.
3	Memory management optimizations are enabled if *roundoff=3*.

Opimization Levels Passed to KAP

Three optimization flags are passed to KAP; their values are dependent on the optimization level specified, as follows:

−O0 implies *−o=0 −r=0 −so=0*
−O1 implies *−o=1 −r=0 −so=2*
−O2 implies *−o=5 −r=0 −so=3*
−O3 implies *−o=5 −r=2 −so=3*
−sopt implies *−o=5 −r=0 −so=3*
−mp implies *−o=0 −r=0 −so=0* and recognize parallel directives

−pfa implies *−o=5 −r=0 −so=3* (plus automatic parallelization)

For combinations of either *−On −pfa* or *−On −sopt*, the greater of the implied options is taken. For example, with *−O1 −sopt*, the resulting set of KAP options is *−o=5 −r=1 −so=3*.

Multiprocessing Support

IRIX 6 and the MIPSpro compilers support multiprocessing primitives for both 32−bit and 64−bit applications. The 64−bit multiprocessor programming environment is a superset of the 32−bit one. It also contains enhancements.

MP Compatibility

This section describes 64−bit and 32−bit Fortran MP compiler compatibility:

- The 64−bit Fortran compiler supports all of the parallelization directives (such as *C$DOACROSS, C$&, C$MP_SCHEDTYPE, C$CHUNK, C$COPYIN*) supported by the 32−bit Fortran compiler. *fef77* also supports PCF style parallel directives, which are documented in the *MIPSpro Fortran 77 Programmer's Guide*.
- The 64−bit Fortran compiler supports the same set of multiprocessing utility subroutine calls (such as *mp_block* and *mp_unblock*) as the 32−bit compiler.

  The 64−bit Fortran compiler supports the same set of environment variables (such as *MP_SET_NUMTHREADS* and *MP_BLOCKTIME*) as the 32−bit compiler.
- The *−mp* and *−pfa* switches are supported on both the 32−bit compilers and the 64−bit compilers.
  - *−mp* allows KAP to recognize hand−inserted parallelization directives.
  - *−pfa* enables automatic parallelization by KAP.

MP Enhancements

The MIPSpro 64−bit Fortran MP I/O library has been enhanced to allow I/O from parallel regions. In other words, multiple threads can read and write to different files as well as read and write to the same file. The latter case, of course, encounters normal overhead due to file locking.

The MIPSpro 64−bit compilers also have been enhanced to allow parallel C and parallel Fortran programs to share a common runtime. This allows you to link parallel C routines with parallel Fortran routines and have a single master. Figure 1−2 illustrates this.

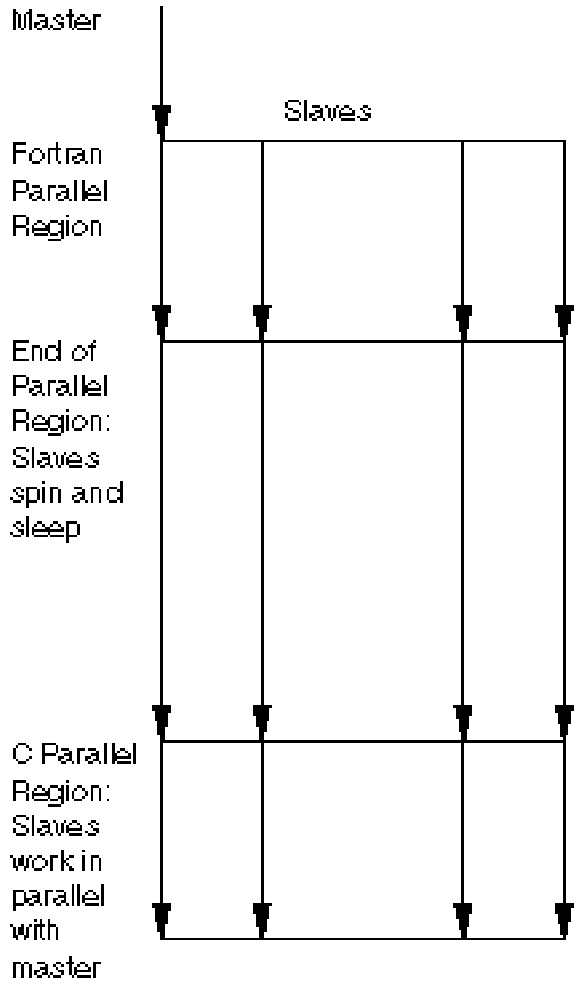

Figure 1−2 Running Parallel C and Parallel Fortran Programs Together

MP Application Testing

To test 64−bit MP applications on the R8000, follow these guidelines:

- First, get the application to run with no parallelization at the highest optimization level.
- When testing the parallel version, first run it with only one thread (either on a single CPU machine or by setting the environment variable *MP_SET_NUMTHREADS* to 1).
- Go down to the *−g* optimization level for the first MP test, and run that version with one thread, then with multiple threads. Then go up the optimization scale, testing both single and multi−threaded.

You can, of course, skip as many steps as you like. In case of failure, however, this method of incremental iterations can most likely help you identify the problem.

Chapter 2

Language Implementation Differences

This chapter describes where the 32−bit and 64−bit compilers differ with respect to calling conventions and language implementations. The first section describes the 64−bit subprogram interface. The next two sections identify differences in the 32−bit and 64−bit implementations of the Fortran 77 and C programming languages, respectively.

Native 64–Bit Subprogram Interface for MIPS Architectures

This section describes the internal subprogram interface for native 64–bit programs. It assumes some familiarity with the current 32–bit interface conventions as specified in the MIPS application binary interface (ABI). The transition to native 64–bit code on the MIPS R8000 requires subprogram interface changes due to the changes in register and address size.

The principal interface for 64–bit native code is similar to the 32–bit ABI standard, with all 32–bit objects replaced by 64–bit objects. In particular, this implies:

- All integer parameters are promoted (that is, sign– or zero–extended to 64–bit integers and passed in a single register). Normally, no code is required for the promotion.
- All pointers and addresses are 64–bit objects.
- Floating point parameters are passed as single– or double–precision according to the ANSI C rules.
- All stack parameter slots become 64–bit doublewords, even for parameters that are smaller (for example, floats and 32–bit integers).

In more detail, the 64–bit native calling sequence has the following characteristics. Square brackets are used to indicate different 32–bit ABI conventions.

- All stack regions are quadword–aligned. [The 32–bit ABI specifies only doubleword alignment.]
- Up to eight integer registers *($4 .. $11)* may be used to pass integer arguments. [The 32–bit ABI uses only the four registers *$4 .. $7*.]
- Up to eight floating point registers (*$f12 .. $f19*) may be used to pass floating point arguments. [The 32–bit ABI uses only the four registers *$f12 .. $f15*, with the odd registers used only for halves of double–precision arguments.]
- The argument registers may be viewed as an image of the initial eight doublewords of a structure containing all of the arguments, where each of the argument fields is a multiple of 64 bits in size with doubleword alignment. The integer and floating point registers are distinct images, that is, the first doubleword is passed in either *$4* or *$f1*, depending on its type; the second in either *$5* or *$f1*; and so on. [The 32–bit ABI associates each floating point argument with an even/odd pair of integer or floating point argument registers.]
- Within each of the 64–bit save area slots, smaller scalar parameters are right–justified, that is, they are placed at the highest possible address (for big–endian targets). This is relevant to float parameters and to integer parameters of 32 or fewer bits. Of these, only *int* parameters arise in C except for prototyped cases - *floats* are promoted to *doubles*, and smaller integers are promoted to *int*. [This is true for the 32–bit ABI, but is relevant only to prototyped small integers since all the other types were at least register–sized.]
- 32–bit integer (*int*) parameters are always sign–extended when passed in registers, whether of signed or unsigned type. [This issue does not arise in the 32–bit ABI.]
- Quad–precision floating point parameters (C *long double* or Fortran *REAL*16*) are always 16–byte aligned. This requires that they be passed in even–odd floating point register pairs, even if doing so requires skipping a register parameter and/or a 64–bit save area slot. [The 32–bit ABI does not consider long double parameters, since they were not supported.]
- *Structs*, *unions*, or other composite types are treated as a sequence of doublewords, and are passed in integer or floating point registers as though they were simple scalar parameters to the extent that they fit, with any excess on the stack packed according to the normal memory layout of the object. More specifically:
  - Regardless of the *struct* field structure, it is treated as a sequence of 64–bit chunks. If a chunk consists solely of a double float field (but not a *double*, which is part of a *union*), it is passed in a floating point register. Any other chunk is passed in an integer register .
  - A *union*, either as the parameter itself or as a *struct* parameter field, is treated as a sequence of integer doublewords for purposes of assignment to integer parameter registers. No attempt is made to identify floating point components for passing in floating point registers.
  - Array fields of *structs* are passed like *unions*. Array parameters are passed by reference (unless the relevant language standard requires otherwise).
  - Right–justifying small scalar parameters in their save area slots notwithstanding, *struct* parameters are always left–justified. This applies both to the case of a *struct* smaller than 64 bits, and to the final chunk of a *struct* which is not an integral multiple of 64 bits in size. The implication of this rule is that the address of the first chunk's save area slot is the address of the *struct*, and the *struct* is laid out in the save area memory exactly as if it were allocated normally

(once any part in registers has been stored to the save area). [These rules are analogous to the 32−bit ABI treatment - only the chunk size and the ability to pass double fields in floating point registers are different.]

- Whenever possible, floating point arguments are passed in floating point registers regardless of whether they are preceded by integer parameters. [The 32−bit ABI allows only leading floating point (FP) arguments to be passed in FP registers; those coming after integer registers must be moved to integer registers.]
- Variable argument routines require an exception to the previous rule. Any floating point parameters in the variable part of the argument list (leading or otherwise) are passed in integer registers. There are several important cases involved:
  - If a *varargs* prototype (or the actual definition of the callee) is available to the caller, it places floating point parameters directly in the integer register required, and there are no problems.
  - If no prototype is available to the caller for a direct call, the caller's parameter profile is provided in the object file (as are all global subprogram formal parameter profiles), and the linker (*ld/rld*) generates an error message if the linked entry point turns out to be a *varargs* routine.

    Note: The linker could insert a stub if required to move floating point parameters to the corresponding integer registers, but we do not intend to do so currently, since ANSI requires prototypes for *varargs* routines.

  - If no prototype is available to the caller for an indirect call (that is, via a function pointer), the caller assumes that the callee is not a *varargs* routine and places floating point parameters in floating point registers (if the callee is *varargs*, it is not ANSI−conformant).
- The portion of the argument structure beyond the initial eight doublewords is passed in memory on the stack and pointed to by the stack pointer at the time of call. The caller does not reserve space for the register arguments; the callee is responsible for reserving it if required (either adjacent to any caller−saved stack arguments if required, or elsewhere as appropriate.) No requirement is placed on the callee either to allocate space and save the register parameters, or to save them in any particular place. [The 32−bit ABI requires the caller to reserve space for the register arguments as well.]
- Function results are returned in *$2* (and *$3* if needed), or *$f0* (and *$f2* if needed), as appropriate for the type. Composite results (*struct*, *union*, or *array*) are returned in *$2*/*$f0* and *$3*/*$f2* according to the following rules:
  - A *struct* with only one or two floating point fields is returned in *$f0* (and *$f2* if necessary). This is a generalization of the Fortran *COMPLEX* case.
  - Any other *struct* or *union* results of at most 128 bits are returned in *$2* (first 64 bits) and *$3* (remainder, if necessary).
  - Larger composite results are handled by converting the function to a procedure with an implicit first parameter, which is a pointer to an area reserved by the caller to receive the result. [The 32−bit ABI requires that all composite results be handled by conversion to implicit first parameters. The MIPS/SGI Fortran implementation has always made a specific exception to return *COMPLEX* results in the floating point registers.]
- There are eight callee−saved floating point registers, *$f24..$f31*. [The 32−bit ABI specifies the six even registers, or even/odd pairs, *$f20..$f31*.]
- Routines are not be restricted to a single exit block. [The 32−bit ABI makes this restriction, though it is not observed under all optimization options.]

There is no restriction on which register must be used to hold the return address in exit blocks. The *.mdebug* format was unable to cope with return addresses in different places, but the DWARF format can. [The 32−bit ABI specifies *$3*, but the implementation supports *.mask* as an alternative.]

PIC (position−independent code, for DSO support) is generated from the compiler directly, rather than converting it later with a separate tool. This allows better compiler control for instruction scheduling and other optimizations, and provides greater robustness.

In the 64−bit interface, *gp* becomes a callee−saved register. [The 32−bit ABI makes *gp* a caller−saved register.]

Table 2−1 specifies the use of registers in native 64−bit mode. Note that "caller−saved" means only that the caller may not assume that the value in the register is preserved across the call.

Table 2−1 Native 64−Bit Interface Register Conventions

Register Name	Software Name	Use	Saver
$0	zero	Hardware zero	
$1 or *$at*	*at*	Assembler temporary	Caller−saved
$2..$3	*v0..v1*	Function results	Caller−saved
$4..$11	*a0..a7*	Subprogram arguments	Caller−saved
$12..$15	*t4..t7*	Temporaries	Caller−saved
$16..$23	*s0..s7*	Saved	Callee−saved
$24	*t8*	Temporary	Caller−saved
$25	*t9*	Temporary	Caller−saved
$26..$27	*kt0..kt1*	Reserved for kernel	
	gp	Global pointer	Callee−saved
$29 or *$sp*	*sp*	Stack pointer	Callee−saved
$30	*s8*	Frame pointer (if needed)	Callee−saved
$31	*ra*	Return address	Caller−saved
hi, *lo*		Multiply/divide special registers	Caller−saved
$f0, *$f2*		Floating point function results	Caller−saved
$f1, *$f3*		Floating point temporaries	Caller−saved
$f4..$f11		Floating point temporaries	Caller−saved
$f12..$f19		Floating point arguments	Caller−saved
$f20..$f23		Floating point temporaries	Caller−saved
$f24..$f31		Floating point	Callee−saved

Table 2−2 gives several examples of parameter passing. It illustrates that at most eight values can be passed through registers. In the table note that:

- *d1..d5* are double precision floating point arguments
- *s1..s4* are single precision floating point arguments
- *n1..n3* are integer arguments

Table 2−2 Native 64−Bit C Parameter Passing

Argument List	Register and Stack Assignments
d1,d2	*$f12, $f13*
s1,s2	*$f12, $f13*
s1,d1	*$f12, $f13*
d1,s1	*$f12, $f13*
n1,d1	*$4,$f13*
d1,n1,d1	*$f12, $5,$f14*
n1,n2,d1	*$4, $5,$f14*
d1,n1,n2	*$f12, $5,$6*
s1,n1,n2	*$f12, $5,$6*
d1,s1,s2	*$f12, $f13, $f14*
s1,s2,d1	*$f12, $f13, $f14*
n1,n2,n3,n4	*$4,$5,$6,$7*
n1,n2,n3,d1	*$4,$5,$6,$f15*
n1,n2,n3,s1	*$4,$5,$6, $f15*
s1,s2,s3,s4	*$f12, $f13,$f14,$f15*
s1,n1,s2,n2	*$f12, $5,$f14,$7*
n1,s1,n2,s2	*$4,$f13,$6,$f15*
n1,s1,n2,n3	*$4,$f13,$6,$7*
d1,d2,d3,d4,d5	*$f12, $f13, $f14, $f15, $f16*
d1,d2,d3,d4,d5,s1,s2,s3,s4	*$f12, $f13, $f14, $f15, $f16, $f17, $f18,$f19,stack*
d1,d2,d3,s1,s2,s3,n1,n2,n3	*$f12, $f13, $f14, $f15, $f16, $f17, $10,$11, stack*

Fortran Implementation Differences

This section lists differences between the 32−bit and the 64−bit Fortran implementations. Command line argument compatibility is described in Chapter 4. The 32−bit Fortran front end is called *fcom* and the 64−bit front end is called *fef77*.

New Fortran Features

fef77 implements *REAL*16* and *COMPLEX*32* and all associated intrinsics as 16−byte floating point entities. *fcom* recognizes them, but converts them to *REAL*8* and *COMPLEX*16* respectively.

Incompatibilities and Differences

- *fcom* and *fef77* are incompatible in the way they fold *REAL*4* constants. *fcom* promotes them internally to *REAL*8*.
- *fcom* allows more constant expressions in parameter statements than *fef77*.
- *fcom* allows lines longer than 256 characters by default; *fef77* currently has a hard limit of 256.
- *fef77* allows parameters (which are ignored) to the program statement.
- *fef77* allows empty parameters in subroutine/function calls, which result in zeroes being passed by value.
- *fef77* implements the Fortran 8*x* array syntax. (Fortran 8*x* is a subset of Fortran 90 that was implemented by various vendors before the Fortran 90 standard was complete).
- *fef77* accepts PCF−style parallel directives in addition to the directives such as *C$DOACROSS,* which *fcom* accepts. PCF−style directives are documented in the *MIPSpro Fortran 77 Programmer's Guide*.

C Implementation Differences

This section lists differences between the 32−bit and the 64−bit C implementations. Because both compilers adhere to the ANSI standard, and because C is a rigorously defined language designed to be portable, there are not many differences between the 32−bit and 64−bit compiler implementations. The only areas where differences can occur are in data types (by definition) and in areas where ANSI does not define the precise behavior of the language.

Table 2−3 summarizes the differences in data types under the 32−bit and 64−bit data type models.

Table 2−3 Differences in Data Type Sizes

C type	32−bit	64−bit
char	8	8
short int	16	16
int	32	32
long int	32	64
long long int	64	64
pointer	32	64
float	32	32
double	64	64
long double[a]	64	128

[a]On 32−bit compiles the *long double* data type generates a warning message indicating that the *long* qualifier is not supported.

Table 2−3 shows that *long ints*, *pointers* and *long doubles* are different under the two models.

Structure and Union Layout Examples

Simple examples illustrate the alignment and size issues of C structures and unions.

Example 1: Structure Smaller Than a Word

```
struct c {
     char c;
```

```
} c1;
```

Byte–aligned, *sizeof* struct is 1.

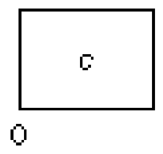

Figure 2–1 Structure Smaller Than a Word

Example 2: Structure With No Padding

```
struct s {
     char c;
     char d;
     short s;
     int   i;
} s1;
```

Word–aligned, *sizeof* struct is 8.

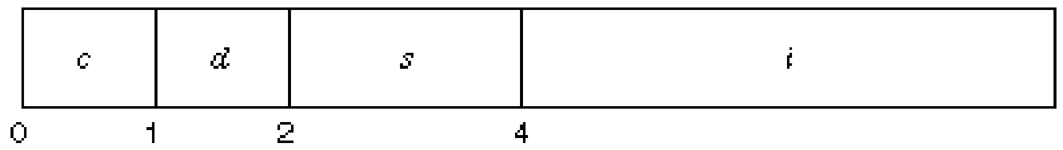

Figure 2–2 Structure With No Padding

Example 3: Structure With Internal Padding

```
struct t {
     char c;
     char d;
     short s;
     long  l;
} t1;
```

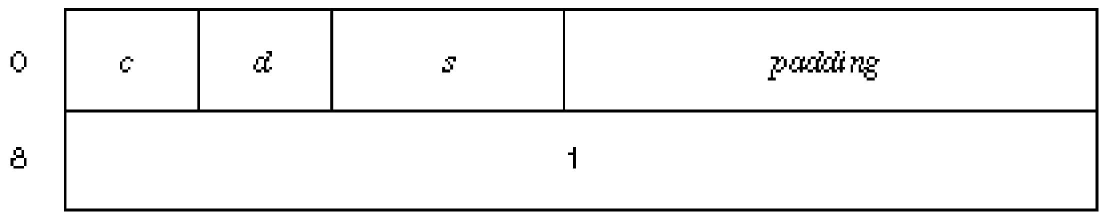

Figure 2–3 Structure With Internal Padding

Example 4: Structure With Internal and Tail Padding

```
struct l {
     char c;
     long  l;
     short s;
} l1;
```

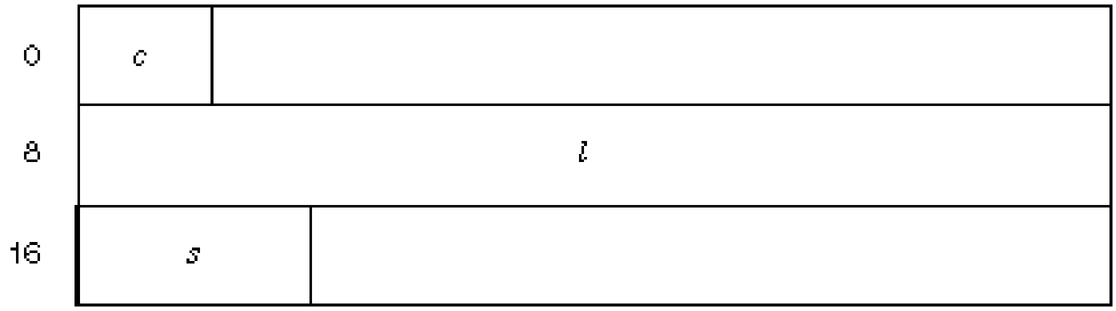

Figure 2–4 Structure With Internal and Tail Padding

Example 5: Union Allocation

```
union u {
     char c;
     short s;
     int  i;
     long  l;
} u1;
```

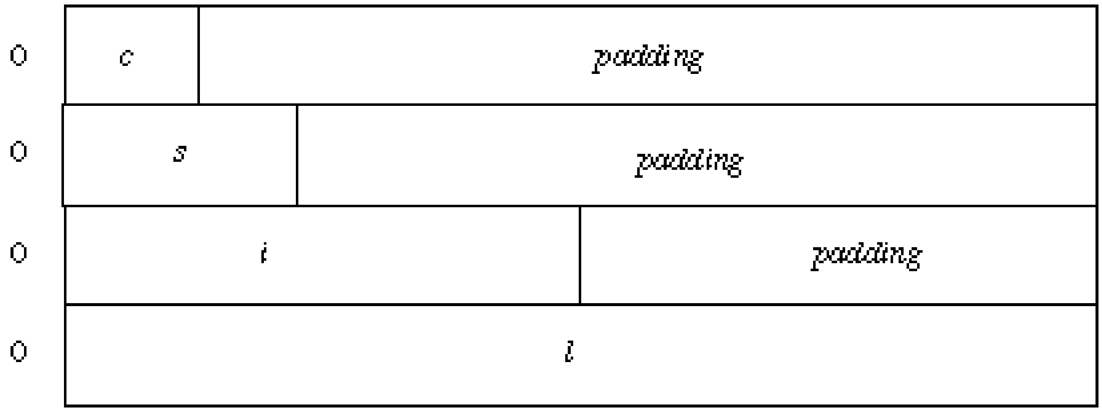

Figure 2–5 Union Allocation

Portability Issues

If code was originally written with portability in mind, the type size differences should not be difficult to reconcile. However, production code is often written without regard for portability.When porting code written without regard to portability the following areas should be handled carefully:

- Equivalence of *pointers* and *ints*
- Equivalence of *long ints* and *ints*
- Code without prototypes

These areas a covered in depth in Chapter 3, "Source Code Porting":

Chapter 3

Source Code Porting

This chapter describes changes you must make to your application source code to port it from a 32−bit to a 64−bit system. The first section outlines changes to Fortran code. The second and third sections deal with C source code issues. The fourth section provide guidelines on writing portable assembly language and C code (with respect to 32−bit and 64−bit systems).

64−Bit Fortran Porting Guidelines

This section describes which sections of your Fortran source code you need to modify to port to a 64−bit system.

Standard ANSI Fortran 77 code should have no problems, but the following areas need attention:

- Code that uses *REAL*16* could get different runtime results due to additional accuracy in the QUAD libraries.
- *fcom* and *fef77* are incompatible with regard to real constant folding.
- Integer variables which were used to hold addresses need to be changed to *INTEGER*8*.
- C interface issues (Fortran passes by reference).
- *%LOC* returns 64−bit addresses.
- *%VAL* passes 64−bit values.

Source code modifications are best done in phases. In the first phase, try to locate all of the variables affected by the size issues mentioned above. In the second phase, locate variables that depend on the variables changed in the first phase. Depending on your code, you may need to iterate on this phase as you track down long sets of dependencies.

Examples of Fortran Portability Issues

The following examples illustrate the variable size issues outlined above:

Example 1: Changing Integer Variables

Integer variables used to hold addresses must be changed to *INTEGER*8*.

32−bit code:

```
integer iptr, asize
iptr = malloc(asize)
```

64−bit code:

```
integer asize
integer*8 iptr
iptr = malloc(asize)
```

Example 2: Enlarging Tables

Tables which hold integers used as pointers must be enlarged by a factor of two.

32−bit code:

```
integer tableptr, asize, numptrs
numptrs = 100
asize = 100 * 4
```

```
tableptr = malloc(asize)
```

64−bit code:

```
integer asize, numptrs
integer*8 tableptr
numptrs = 100
asize = 100 * 8
tableptr = malloc(asize)
```

Example 3: Using *#if* Directives with Predefined Variables.

You should use *#if* directives so that your source code can be compiled either *−32* or *−64*. The compilers support predefined variables such as *_MIPS_SZPTR* or *_MIPS_SZLONG*, which can be used to differentiate 32−bit and 64−bit source code. A later section provides a more complete list of predefined compiler variables and their values for 32−bit and 64−bit operation. For example, the set of changes in the previous example could be coded:

```
integer asize, numptrs

#if (_MIPS_SZPTR==64)
integer*8 tablept
asize = 100 * 8
#else
integer*4 tableptr
asize = 100 * 4
#endif

tableptr = malloc(asize)
```

Example 4: Storing *%LOC* Return Values

%LOC returns 64−bit addresses. You need to use an *INTEGER*8* variable to store the return value of a *%LOC* call.

```
#if (_MIPS_SZLONG==64)
INTEGER*8 HADDRESS
#else
INTEGER*4 HADDRESS
#endif


C determine memory location of dummy heap array
   HADDRESS = %LOC(HEAP)
```

Example 5: Modifying C Routines Called by Fortran

C routines which are called by Fortran where variables are passed by reference must be modified to hold 64−bit addresses.Typically, these routines used *ints* to contain the addresses in the past. For 64−bit use, at the very least, they should use *long ints*. There are no problems if the original C routines simply define the parameters as *pointers*.

Fortran:

```
call foo(i,j)
```

C:

```
foo_( int *i, int *j) or at least
foo_( long i, long j)
```

Example 6: Declaring Fortran Arguments as long ints

Fortran arguments passed by *%VAL* calls to C routines should be declared as *long ints* in the C routines.

Fortran:

```
call foo(%VAL(i))
```

C:

```
foo_( long i )
```

Example 7: Changing Argument Declarations in Fortran Subprograms

Fortran subprograms called by C where *long int* arguments are passed by address need to change their argument declarations.

C:

```
long l1, l2;
foo_(&l1, &l2);
```

Fortran:

```
subroutine foo(i, j)
#if (_MIPS_SZLONG==64)
INTEGER*8 i,j
#else
INTEGER*4 i,j
#endif
```

64−Bit C Porting Guidelines

This section details the issues involved in porting 32−bit C application code to 64 bits. It addresses both the porting of existing code and guidelines for writing code to be ported at a later date.

Porting programs written in C to a Silicon Graphics 64−bit MIPS architecture platform, using the native 64−bit C compilers and related tools, should be straightforward. However, depending on assumptions made in writing the original C code, it may require some changes.

The C integer data types historically have been sized based on matching desired functionality with the target architecture's ability to efficiently implement integers of various sizes.

The SGI 64−bit platforms support the LP64 model for native 64−bit programs. In this model, *pointers* and *long* integers are 64 bits.

In the sections below, we discuss what problems to expect in porting existing C code to the LP64 native model, and suggest approaches for avoiding porting problems in new code.

Porting to the LP64 Model

For code which currently runs on SGI 32−bit platforms, porting to the LP64 model is straightforward. (It may also be unnecessary. Unless the program requires a 64−bit address space, 64−bit data, or other native−only functionality, it may still be run as a 32−bit program.)

Porting requires, at minimum, recompiling and relinking the program. You must specify a 64−bit target if you are doing this on a 32−bit workstation; on a 64−bit workstation this is the default (and you must request the 32−bit compatibility model if you want it). In some cases, the differences between the models imply changes in SGI−provided system header files and/or libraries; in such cases, your selection of the 32−bit or LP64 model selects the correct version automatically.

Within your code, most porting problems arise from assumptions, implicit or explicit, about either absolute or relative sizes of the *int*, *long int*, or *pointer* types. The most common are likely to be:

- *sizeof(void *) == 4*

  This assumption is analogous to the previous one. But mappings to external data structures should seldom be a problem, since the external definition should also assume 64-bit pointers in the LP64 model.

- constants

  The change in type sizes may yield some surprises related to constants. You should be especially careful about using constants with the high-order (sign) bit set. For instance, the hex constant 0xffffffff yields different results in the expression:

  ```
  long x;
  ... ( (long) ( x + 0xffffffff ) ) ...
  ```

  In both models, the constant is interpreted as a 32-bit *unsigned int*, with value 4,294,967,295. In the 32-bit model, the addition result is a 32-bit *unsigned long*, which is cast to *type long* and has value *x-1* because of the truncation to 32 bits. In the LP64 model, the addition result is a 64-bit *long* with value *x+4,294,967,295*, and the cast is redundant.

- arithmetic assumptions

  Related to some of the above cases, code which does arithmetic (including shifting) which may overflow 32 bits, and assumes particular treatment of the overflow (for example, truncation), may exhibit different behavior in the LP64 model, depending on the mix of types involved (including signedness).

  Similarly, implicit casting in expressions which mix int and long values may behave unexpectedly due to sign/zero extension. In particular, remember that integer constants are sign or zero extended when they occur in expressions with long values.

  Once identified, each of these problems is easy to solve. Change the relevant declaration to one which has the desired characteristics in both target environments, add explicit type casts to force the correct conversions, use function prototypes, or use type suffixes (for example, 'l' or 'u') on constants to force the correct type.

Writing Code Portable to 64-Bit Platforms

The key to writing new code which is compatible with the 32-bit and LP64 data models described is to avoid those problems described above. Since all of the assumptions described sometimes represent legitimate attributes of data objects, this requires some tailoring of declarations to the target machines' data models.

We suggest observing the following guidelines to produce code without the more common portability problems. They can be followed from the beginning in developing new code, or adopted incrementally as portability problems are identified.

In a header file which can be included in each of the program's source files, define (*typedef*) a type for each of the following functions:

- For each specific integer data size required, that is, where exactly the same number of bits is required on each target, define a *signed* and *unsigned* type, for example:

  ```
  typedef signed char int8_t
  typedef unsigned char uint8_t
  ...
  typedef unsigned long long uint64_t
  ```

- If you require a large scaling integer type, that is, one which is as large as possible while remaining efficiently supported by the target, define another pair of types, for example:

  ```
  typedef signed long intscaled_t
  typedef unsigned long uintscaled_t
  ```

- If you require integer types of at least a particular size, but chosen for maximally efficient implementation on the target, define another set of types, similar to the first but defined as larger standard types where appropriate for efficiency.

Having constructed the above header file, use the new *typedef*ed types instead of the standard C type names. You need (potentially) a distinct copy of this header file (or conditional code) for each target platform supported. As a special case of this, if you are providing libraries or interfaces to be used by others, be particularly careful to use these types (or similar application specific types) chosen to match the specific requirements of the interface. Also in such cases, you should choose the actual names used to avoid name space conflicts with other libraries doing the same thing. If this is done carefully, your clients should be able to use a single set of header files on all targets. However, you generally need to provide distinct libraries (binaries) for the 32−bit compatibility model and the LP64 native model on 64−bit SGI platforms, though the sources may be identical.

Be careful that constants are specified with appropriate type specifiers so that they extend to the size required by the context with the values that you require. Bit masks can be particularly troublesome in this regard:avoid using constants for negative values. For example, 0xffffffff may be equivalent to a −1 on 32−bit systems, but it is interpreted as 4,294,967,295 (*signed* or *unsigned*) on 64−bit systems. The *inttypes.h* header file provides *cpp* macros to facilitate this conversion.

Defining constants which are sensitive to type sizes in a central header file may help in modifying them when a new port is done. Where *printf()*/*scanf()* are used for objects whose types are *typedef*ed differently among the targets you must support, you may need to define constant format strings for each of the types defined in step (1), for example:

```
#define _fmt32 "%d"
#define _fmt32u "%u"
#define _fmt64 "%ld"
#define _fmt64u "%lu"
```

The *inttypes.h* header file also defines *printf*()/*scanf*() format extensions to standardize these practices.

Fundamental Types for C

This section discusses 'fundamental types' useful in converting C code from 32−bit to 32− or 64−bit. These take the form of *typedefs*, and are available in the file *<sgidefs.h>*. These typedefs are enabled by compiler predefines, which are also described. This discussion is entirely from the C point of view, although the predefines discussed are also emitted by the other compilers.

It is desirable to have source code that can be compiled either in 32−bit mode or 64−bit mode. An example is *libc*, which we provide in both 32−bit and 64−bit form. (In this discussion, 32−bit code means *mips1* or *mips2*, 64−bit code means *mips3* or *mips4*.)

As previously mentioned, the compilation model chosen for 64−bit objects is referred to as LP64, where *longs* and *pointers* are 64 bits, and *ints* remain at 32 bits. Since *ints* and *pointers* are no longer the same size, and *ints* and *longs* are not the same size, a lot of code can break in this compilation model.

The *typedefs* discussed, in their naming convention, explicitly call out certain attributes of the *typedef*. The goal of this, by naming those attributes, is to ease the long term maintenance of code which has to compile in both the 32−bit and 64−bit models.

The *typedefs* are enabled by predefines from the compilers. The predefines that the compilers emit are:

For MIPS1executables:

```
-D_MIPS_FPSET=16
-D_MIPS_ISA=_MIPS_ISA_MIPS1
-D_MIPS_SIM=_MIPS_SIM_ABI32
-D_MIPS_SZINT=32
-D_MIPS_SZLONG=32
-D_MIPS_SZPTR=32
```

For MIPS3 executables:

```
-D_MIPS_FPSET=32
-D_MIPS_ISA=_MIPS_ISA_MIPS3
-D_MIPS_SIM=_MIPS_SIM_ABI64
-D_MIPS_SZINT=32
-D_MIPS_SZLONG=64
-D_MIPS_SZPTR=64
```

For MIPS4 executables:

```
-D_MIPS_FPSET=32
-D_MIPS_ISA=_MIPS_ISA_MIPS4
-D_MIPS_SIM=_MIPS_SIM_ABI64
-D_MIPS_SZINT=32
-D_MIPS_SZLONG=64
-D_MIPS_SZPTR=64
```

The explanation of these predefines is as follows:

- MIPS_ISA is Mips Instruction Set Architecture. MIPS_ISA_MIPS1 and MIPS_ISA_MIPS3 would be the most common variants for kernel level assembler code.
- MIPS_ISA_MIPS4 is the ISA for R8000 applications. MIPS_SIM is Mips Subprogram Interface Model −− this describes the subroutine linkage convention and register naming/usage convention.
- _MIPS_FPSET describes the number of floating point registers. The MipsIII compilation model makes use of the extended floating point registers available on the R4000.
- _MIPS_SZINT, _MIPS_SZLONG, and _MIPS_SZPTR describe the size of each of those types.

An example of the use of these predefined variables:

```
#if (_MIPS_SZLONG == 32)
typedef int    ssize_t;
#endif
#if (_MIPS_SZLONG == 64)
typedef long    ssize_t;
#endif
```

The *typedefs* following are largely self−explanatory. These are from *<sgidefs.h>*:

```
__int32_t     Signed 32 bit integral type
__uint32_t    Unsigned 32 bit integral type
__int64_t     Signed 64 bit integral type
__uint64_t    Unsigned 64 bit integral type
```

These are "pointer−sized *int*" and "pointer−sized *unsigned int* respectively. As such, they are guaranteed to have the same number of bits as a pointer.

```
__psint_t
__psunsigned_t
```

These are ’scaling *int*’ and ’scaling *unsigned*’ respectively, and are intended for variables that you want to grow as the code is compiled in the 64−bit model.

```
__scint_t
__scunsigned_t
```

The usefulness of these types is that they free the coder from having to know the underlying compilation model −− indeed, that model can change, and the code should still work. In this respect, use of these *typedefs* is better than replacing the assumption, that an *int* and a *pointer* are the same size with the new assumption, that a *long* and a *pointer* are the same size.’

Assembly Language Coding Guidelines

This section describes techniques for writing assembler code which can be compiled and run as either a 32−bit or 64−bit executable. These techniques are based on using certain predefined variables of the compiler, and on macros defined in *sys/asm.h* and *sys/regdef.h* which rely on those compiler predefines. Together, they enable a fairly easy conversion of existing assembly code to run in either the 32−bit or LP64 compilation model. They also allow retargeted assembler code to look fairly uniform in the way it is converted.

Overview and Predefined Variables

There are two sets of issues: the LP64 model, and the new calling conventions. Each of these issues is

solved by a combination of predefined variables that the compiler emits, and macros in *<sys/asm.h>* and *<sys/regdef.h>*, that use those predefine variables to define macros appropriately.
The predefines that the assembler emits are:

For MIPS1/2 executables:

```
-D_MIPS_FPSET=16
-D_MIPS_ISA=_MIPS_ISA_MIPS1
-D_MIPS_SIM=_MIPS_SIM_ABI32
-D_MIPS_SZINT=32
-D_MIPS_SZLONG=32
-D_MIPS_SZPTR=32
```

For MIPS3 executables:

```
-D_MIPS_FPSET=32
-D_MIPS_ISA=_MIPS_ISA_MIPS3
-D_MIPS_SIM=_MIPS_SIM_ABI64
-D_MIPS_SZINT=32
-D_MIPS_SZLONG=64
-D_MIPS_SZPTR=64
```

For MIPS4 executables:

```
-D_MIPS_FPSET=32
-D_MIPS_ISA=_MIPS_ISA_MIPS4
-D_MIPS_SIM=_MIPS_SIM_ABI64
-D_MIPS_SZINT=32
-D_MIPS_SZLONG=64
-D_MIPS_SZPTR=64
```

The explanation of these predefined variables is as follows:

- *MIPS_ISA* is MIPS Instruction Set Architecture. *MIPS_ISA_MIPS1* and *MIPS_ISA_MIPS3* would be the most common variants for kernel–level assembler code.
- *MIPS_ISA_MIPS4* is the ISA for R8000 applications. *MIPS_SIM* is MIPS Subprogram Interface Model - this describes the subroutine linkage convention and register naming/usage convention.
- *_MIPS_FPSET* describes the number of floating point registers. The MipsIII compilation model makes use of the extended floating point registers available on the R4000.
- *_MIPS_SZINT*, *_MIPS_SZLONG*, and *_MIPS_SZPTR* describe the size of each of those types.

An example of the use of these macros:

```
  #if (_MIPS_ISA == _MIPS_ISA_MIPS1 || _MIPS_ISA == _MIPS_ISA_MIPS2)
#define SZREG           4
#endif

  #if (_MIPS_ISA == _MIPS_ISA_MIPS3 || _MIPS_ISA == _MIPS_ISA_MIPS4)
  #define SZREG           8
  #endif
```

LP64 Model Implications for Assembly Language Code

The MIPSpro 64–bit C compiler generated code in the LP64 model –– that is, *pointers* and *longs* are 64 bits, *ints* are 32 bits. This means that all assembler code which deals with either *pointers* or *longs* needs to be converted to using doubleword instructions for MipsIII/IV code, and must continue to use word instructions for MipsI/II code.

Macros in *<sys/asm.h>*, coupled with the compiler predefines, provide a solution to this problem. These macros look like *PTR_<op>* or *LONG_<op>*, where *op* is some operation such as *L* for load, or *ADD*, etc.. These *ops* use standard defines such as *_MIPS_SZPTR* to resolve to doubleword opcodes for MIPS3, and word opcodes for MIPS1. There are specific macros for PTR ops, for LONG ops, and for INT ops.

The second implication of LP64 is that there is a different subroutine linkage convention, and a different register naming convention. The compiler predefine *_MIPS_SIM* enables macros in *<sys/asm.h>* and

<sys/regdef.h> Some important ramifications of that linkage convention are described below.

In the *_MIPS_SIM_ABI64* model there are 8 argument registers - *$4 .. $11*. These additional 4 argument registers come at the expense of the temp registers in *<sys/regdef.h>.* In this model, there are no registers *t4 .. t7*, so any code using these registers does not compile under this model. Similarly, the register names *a4 .. a7* are not available under the *_MIPS_SIM_ABI32* model. (It should be pointed out that those temporary registers are not lost –– the argument registers can serve as scratch registers also, with certain constraints.)

To make it easier to convert assembler code, the new names *ta0*, *ta1*, *ta2*, and *ta3* are available under both *_MIPS_SIM models*. These alias with *t4 .. t7* in the 32–bit world, and with *a4 ..a7* in the 64–bit world.

Another facet of the linkage convention is that the caller no longer has to reserve space for a called function to store its arguments in. The called routine allocates space for storing its arguments on its own stack, if desired. The *NARGSAVE* define in *<sys/asm.h>* helps with this.

In conclusion, predefines from the compiler enable a body of macros to generate 32/64–bit asm code. Those macros are defined in *<sys/asm.h>*, *<sys/regdef.h>*, and *<sys/fpregdef.h>*

The following is an example of some extensions to *asm.h* to handle assembly language coding for 64–bit and KPIC (KPIC requires that the asm coder deal with PIC issues).

A set of defines and macros are defined and used in converting *libc*. These macros are aimed at dealing with the new 64 calling convention and the fact the *ptrs* and *longs* are 64–bit.

Most of the changes required are very straightforward –– usually it's just a more stylized approach toward nested assembly language functions.

Here is the new version of *sigsetjmp.s,* followed by the relevant portions of *asm.h*:

```
   LOCALSZ=        4               # save a0, a1, ra, gp
 FRAMESZ=        (((NARGSAVE+LOCALSZ)*SZREG)+ALSZ)&ALMASK
 RAOFF=          FRAMESZ-(1*SZREG)
 A0OFF=          FRAMESZ-(2*SZREG)
 A1OFF=          FRAMESZ-(3*SZREG)
 GPOFF=          FRAMESZ-(4*SZREG)

   NESTED(_sigsetjmp,FRAMESZ,ra)
        .set    noreorder
        .cpload t9               # set up gp for KPIC
        .set    reorder
        PTR_SUBU sp,FRAMESZ
        .cprestore GPOFF         # trigger t9/jalr
#a0 pts to environment, a1 determines whether to save mask
        REG_S   ra,RAOFF(sp)
        REG_S   a0,A0OFF(sp)
        jal     _sigstat_save    # saves mask or doesn't
        beq     v0,zero,setok
        j       err

   setok:
          REG_L   a0,A0OFF(sp)
          REG_L   ra,RAOFF(sp)
          PTR_ADDU sp,FRAMESZ
          j       setjmp         # now save the environment

   err:
REG_L   ra,RAOFF(sp)
PTR_ADDU sp,FRAMES
j       _cerror

        END(_sigsetjmp)
```

The *.cpload*/*.cprestore* is only used for generating KPIC code –– and tells the assembler to initialize, save, and restore the *gp*.

The following are relevant parts of *asm.h*:

```
#if (_MIPS_SIM == _MIPS_SIM_ABI32)
#define NARGSAVE        4
#define ALSZ            7
#define ALMASK          ~7
#endif
#if (_MIPS_SIM == _MIPS_SIM_ABI64)
#define NARGSAVE        0
#define ALSZ            15
#define ALMASK          ~0xf
#endif

#if (_MIPS_ISA == _MIPS_ISA_MIPS1 || _MIPS_ISA ==_MIPS_ISA_MIPS2)
#define SZREG           4
#endif

#if (_MIPS_ISA == _MIPS_ISA_MIPS3 || _MIPS_ISA == _MIPS_ISA_MIPS4)
#define SZREG           8
#endif

#if (_MIPS_ISA == _MIPS_ISA_MIPS1 || _MIPS_ISA == _MIPS_ISA_MIPS2)
#define REG_L   lw
#define REG_S   sw
#endif

#if (_MIPS_ISA == _MIPS_ISA_MIPS3 || _MIPS_ISA == _MIPS_ISA_MIPS4)
#define REG_L   ld
#define REG_S   sd
#endif

#if (_MIPS_SZINT == 32)
#define INT_L   lw
#define INT_S   sw
#define INT_LLEFT       lwl
#define INT_SLEFT       swl
#define INT_LRIGHT      lwr
#define INT_SRIGHT      swr
#define INT_ADD         add
#define INT_ADDI        addi
#define INT_ADDIU       addiu
#define INT_ADDU        addu
#define INT_SUB         sub
#define INT_SUBI        subi
#define INT_SUBIU       subiu
#define INT_SUBU        subu
#define INT_LL          ll
#define INT_SC          sc
#endif

#if (_MIPS_SZINT == 64)
#define INT_L   ld
#define INT_S   sd
#define INT_LLEFT       ldl
#define INT_SLEFT       sdl
#define INT_LRIGHT      ldr
#define INT_SRIGHT      sdr
```

```
#define INT_ADD         dadd
#define INT_ADDI        daddi
#define INT_ADDIU       daddiu
#define INT_ADDU        daddu
#define INT_SUB         dsub
#define INT_SUBI        dsubi
#define INT_SUBIU       dsubiu
#define INT_SUBU        dsubu
#define INT_LL          lld
#define INT_SC          scd
#endif

#if (_MIPS_SZLONG == 32)
#define LONG_L  lw
#define LONG_S  sw
#define LONG_LLEFT      lwl
#define LONG_SLEFT      swl
#define LONG_LRIGHT     lwr
#define LONG_SRIGHT     swr
#define LONG_ADD        add
#define LONG_ADDI       addi
#define LONG_ADDIU      addiu
#define LONG_ADDU       addu
#define LONG_SUB        sub
#define LONG_SUBI       subi
#define LONG_SUBIU      subiu
#define LONG_SUBU       subu
#define LONG_LL         ll
   #define LONG_SC         sc
   #endif

   #if (_MIPS_SZLONG == 64)
   #define LONG_L  ld
   #define LONG_S  sd
   #define LONG_LLEFT      ldl
   #define LONG_SLEFT      sdl
   #define LONG_LRIGHT     ldr
   #define LONG_SRIGHT     sdr
   #define LONG_ADD        dadd
   #define LONG_ADDI       daddi
   #define LONG_ADDIU      daddiu
   #define LONG_ADDU       daddu
   #define LONG_SUB        dsub
   #define LONG_SUBI       dsubi
   #define LONG_SUBIU      dsubiu
   #define LONG_SUBU       dsubu
   #define LONG_LL         lld
   #define LONG_SC         scd
   #endif

   #if (_MIPS_SZPTR == 32)
   #define PTR_L   lw
   #define PTR_S   sw
   #define PTR_LLEFT       lwl
   #define PTR_SLEFT       swl
   #define PTR_LRIGHT      lwr
   #define PTR_SRIGHT      swr
   #define PTR_ADD         add
```

```
#define PTR_ADDI        addi
#define PTR_ADDIU       addiu
#define PTR_ADDU        addu
#define PTR_SUB         sub
#define PTR_SUBI        subi
#define PTR_SUBIU       subiu
#define PTR_SUBU        subu
#define PTR_LL          ll
#define PTR_SC          sc
#endif

#if (_MIPS_SZPTR == 64)
#define PTR_L   ld
#define PTR_S   sd
#define PTR_LLEFT       ldl
#define PTR_SLEFT       sdl
#define PTR_LRIGHT      ldr
#define PTR_SRIGHT      sdr
#define PTR_ADD         dadd
#define PTR_ADDI        daddi
#define PTR_ADDIU       daddiu
#define PTR_ADDU        daddu
#define PTR_SUB         dsub
#define PTR_SUBI        dsubi
#define PTR_SUBIU       dsubiu
#define PTR_SUBU        dsubu
#define PTR_LL          lld
#define PTR_SC          scd
#endif
```

Chapter 4

Compilation Issues

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

Environment Variables

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

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

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

Command Line Switches

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

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

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

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

Fortran Switch Differences

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

−32 By definition.

−mips1 Generate code using the MIPS1 instruction set (the default for the 32−bit compiler).

−mips2 Generate code using the MIPS2 instruction set.

−66 Suppress extensions that enhance FORTRAN 66 compatibility.

−w66 Suppress only FORTRAN 66 compatibility warnings messages.

−usefpidx Force the compiler to use the floating point DO loop variable as the loop counter.

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

−64 By definition.

−mips3 Generate code using the MIPS3 instruction set.

−mips4 Generate code using the MIPS4 instruction set (the default for the 64−bit compiler on Power Challenge).

C Switch Differences

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

−32,−mips1,−mips2
As in 32−bit Fortran above.

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

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

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

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

−64,−mips3 −mips4
As in 64−bit Fortran above.

−A Add following *cpp* assertions.

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

−help Print list of possible options.

−keep Keep intermediate files.

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

−pcalist Run Power C Analyzer and keep listing.

−show1 Show phases but don’t have phases print additional info.

−showt Show time taken by each phase.

−unsigned Characters are unsigned by default.

−woffall Turn off all warnings.

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

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

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

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

Optimization Switches of the 64−Bit Compilers

In addition to the switches listed above, both the 64−bit Fortran and 64−bit C compilers support many more switches. These switches are used to control the types of optimizations the compilers perform. This section outlines the various optimizations that the 64−bit compilers can perform, and lists the switches used to control them.

The optimizations and switches fall into the following categories:

- General Optimizations

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

General Optimizations

−O0

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

−O1

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

−O2
−O

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

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

−O3

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

Floating Point Optimizations

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

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

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

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

−OPT:roundoff=n

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

- *roundoff=0*

  Do no transformations which could affect floating point results. This is the default for optimization levels *−O0* to *−O2*.
- *roundoff=1*

  Allow transformations with limited effects on floating point results. For roundoff, limited means that only the last bit or two of the mantissa are affected. For overflow (underflow), it means that intermediate results of the transformed calculation may overflow within a factor of two of where the original expression might have overflowed (underflowed). Note that limited effects may be less limited when compounded by multiple transformations.
- *roundoff=2*

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

  Re−association can have a substantial effect on the performanceof software pipelined loops by breaking recurrences. This is therefore the default for optimization level *−O3*.
- *roundoff=3*

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

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

−OPT:IEEE_arithmetic=n

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

- *IEEE_arithmetic=1*

  No degradation: do no transformations which degrade floating point accuracy from IEEE requirements. The generated code may use instructions like *madds* which provide greater accuracy than required by IEEE 754. This is the default.
- *IEEE_arithmetic=2*

  Minor degradation: allow transformations with limited effects on floating point results, as long as exact results remain exact. This option allows use of the MIPS 4 *recip* and *rsqrt* operations.
- *IEEE_arithmetic=3*

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

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

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

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

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

Consider another example:

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

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

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

–OPT:div_split[=(ON|OFF)]

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

–OPT:fast_complex[=(ON|OFF)]

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

–OPT:fast_exp[=(ON|OFF)]

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

–OPT:fast_sqrt[=(ON|OFF)]

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

–OPT:fold_reassociate[=(ON|OFF)]

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

–OPT:IEEE_comparisons[=ON]

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

It also disables optimizations which reverse the sense of a comparison, for example, turning *x < y* into *! (x*

>= y), since both *x<y* and *x>=y* may be FALSE if one of the operands is a *NaN.*

−OPT:recip[=(ON|OFF)]

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

−OPT:rsqrt[=(ON|OFF)]

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

−TARG:madd[=(ON|OFF)]

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

Miscellaneous Optimizations

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

−OPT:space

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

−OPT:alias=name

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

- *alias=any*

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

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

  For example, consider the code:

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

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

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

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

```
}
```

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

- *alias=restrict*

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

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

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

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

Loop Unrolling Optimizations

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

−OPT:unroll_times_max=n

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

−OPT:unroll_size=n

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

−OPT:unroll_bblimit=n

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

Global Code Motion

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

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

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

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

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

–GCM:array_speculation[=(ON|OFF)]

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

–GCM:pointer_speculation[=(ON|OFF)]

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

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

Consider an example:

```
...
if ( p->next != NULL ) {
sum += p->next->val;
}
...
If this option is set, the load of p->next->val can be moved before the if
 (it is through a potentially NULL pointer).
```

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

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

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

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

Software Pipelining

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

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

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

−SWP:=(ON|OFF)

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

−SWP:back_substitution[=(ON|OFF)]

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

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

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

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

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

−SWP:backtracks_max=n

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

−SWP:body_ins_count_max=n

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

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

−SWP:fix_recurrences[=(ON|OFF)]

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

−SWP:if_conversion[=(ON|OFF)]

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

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

```
c(i) = a(i)
ELSE
c(i) = b(i)
END IF
END DO
```

The loop body can be compiled for MIPS4 as:

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

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

−SWP:interleave_reductions[=(ON|OFF)]

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

```
DO i=1,n
   sum = sum + a(x)
END DO
```

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

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

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

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

−SWP:trip_count_min=n

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

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

Target Environment Issues

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

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

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

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

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

−TENV:X=n

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

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

Data Alignment Options

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

−align8 / −align16 / −align32 / −align64

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

−TENV:align_aggregates=n

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

−TENV:misalignment=n

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

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

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

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

Compilation Messages

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

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

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

Two examples illustrate this:

C:

```
%cc −32bit test.c

cfe: Error: test.c, line 4: Syntax Error
```

Fortran:

```
% f77 −32bit err.f

Error on line 6 of err.f: syntax error
```

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

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

C:

```
%cat test.c

#include <stdio.h>
main ()
{
      printf ("hhelp")

}

%cc −64bit test.c
"test.c", line 4: error(1065): expected a ";"

}
^
```

Fortran:

```
%cat err.f
```

```
   program test
c
c
2 write(6,8)
8 formatr('This is a test')
   end

% f77 -64bit err.f
### 8 formatr('This is a test')
### in line 6 procedure TEST of file err.f ###
### error Statement unrecognizable as any known statement type. ###
### error Label "8" is not the label of a format statement. ###

2 errors in file /tmp/ctmi.AAAa000Y
```

Important Warning Messages

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

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

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

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

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

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

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

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

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

Linking Your Application

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

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

Libraries

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

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

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

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

Chapter 5
Runtime Issues

This chapter outlines why your 32−bit and 64−bit applications may run differently, due both to compiler differences and to architectural modes. It describes the Performance and Precise Exception Modes of the R8000 microprocessor architecture and how they affect the calculations of applications. This chapter also briefly outlines a methodology to bring up and debug applications.

Runtime Differences

Your 64−bit application may produce slightly different floating point calculations on the R8000 than on its 32−bit counterpart. There can be a variety of causes for this. These include reassociation of operations by optimizations, algorithm changes in libraries and hardware changes.

Reassociation of Operations by Optimizations

The order in which equivalent floating point expressions are executed can cause differences in their results. The 32−bit and 64−bit compiler systems perform different optimizations which can cause reordering of instructions leading to slightly different results. The compilers may also perform operation reductions which can affect the results.

Algorithm Changes in Libraries

The 64−bit compiler comes with new math libraries which use different algorithms than those used with the 32−bit compiler to improve their performance. The values which they return can cause potentially noticeable differences in application results.

Hardware Changes

The R8000 microprocessor includes four floating point multiply/add / subtract instructions which allow two floating point computations to be performed with one instruction. The intermediate result is calculated to infinite precision and is not rounded prior to the addition or subtraction. The result is then rounded according to the rounding mode specified by the instruction. This can yield slightly different calculations than a multiply instruction (which is rounded) and an add instruction (which is rounded again).

The R8000 microprocessor architecture also defines two execution environments which can affect your application if it generates floating point exceptions such as underflow. Performance Mode enhances the execution speed of floating point applications, by rounding denormalized numbers to zero and allowing the hardware to trap exceptions imprecisely. Precise Exception Mode, on the other hand, is fully compatible to the existing MIPS floating point architecture.

It should be emphasized that running in Performance Mode does not affect those applications which don’t cause floating point exceptions.

A program, *fpmode*, allows you to run your application in either Performance (imprecise) or Precise Mode. Its usage is as follows:

```
%fpmode precise   commandargs
```

or

```
%fpmode imprecise commandargs
```

A full discussion of the Extended MIPS Floating Point Architecture is provided as a reference.

Extended MIPS Floating−Point Architecture

The MIPS architecture fully complies with the ANSI/IEEE Standard 754−1985, *IEEE Standard for Binary Floating−Point Arithmetic*. Most application programs utilize only a fraction of all the features required by the Standard. These applications can gain additional performance if executed in an environment that supports only those features of the Standard that are actually necessary for the correct execution of the application. The Extended MIPS Floating−Point Architecture defines two execution environments:

- Performance Mode enhances the execution speed of most applications by rounding denormalized numbers to zero and by allowing the hardware to trap exceptions imprecisely. This mode requires compiler and library support to fully comply with the Standard.

  In Performance Mode, the hardware and operating system are relieved of the requirements to precisely trap floating−point exceptions and to compute using denormalized operands. This mode is defined in such a way that it is adequate for a majority of application programs in use today, yet it can also be used in conjunction with compiler and library support to fully implement the Standard in the future.

  Performance Mode improves the floating−point execution speed of processors. On the R4000, Performance Mode enables flushing operands to zero, thus avoiding the software emulation overhead of denormalized computations. On the R8000, Performance Mode enables floating−point instructions to execute out−of−order with respect to integer instructions, improving performance by a factor of two or more.

  Performance Mode is the standard execution environment on R8000 based Power Challenge systems.
- Precise Exception Mode fully complies with the Standard and is compatible in every way to the preexisting MIPS floating−point architecture.

  In Precise Exception Mode the responsibility for compliance lies entirely with the hardware and operating system software; no compiler support is assumed. Since there is no information about the application, the hardware must assume the most restrictive features of the Standard applies at all times. The result is lost performance opportunities on applications that utilize only a subset of the features called for by the Standard.

Performance Mode

The purpose of this section is to define Performance Mode and explain why it is necessary and desirable.

Background

The IEEE Standard defines floating−point numbers to include both normalized and denormalized numbers. A denormalized number is a floating−point number with a minimum exponent and a nonzero mantissa which has a leading bit of zero. The vast majority of representable numbers in both single and double precision are normalized numbers. An additional small set of very tiny numbers (less than 2^{-126} (~10^{-38}) in single precision, less than 2^{-1022} (10^{-308}) in double precision are represented by denormalized numbers. The importance of approximating tiny real values by denormalized numbers, as opposed to rounding them to zero, is controversial. It makes no perceptible difference to many applications, but some algorithms need them to guarantee correctness.

Figure 5−1 shows pictorially the IEEE definition of floating−point numbers. Only the positive side of the real number line is shown, but there is a corresponding negative side also. The tick marks under the real number line denote example values that can be precisely represented by a single or double precision binary number. The smallest representable value larger than zero is *minD*, a denormalized number. The smallest normalized number is *minN*. The region between zero and just less than *minN* contains tiny values. Larger values starting with *minN* are not tiny.

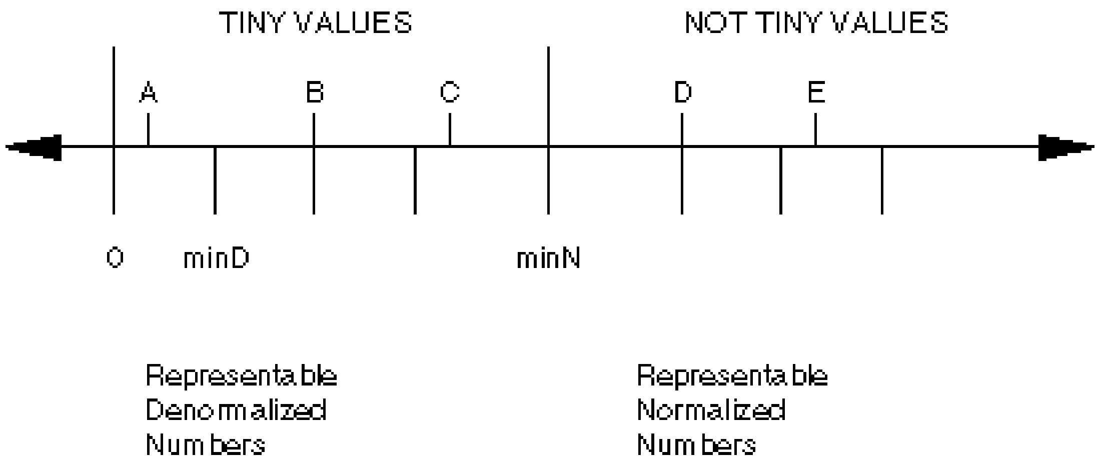

Figure 5−1 Floating Point Numbers

The different cases that must be considered are represented by the values A−E. According to the IEEE Standard, the behavior of an operation that produces these result values is defined as follows.

Table 5−1 Operation Results According to IEEE Standard

Value	Result	Flags
A:TooSmall	rnd(A)	U=1,I=1
B:ExactDenorm	B	U=1, I=0 if Enable U=U=0, I=0 if EnableU=0
C:InexactDenorm	rnd(C)	U=1,I=1
D:ExactNorm	D	U=0,I=0
E:InexactNorm	rnd(E)	U=0, I=1

The flags *U* and *I* abbreviate Underflow and Inexact, respectively. The function *rnd()* rounds the operand to the nearest representable floating point number based on the current rounding mode, which can be round−to−zero, round−to−nearest, round−to−plus−infinity, and round−to−minus−infinity. For example, *rnd(A)* is either zero or *minD*. A trap occurs if a flag is set and the corresponding enable is on. For example, if an operation sets *I*=1 and *EnableI*=1, then a trap should occur. Note that there is a special case for representable tiny values: the setting of the *U* flag depends on the setting of its enable.

Supporting denormalized numbers in hardware is undesirable because many high performance hardware algorithms are designed to work only with normalized numbers, and so a special case using additional hardware and usually additional execution time is needed to handle denormalized numbers. This special case hardware increases the complexity of the floating−point unit and slows down the main data path for normalized numbers, but is only rarely used by a few applications. Therefore most processor designers have generally deemed it not cost effective to support computations using denormalized numbers in hardware. To date no implementation of the MIPS architecture supports denormalized number in hardware.

Computations using denormalized numbers can also be supported by software emulation. Whenever a floating−point operation detects that it is about to either generate a denormalized result or begin calculating using a denormalized operand, it can abort the operation and trap to the operating system. A routine in the kernel, called *softfp*, emulates the computation using an algorithm that works correctly for denormalized numbers and deposits the result in the destination register. The operating system then resumes the application program, which is completely unaware that a floating−point operation has been emulated in software rather than executed in hardware. Emulation via *softfp* is the normal execution environment on all IRIX platforms today.

The problem with the software emulation approach is two−fold. Firstly, emulation is slow. Computations using denormalized operands frequently generate denormalized results. So, once an application program creates a denormalized intermediate value, the execution speed of the application drastically slows down as it propagates more and more denormalized intermediate results by software emulation. If the application truly requires representation of denormalized numbers in order to perform correctly, then the slowdown is worthwhile. But in many cases the application also performs correctly if all the denormalized intermediate results were rounded to zero. For these applications software emulation of denormalized computations is just a waste of time.

The second problem with software emulation is that it demands precise floating−point exceptions. In order for *softfp* to substitute the result of an arbitrary floating−point instruction, the hardware must be capable of aborting an already−executing floating−point instruction based on the value of the input operand or result, aborting any subsequent floating−point instruction that may already be in progress, and trapping to the operating system in such a way that the program can be resumed. Providing precise exceptions on floating−point operations is always difficult since they take multiple cycles to execute and should be overlapped with other operations. It becomes much more difficult when, to achieve higher performance, operations are executed in a different order than that specified in the program. In this case instructions logically after a floating−point operation that needs to be emulated may have already completed execution! While there are known techniques to allow *softfp* to emulate the denormalized operation, all these techniques require considerable additional hardware.

Performance Mode Definition

In defining a new floating−point execution environment there are several goals:

- Give sufficient latitude to facilitate the design of all conceivable future high performance processors.
- Fully comply with the IEEE Standard via a combination of compiler, library, operating system and hardware.
- Preserve the correct operation of a broad subset of existing applications compiled under the preexisting floating−point environment (which we now call Precise Exception Mode).
- Provide a software−only solution to retrofit the new mode on existing hardware.

The first goal is important because we do not want to be changing floating−point architectures with every implementation. The second goal is important because we want to continue to say we have "IEEE arithmetic" machines. The third goal gives our customers a smooth transition path. The fourth goal lets our customers upgrade their old machines.

Performance mode is defined by omitting denormalized numbers from the IEEE Standard and by deleting the requirement to precisely trap floating−point exceptions. Referring to Table 5−2, the behavior of an operation that produces result values A−E in Performance Mode is defined as follows.

Table 5−2 Operation Results Using Performance Mode

Value	Input	Result	Flags
A: TooSmall	–	0 or minN	U=1, I=1
B: ExactDenorm	0 or min	0 or minN	U=1, I=1
C: InexactDenorm	–	0 or minN	U=1, I=1
D: ExactNorm	D	D	U=0, I=0
E: InexactNorm	–	rnd(E)	U=0, I=1

Tiny results are mapped to either zero or the minimum normalized number, depending on the current Rounding Mode. Note that the inexact flag *I* is set in case *B* because although there is an exact denormalized representation for that value, it is not being used. Denormalized input operands, *B*, are similarly mapped to zero or *minN*. Note that there are no inexact inputs since they cannot be represented. The normalized cases are identical to those in Precise Exception mode.

All IEEE Standard floating−point exceptions are trapped imprecisely in Performance Mode. Regardless of whether the exceptions are enabled or disabled, the result register specified by the offending instruction is unconditionally updated as if all the exceptions are disabled, and the exception conditions are accumulated into the flag bits of the *FSR*, the floating point control and status register.

There are two classes of exceptions in Performance Mode. If any flag bit (invalid operation, division by zero, overflow, underflow, inexact) and its corresponding enable bit are both set, then an imprecise trap occurs at or after the offending instruction up to the next trap barrier. In addition, if *FS*=0 (*FS* is a special control bit in the *FSR*) then an imprecise trap occurs when a tiny result that would be represented as a denormalized number gets mapped into zero or minN. *FS*=0 also causes an imprecise trap if an input operand is a denormalized number that gets trapped into zero or *minN*.

A floating−point trap barrier is defined by a code sequence that begins with an instruction moving the *FSR* to an integer register and concludes with an instruction that uses the integer register containing the *FSR* contents. Any number of other instructions are allowed in between as long as they are not floating−point computation instructions (that is, they cannot set flag bits). All imprecise floating−point traps that occur on behalf of an instruction before the barrier are guaranteed to have occurred before the conclusion of the barrier. At the conclusion of the barrier the flag bits accurately reflect the accumulated results of all floating point instructions before the barrier. The floating−point barrier is defined in this way to give implementations maximum flexibility in overlapping integer and floating−point operations serialization of the two units is deferred as late as possible to avoid performance loss.

The cause bits of the *FSR* present a serious problem in Performance Mode. Ideally they should contain the result of the latest floating−point operation. However, this may be very difficult or expensive to implement when floating−point instructions are issued or even completed out of order. In order to maximize the opportunity for correctly running existing binaries and yet retain full flexibility in future out−of−order implementations, the cause bits of the *FSR* are defined to be cleared by each floating−point operation. Future applications, however, should avoid looking at the cause bits, and instead should use the flag bits.

The selection of Performance or Precise Exception Mode is defined as a protected or kernel−only operation. This is necessary for several reasons. When executing existing binaries that operate correctly in Performance Mode, we do not want the program to accidently go into Precise Exception Mode. Since existing programs regularly clear the entire *FSR* when they want to clear just the rounding mode bits, Performance Mode cannot be indicated by setting a bit in the *FSR*. On the other hand, existing programs that must run in Precise Exception Mode must not accidently go into Performance Mode. Thus Performance Mode cannot be indicated by clearing a bit in the *FSR* either. We cannot use a new user−accessible floating−point control register to indicate Performance Mode because when a new program running on an existing processor that does not understand Performance Mode writes to this nonexisting control register, it is undefined what happens to the floating−point unit. Finally, on the R8000 there are implementation restrictions on what instructions may proceed and follow a mode change, so such changes can only be done safely by the kernel.

R8000 and R4400 Implementations

The R4000 already made a step in the direction of Performance Mode by defining the *FS* bit in the *FSR*, the floating−point control and status register. When *FS* is set, denormalized results are flushed to zero or *minN* depending on current Rounding Mode instead of causing an unimplemented operation exception. This feature eliminates the most objectionable aspect of Precise Exception Mode, namely the slow propagation of denormalized intermediate results via *softfp*. However, it does not eliminate the need to precisely trap floating−point exceptions because denormalized input operands must still be handled by *softfp*.

The R8000 extends the R4000 floating−point architecture to include another control bit whose states are labeled *PERF* and *PREX*, for Performance Mode and Precise Exception Mode, respectively. In Performance Mode the R8000 hardware does the following:

Table 5−3 R8000 Performance Mode

Value	Input	Result	Flags
A:TooSmall	–	0 or minN	U=1, I=1 E=1 if FS=0
B:ExactDenorm	0	0 or minN	U=1, I=1 E=1 if FS=0
C:InexactDenorm	–	0 or minN	U=1, I=1 E=1 if FS=0
D:ExactNorm	D	D	U=0, I=0
E:InexactNorm	–	rnd(E)	U=0, I=1

The *E* bit, which becomes sticky in Performance Mode, signifies that a denormalized number was mapped to 0 or *minN*. Note that the R8000 can only flush denormalized input operands to zero, as opposed to either zero or *minN*. This deviation is unfortunate but unlikely to be noticeable and is not scheduled to be fixed.

In Precise Exception Mode the R8000 hardware does the following:

Table 5−4 R8000 Precise Exception Mode

Value	Input	Result	Flags

A:TooSmall	–	trap	U=1, I=1
B:ExactDenorm	trap	trap	U=1, I=1
C:InexactDenorm	–	trap	U=1, I=1
D:ExactNorm	D	D	U=0, I=0
E: InexactNorm	–	rnd(E)	U=0, I=1

Note that unlike the R4400, the R8000 totally ignores the *FS* bit in this case and relies on *softfp* to emulate the result. This simplification degrades the performance of Precise Exception Mode but does not alter the results.

Performance Mode is retrofitted on the R4400 by enhancing the kernel and *softfp*. The emulation of Performance Mode deviates from the definition in that the cause bits of the *FSR* are not cleared by every floating–point operation, but instead continue to be updated based on the result of the operation. This deviation is necessary to achieve acceptable performance.

Full IEEE Compliance in Performance Mode

Full IEEE Standard compliance including precise exceptions and support for denormalized computations is possible in Performance Mode with help from the compiler. Although most applications never need it, some programming languages (for example, Ada) require more precision in exception reporting than what the hardware provides in Performance Mode. Also, a small number of algorithms really do need to perform computations on denormalized numbers to get accurate results.

The concept behind achieving precise exceptions in Performance Mode relies on two observations. Firstly, a program can be divided into a sequence of blocks, each block containing a computation section followed by an update section. Computation sections can read memory, calculate intermediate results, and store to temporary locations which are not program variables, but they cannot modify program visible state. Update sections store to program visible variables, but they do not compute. Floating–point exceptions can only occur on behalf of instructions in computation sections, and can be confined to computation sections by putting a floating–point exception barrier at the end of computation sections.

Secondly, it is always possible to partition the computation and update sections in such a way that the computation sections are infinitely reexecutable. We call such computation sections Idempotent Code Units.

Intuitive, an ICU corresponds to the right hand side of an assignment statement, containing only load and compute instructions without cyclic dependencies. In practice ICUs can also contain stores if they spill to temporary locations. As long as the input operands remain the same, the result generated by an ICU remains the same no matter how many times the ICU is executed. We achieve precise exceptions in Performance Mode by compiling the program (or parts of the program) into blocks of the following form (registers are marked with %):

```
restart = current pc      #restart point
. . .                     #Idempotent Code Unit
%temp = FSR               #trap barrier
%add %r0 = %r0 + %temp    #end of trap barrier
      Fixup:nop                 #break point inserted here
      . . .                     #update section
```

A map specifying the locations of all ICU's and their Fixup point is included in the binary file, and the program is linked with a special floating–point exception handler.

When a floating–point exception trap occurs, the handler switches the processor to Precise Exception mode, inserts a break point at location Fixup, and re–executes the ICU by returning to the program counter in *%restart*. This time the exception(s) are trapped precisely, and denormalized computations can be emulated. When the program reaches the break point inserted at *Fixup*, another trap occurs to allow the handler to remove the break point, reinsert the nop, and return to Performance Mode.

Application Bringup and Debugging

The first step in bringing up applications is to compile and run them at the lowest optimization level. Once a stable baseline of code is established, you can compare it with code that does not run, to isolate problems. This methodology is expanded upon as follows:

- Use the source base of a 32−bit working application which is compiled *−mips2* (compiled with the *ucode* compilers).
- Compile *−64 −mips4 −g* and fix source problems. Refer to other sections in this Guide for help on general porting issues.
- Run the *−64 −mips4 −g* binaries on a PowerChallenge, R8000 IRIX 6 system. At this point the ucode 32−bit binaries, should also be run, side by side, on the PowerChallenge system to help isolate where the problems are creeping in.
- Repeat the previous step, going up the optimization scale.
- Compile *−64*, *−mips4*, −O3 and tune applications on Power Challenge to get the best perfromance. Refer to Chapter 6 in this Guide for help on tuning for the R8000.

The good news is that if you can get everything working *−64 −mips4 −g*, then you have a 64−bit baseline with which to compare non−working code. The MIPSpro compiler allows you to link object files compiled at different optimizations. By repeatedly linking objects from the working baseline, with those of the non−working set and testing the resulting application, you should be able to identify which objects from the non−working set are bad.

Once you have narrowed down your search to a small area of code, use the *dbx* debugger. You can then compare the variables and program behavior of your working application against the variables and program behavior of the code that fails. By isolating the problem to a line of code, you can then find the cause of the failure.

Chapter 6

Performance Tuning for the R8000

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

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

Software Pipelining

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

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

Why Software Pipelining Delivers Better Performance

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

```
DO i = 1, n
v(i) = v(i) + X * w(i)
ENDDO
```

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

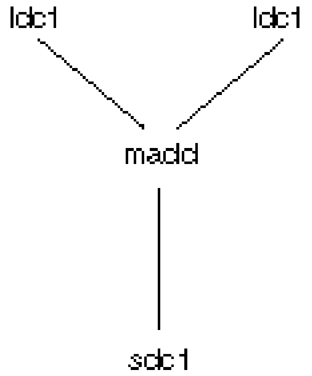

Figure 6–1 A Simple DAXPY Implementation

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

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

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

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

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

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

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

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

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

4: t1 = ldc1    t2 = ldc1   t7 = madd t1 X t2
5: t4 = ldc1    t5 = ldc1   t8 = madd t4 X t5
```

```
6: sdc1 t3      sdc1 t6     bne L1

DONE:
```

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

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

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

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

winddown:
1: sdc1 t7      sdc1 t8     br ALLDONE

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

ALLDONE:
```

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

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

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

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

compensation n-1:
```

```
winddown:
      ...
      goto done

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

      rx = ry
      ...
      goto winddown
      ...

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

simple_loop:

      do i = i,n
         original_loop
      enddo

done:
```

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

Looking at the Code Produced by Software Pipelining

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

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

This creates an annotated *.s* file

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

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

Example 1: Output from Using the –S Compiler Switch

```
%cat test.f
          program test
c
c
          real*8 v(1000),w(1000)
          n = 1000
          x = 5
          DO j = 1, n
             DO i = 1, 120
                 v(i) = v(i) + x * w(i)
             ENDDO
```

```
            ENDDO
         ENDDO
         end
%f77 -64 -mips4 -O3 -S test.f
%grep swps test.s
#<swps>
#<swps> Pipelined loop line 11 steady state
#<swps>
#<swps>        Not unrolled before pipelining
#<swps>      3 cycles per iteration
#<swps>      8 flops       ( 66% of peak) (madds count as 2
#<swps>      4 flops       ( 66% of peak) (madds count as 1
#<swps>      4 madds       ( 66% of peak)
#<swps>      3 mem refs    ( 50% of peak)
#<swps>      2 integer ops ( 33% of peak)
#<swps>      6 short trip threshold
#<swps>
```

This shows that the inner loop starting at line 11 was software pipelined. No unrolling was done before pipelining. It used 3 cycles for each loop iteration and calculated the statistics as follows:

- If each *madd* counts as two floating point operations, the R8000 can do four floating point operations per cycle (two *madds*), so its peak for this loop is 12. Eight floating point references are 8/12 or 66% of peak. The figure for *madds* is likewise calculated.
- If each *madd* counts as one floating point operation, the R8000 can do two floating point operations per cycle, so its peak for this loop is 6. Four floating point operations are 4/6 or 66% of peak.
- The R8000 can do two memory operations per cycle, so its peak for this loop is 6. Three memory references are 3/6 or 50% of peak.
- The R8000 can do two integer operations per cycle, so its peak for this loop is 6. Two integer operations are 2/6 or 33% of peak. The statistics also point out that loops less than 6 iterations would not go through the software pipeline replication area, but would be executed in the simple_loop section shown above.

How to Diagnose What Went Wrong

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

1. Did it software pipeline at all?

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

2. How well did it pipeline ?

   Look at statistics in the *.s* file.

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

   Your generated code may not have the operations you expected.

4. Think about how you would hand code it.

   What operations did it need?

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

   Is it very different? Why?

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

   Sometimes this is a compiler error.

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

KAP SWP Conflicts

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

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

to your compilation switches.

Matrix Multiply - A Tuning Case Study

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The Effect of Bellows Stalls

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

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

Example 2: Single Precision Loops

SAXPY (Single precision A times X Plus Y)

The simple SAXPY code below runs at 58% or 72% of theoretical maximum, depending on how the arrays *x* and *y* are aligned relative to each other.

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

return
end
```

The alignment was controlled by putting the arrays in a common block in the calling main program.
common x,y yielded 58 % of max
common x,idum,y yielded 72 % of max

In the faster version, *x(i)* and *y(i)* are located in opposite banks because they are separated by the dummy variable, *idum*, in the declaration.The difficulty with this single–precision example is that the cache is interleaved at double word boundaries. As opposed to double precision accesses, consecutive single precision accesses do not go to alternate banks. Thus *y(i)* and *y(i+1)* are in one bank, while *y(i+2)* and *y(i+3)* are in the other bank.

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

y(i), y(i+2) in one cycle
y(i+1), y(i+3) in a different cycle

This trick can be applied to any number of arrays and the normal bellows effect should take care of unknown base alignment of the arrays *y(i)*, *z(i)*:
y(i+2), z(i+2)
y(i+1), z(i+1)
...

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

The following code gets 99.7% of theoretical maximum.

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

return
end
```

Example 3: Double Precision Loops

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

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

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

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

10       continue
20    continue
```

This is because the array is laid out in the secondary cache like this:

Table 6–1 Fortran Array Layout in Secondary Cache

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

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

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

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

The inner loop elements are in alternating banks now and the bellows stalls are avoided.

Table 6−2 Fortran Arrays in Aternating Banks

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

Use of the IVDEP Directive

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

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

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

Now suppose you have a loop like:

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

The compiler has no way to know that

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

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

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

for all values of *i*.

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

Placed above a loop, the statement:

```
cdir$ ivdep
```

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

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

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

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

cdir$ ivdep

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

−O3 −mips4 −non_shared

to compile.

For example

```
          subroutine m(a,nn,b,ldb,ix,kb,m,n,p,alpha,id1,id2,
       +          lda,aa)
c
          implicit double precision(a-h,o-z)
          real*8 alpha,beta
          dimension a(*),b(ldb,n),ix(nn),aa(lda,*)
          integer p
c
          double precision t
c
          do  100 j=1,(n-1),2
            idc = ix(id1+j) + id2
            idc1 = ix(id1+j+1) + id2
            do 50 k=1,(p-3),4
              t00 = b(k,j)
              t01 = b(k,j+1)
              t10 = b(k+1,j)
              t11 = b(k+1,j+1)
              t20 = b(k+2,j)
              t21 = b(k+2,j+1)
              t30 = b(k+3,j)
              t31 = b(k+3,j+1)

cdir$ ivdep
              do 30 i=1,m
                a(idc+i) = a(idc+i) + aa(i,k)*t00 +
&             aa(i,k+1)*t10 + aa(i,k+2)*t20 + aa(i,k+3)*t30
                a(idc1+i) = a(idc1+i) +
&             aa(i,k)*t01 +
&             aa(i,k+1)*t11 + aa(i,k+2)*t21 + aa(i,k+3)*t31
   30         continue
   50       continue
            do 80 k=k,p
              t00 = b(k,j)
              t01 = b(k,j+1)
              do i =1,m
                a(idc+i) = a(idc+i) + aa(i,k)*t00
                a(idc1+i) = a(idc1+i) + aa(i,k)*t01
              enddo
   80       continue

   100    continue

          do j=j,n
               idc = ix(id1+j) + id2
               do k=1,p
                 t00 = b(k,j)
                 do i=1,m
                   a(idc+i) = a(idc+i) + aa(i,k)*t00
```

```
            enddo
          enddo
        enddo
      return
      end
```

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

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

−OPT:unroll_size=1000

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

Chapter 7

Miscellaneous FAQ

This chapter summarizes important concepts found throughout this manual in the form of frequently asked questions and their answers.

Q. Why can't I link my 32−bit application with a 64−bit library?

A. There are differences in the subroutine calling interface.

Q. How Do I disable KAP ?

A. To disable KAP add the following to your compile line *−WK,−o=0,−ro=0,−so=0*

Q. How can I see what KAP does to my code?

A. To view the file add the following to your compile line *−WK,−cmp=file.cmp*

Q. My */tmp* directory becomes full when I do a compile. What should I do ?

A. *%setenv TMP_DIR /directory* on a free partition.

Q. How do I know which compiler is being invoked ?

A. *cc −show* will show each of the components as they are being run.

Q. My 64−bit shared application gets mysterious messages from *rld* at runtime.What is the problem ?

A. It's possible that you are linking with 32−bit *.so's*. Check and reset your *_RLD_ROOT* environment variable.

Q. How can I avoid always setting *−32* or *−64* on my compilation command line ?

A. *%setenv SGI_ABI −32* or
%setenv SGI_ABI −64.

Q. How do I know if my code software pipelined OK ?

A. Compile your file (say *foo.f*) *−64 −O3 −mips4 −S*. Then *grep #<swp foo.s#*
#<swpf will be printed for loops that failed to software pipeline.
#<swps will be printed for statistics and other info about the loops that did software pipeline.

Q. I compiled *−O*, but my code did not software pipeline. Why not?

A. Software pipelining occurs at *−O3*. *−O* implies *−O2*.

Q. Ok, I now compiled *−O3*. Why didn't my code software pipeline?

A. Does it have a call or branch in the inner loop?
Is the inner loop too long?
Does the inner loop execute too few iterations?
Are there divide recurrences in your inner loop?

Q. What predefine should my source code look at to know if its being compiled 32−bit or 64−bit.

A. One way is:
#if (_MIPS_SZLONG == 64)

Q. How can I force a large loop to be unrolled by the compiler?

A Add *−OPT:unroll_size=1000* to your command line?

Q. Why does my application gets different floating point numbers in 64−bit mode?

A. If its compiled with optimization, the order of operations may different resulting in slightly different calculations due to rounding. *madd* instructions also round differently than a multiply and an add. The math libraries use different algorithms.

Q. Why doesn't 64−bit *printf* print my floating point values?

A. C Programs using *printf()* to print out floating point (or double) values must include stdio.h.

```
%cat example.c
main()
{
   float f = 3.14;
 double d = 2.718;
 printf ("%f\n",f);
 printf ("%e\n",d);
}
%cc −32 example.c
%a.out
3.140000
2.718000e+00

%cc −64 example.c
%a.out
0.000000
1.302121e−315
%cat example_fix.c
#include <stdio.h>
main()
{
float f = 3.14;
double d = 2.718;
printf ("%f\n",f);
printf ("%e\n",d);
}
% cc −64 example_fix.c
%a.out
3.140000
2.718000e+00
```