MIPSpro™ N32/64 Compiling and Performance Tuning Guide

New Features in This Manual

Information regarding the use of multiple DSOs on one page of memory has been added to Chapter 3, "Using Dynamic Shared Objects", page 33.

Record of Revision

Version	Description
	August 1994 Original Printing.
7.3	April 1999 Adds a description of an optimization that reorders parts of an executable program. Updates information on searching for DSOs.
009	September 2002 Updated to support the MIPSpro 7.4 release which runs on IRIX operating systems, version 6.5 and later.
010	June 2003 Updated to support the MIPSpro 7.4.1m release and to add tuning information for DSOs.

Contents

Figures

Tables

About This Guide

This guide describes the components of MIPSpro compiler system, other programming tools and interfaces, and dynamic shared objects (DSO). It also explains ways to improve program performance.

The SGI compiler systems produce either new 32-bit (n32) object code, 64-bit object code, or old 32-bit (o32) object code. This guide describes the MIPSpro compilers that produce new 32-bit and 64-bit object code. For additional information about n32 and 64-bit compilation, see the *MIPSpro 64-Bit Porting and Transition Guide* and the *MIPSpro N32 ABI Handbook*. For information about compilers that produce old 32-bit objects, refer to the *MIPS O32 Compiling and Performance Tuning Guide*.

This guide is written for anyone who wants to program effectively using the MIPSpro compilers. It is written for a reader who is familiar with the IRIX (or UNIX) operating system and a programming language such as C or Fortran. This guide does not explain how to write and compile programs.

This guide does not cover all of the differences between n32, 64, and o32 compilation modes. Refer to the *MIPSpro 64-Bit Porting and Transition Guide* and the *MIPS O32 Compiling and Performance Tuning Guide* for information about the differences between these modes, language implementation differences, source code porting, compilation issues, and run-time execution.

Be sure to read the *Release Notes* for your compiler. They contain important information about this release of the MIPSpro compiler system.

Related Publications

The following documents contain information that may be useful as you use the MIPSpro compilers:

- *MIPSpro Fortran Language Reference Manual, Volume 1*

- *MIPSpro Fortran Language Reference Manual, Volume 2*

- *MIPSpro Assembly Language Programmer's Guide*

- *Application Programmer's I/O Guide*

- *SpeedShop User's Guide*

- *ProDev WorkShop: Debugger User's Guide*

- *ProDev WorkShop: Performance Analyzer User's Guide*

- *ProDev WorkShop: Tester User's Guide*

- *dbx User's Guide*

- *Origin 2000 and Onyx2 Performance Tuning and Optimization Guide*

- *MIPSpro N32 ABI Handbook*

- *MIPSpro 64-Bit Porting and Transition Guide*

- *Getting Started with XFS Filesystems*

- *System V Applications Binary Interface—Revised First Edition*. Prentice Hall, ISBN 0-13-880410-9

- *System V Application Binary Interface MIPS Processor Supplement*. Prentice Hall, ISBN 0-13-880170-3.

Related Fortran Publications

The following commercially available reference books are among those that you should consult for more information on the history of Fortran and the Fortran language itself:

- Adams, J., W. Brainerd, and J. Martin. *Fortran 95 Handbook : Complete ISO/ANSI Reference*. MIT Press, 1997. ISBN 0262510960.

- Chapman, S. *Fortran 90/95 for Scientists and Engineers*. McGraw Hill Text, 1998. ISBN 0070119384.

- Chapman, S. *Introduction to Fortran 90/95*. McGraw Hill Text, 1998. ISBN 0070119694.

- Counihan, M. *Fortran 95 : Including Fortran 90, Details of High Performance Fortran (HPF), and the Fortran Module for Variable-Length Character Strings*. UCL Press, 1997. ISBN 1857283678.

- Gehrke, W. *Fortran 95 Language Guide*. Springer Verlag, 1996. ISBN 3540760628.

- International Standards Organization. *ISO/IEC 1539–1:1997, Information technology — Programming languages — Fortran*. 1997.

- Metcalf, M. and J. Reid. *Fortran 90/95 Explained*. Oxford University Press, 1996. ISBN 0198518889.

Conventions

The following conventions are used throughout this document:

Convention	Meaning
command	This fixed-space font denotes literal items such as commands, files, routines, path names, signals, messages, and programming language structures.
variable	Italic typeface denotes variable entries and words or concepts being defined.
user input	This bold, fixed-space font denotes literal items that the user enters in interactive sessions. (Output is shown in nonbold, fixed-space font.)
[ ]	Brackets enclose optional portions of a command or directive line.
...	Ellipses indicate that a preceding element can be repeated.

Obtaining Publications

You can obtain SGI documentation in the following ways:

- See the SGI Technical Publications Library at http://docs.sgi.com. Various formats are available. This library contains the most recent and most comprehensive set of online books, release notes, man pages, and other information.

- If it is installed on your SGI system, you can use InfoSearch, an online tool that provides a more limited set of online books, release notes, and man pages. With an IRIX system, select **Help** from the Toolchest, and then select **InfoSearch**. Or you can type infosearch on a command line.

- You can also view release notes by typing either grelnotes or relnotes on a command line.

- You can also view man pages by typing man *title* on a command line.

Reader Comments

If you have comments about the technical accuracy, content, or organization of this publication, contact SGI. Be sure to include the title and document number of the publication with your comments. (Online, the document number is located in the front matter of the publication. In printed publications, the document number is located at the bottom of each page.)

You can contact SGI in any of the following ways:

- Send e-mail to the following address:

  techpubs@sgi.com

- Use the Feedback option on the Technical Publications Library Web page:

  http://docs.sgi.com

- Contact your customer service representative and ask that an incident be filed in the SGI incident tracking system.

- Send mail to the following address:

  Technical Publications
  SGI
  1600 Amphitheatre Parkway, M/S 535
  Mountain View, California 94043–1351

- Send a fax to the attention of "Technical Publications" at +1 650 932 0801.

SGI values your comments and will respond to them promptly.

About the MIPSpro Compiler System

The MIPSpro compiler system consists of a set of components that enable you to create new 32-bit and 64-bit executable programs (as well as old 32-bit executables) using languages such as C, C++, and Fortran.

A new 32-bit mode, n32, was introduced with the IRIX 6.1 operating system. This new 32-bit mode has the following features:

- Full access to all features of the hardware

- MIPS III and MIPS IV instruction set architecture (ISA)

- Improved calling convention

- 32 64-bit floating-point registers

- 32 64-bit general purpose registers

- Dwarf debugging format

The new 32-bit mode (n32) provides better performance than the old 32-bit mode available in IRIX releases prior to 6.1. When you compile with the –n32 option, the chip executes in 64-bit mode and the software restricts addresses to 32 bits. For more information about n32, refer to the *MIPSpro N32 ABI Handbook*.

In addition, the MIPSpro compiler system:

- Uses executable and linking format (ELF) for object files. ELF is the format specified by System V Release 4 Applications Binary Interface (SVR4 ABI). Refer to "Executable and Linking Format", page 8, for additional information.

- Uses shared libraries, called dynamic shared objects (DSOs). DSOs are loaded at run time, instead of at link time, by the run-time linker, `rld`. The code for DSOs is not included in executable files; thus, executables built with DSOs are smaller than those built with non-shared libraries, and multiple programs can use the same DSO at the same time. For more information, see Chapter 3, "Using Dynamic Shared Objects", page 33.

- Creates position-independent code (PIC) by default to support dynamic linking. See "Position-Independent Code", page 9, for additional information.

Table 1-1, page 2, summarizes the compiler system components and the task each performs.

Table 1-1 Compiler System Functional Components

Tool	Task	Examples
Text editor	Write and edit programs	`vi`, `jot`, `emacs`
Compiler driver	Compile, link, and load programs	`cc`, `CC`, `f77`, `f90`, `as`
Object file analyzer	Analyze object files	`dis`, `dwarfdump`, `elfdump`, `file`, `nm`, `size`
SpeedShop/WorkShop	Analyze program performance	`cvperf`, `prof`, `sscompare`, `ssrun`
Archiver	Produce object-file libraries	`ar`
Linker	Link object files	`ld`
Runtime linker	Link Dynamic Shared Objects at runtime	`rld`
Debugger	Debug programs	`cvd`, `dbx`

A single program called a compiler driver (such as `cc`, `CC`, `f77`, or `f90`) invokes the following major components of the compiler system (refer to Figure 1-1, page 4)

- Compiler front end (C, C++, FORTRAN 77, Fortran 90, Assembler)

- Compiler back end (optimization and code generation)

- Inliner and Interprocedural Analysis(IPA)

- Linker (`ld`)

You can invoke a compiler driver with various options (described in the relevant man page, such as `cc`(1)) and with one or more source files as arguments. All specified source files are automatically sent to the macro preprocessor. To prevent running the preprocessor, use the `-nocpp` option on the driver command line.

Your program can take advantage of multiple CPUs (when present) to achieve higher computation rates. The optional parallel analyzers produce parallelized source code from standard source code. For more information about these packages and how to obtain them, contact your dealer or sales representative.

The compiler front end translates the source code into an intermediate tree representation. The compiler back end translates the intermediate code into object

code. The language compilers share the same back end, which combines optimization and code generation in one phase. (For more information about optimization, see Chapter 4, "Optimizing Program Performance", page 67).

The linker `ld` combines several object files into one, performs relocation, and resolves external symbols. The driver automatically runs `ld` unless you specify the `-c` option to skip the linking step.

When you compile or link programs, by default the compiler searches specific libraries depending on the compilation mode (shown in Table 1-2). Certain default libraries are automatically linked.

Table 1-2 Compiler Mode and Default Library Search Path

Mode	Path
o32	`/usr/lib`, `/lib`, and `/usr/local/lib`
n32	`/usr/lib32`, `/lib32`, and `/usr/local/lib`
64	`/usr/lib64`, `/lib64`, and `/usr/local/lib`

Compiler drivers and their respective libraries are listed in Table 1-3.

Table 1-3 Compilers and Default Libraries

Compiler	Default Libraries
`cc`	`libc.so`
`CC`	`libC.so`, `libc.so`, `libCsup.so`
`f77`, `f90`	`libftn.so`, `libftn90.so`, `libc.so`, `libm.so`

To see the various utilities a program passes through during compilation, invoke the appropriate driver with the `-show` option.

Figure 1-1 shows compilation flow from source file to executable file (`a.out`).
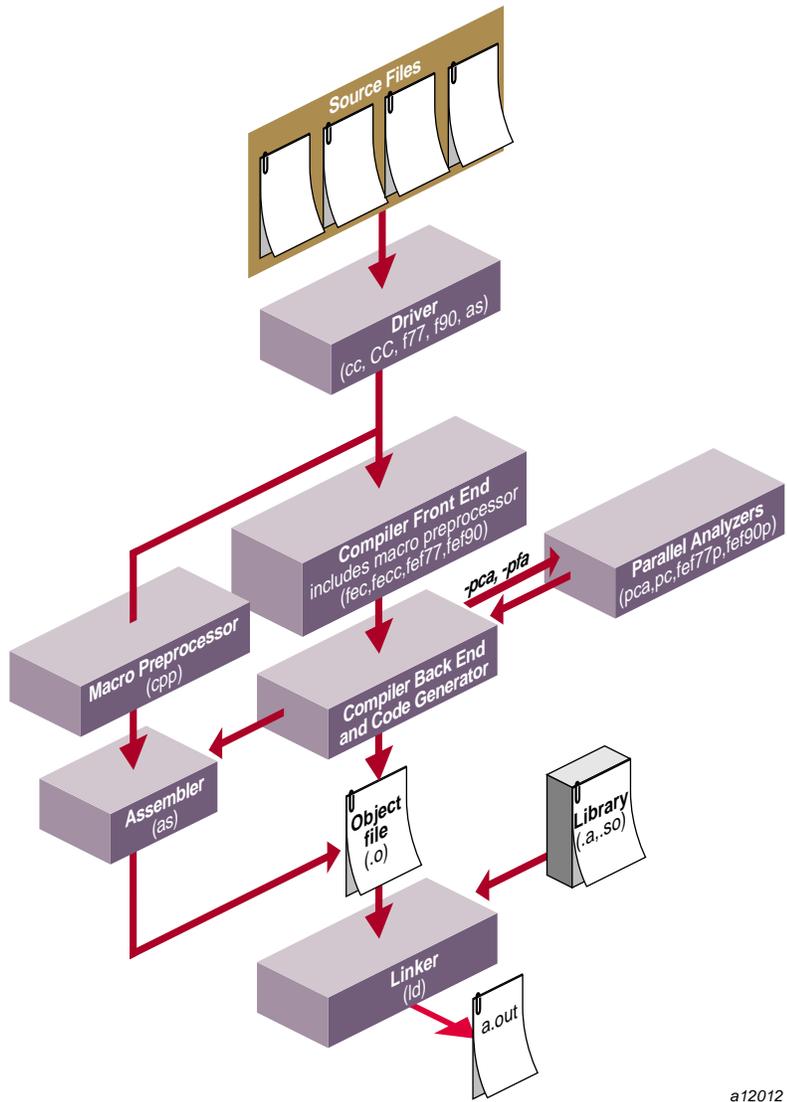
a12012

Figure 1-1 Compiler System Flowchart

Using the MIPSpro Compiler System

This chapter provides information about the MIPSpro compiler system and describes the object file format and dynamic linking. Specifically, this chapter covers the topics listed below:

- "Selecting Compilation Modes", page 5, explains how to specify n32, 64, or o32 compilation mode and how to set up a compiler.defaults file.

- "Object File Format and Dynamic Linking", page 8, discusses object files, including executable and linking format, dynamic shared objects, and position-independent code.

- "Source File Considerations", page 9, explains source file naming conventions and the procedure for including header files.

- "Compiler Drivers", page 19, lists and explains general compiler-driver options.

- "Linking", page 19, explains how to link programs manually (using ld or a compiler) and how to compile multilanguage programs. It also covers Dynamic Shared Objects (DSOs) and how to link them into a program.

- "Getting Information About Object Files", page 26, provides information on how to use the object file tools to analyze object files.

- "Using the Archiver to Create Libraries", page 30, explains how to use the archiver, ar.

- "Debugging", page 31, describes the compiler-driver options for debugging.

For further information on DSOs, see Chapter 3, "Using Dynamic Shared Objects", page 33. For information on optimizing your program, see Chapter 4, "Optimizing Program Performance", page 67.

Selecting Compilation Modes

You can select compilation modes by explicitly specifying them on a compiler command line, defining an environment variable, or specifying a file that defines some of the defaults. This section covers the following topics:

- Using a Defaults Specification File

- Setting an Environment Variable

- When to use -n32 or -64

Using a Defaults Specification File

You can set the following options without explicitly specifying them every time you invoke a compiler.

- The Application Binary Interface (ABI)

- The instruction set architecture (ISA)

- The processor type

- The optimization level

- The IEEE arithmetic level

Just set the environment variable COMPILER_DEFAULTS_PATH to a colon-separated list of paths designating where the compiler is to look for the compiler.defaults file. If no compiler.defaults file is found, or if the environment variable is not set, the compiler looks in /etc/compiler.defaults. If this file is not found, the compiler resorts to the built-in defaults.

The compiler.defaults file contains a -DEFAULT: *option* group specifier that specifies the default ABI, ISA, and processor. The compiler issues a warning if you specify anything other than -DEFAULT: *option* in the compiler.defaults file.

The format of the -DEFAULT: *option* group is specified in each of the language manuals.

Use the -show_defaults option to print the compiler.defaults being used (if any) and their values. This option is for diagnostic purposes and does not compile any code.

Explicit command-line options override all compiler default settings, and the SGI_ABI environment variable overrides the ABI setting in the compiler.defaults file. The following command overrides a compiler.defaults file that sets -DEFAULT:abi=n32:isa=mips4:proc=r10k and compiles -64 -mips4 -r10000:

```
% cc -64 foo.c
```

The following command overrides the `compiler.defaults` file and sets the ABI to `-o32` and the ISA to `-mips2`. The `-o32` ABI supports only `-mips2` (the default) and `-mips1` compilations.

```
% cc -o32 foo.c
```

The processor type is ignored by `-o32` compilations. Refer to the release notes and man pages for your compiler for information about default settings. Refer to the man pages for your command-line options.

Setting an Environment Variable

You can set an environment variable (shown in Table 2-1) to specify the compilation mode to use.

Table 2-1 Compilation Mode Environment Variable Specifications

Environment Variable	Description
setenv SGI_ABI -n32	Sets the environment for new 32-bit compilation.
setenv SGI_ABI -64	Sets the environment for 64-bit compilation.
setenv SGI_ABI -o32	Sets the environment for old 32-bit compilation.

When to Use **-n32** or **-64**

How do you know when to use `-n32` or `-64` to compile your code? Compile `-n32` when you want:

- To generate smaller executables than `-64`.

- Executables to have fewer data cache misses and less memory paging than `-64`.

- To access 64 bits: `long long` and `INTEGER*8` are 64-bits long.

Compile `-64` if your program:

- Requires more than 2 gigabytes of address space.

- Will overflow a 32-bit `long` integer.

Object File Format and Dynamic Linking

This section describes how the compiler system:

- Uses executable and linking format (ELF) for object files.

- Uses shared libraries called dynamic shared objects (DSOs).

- Creates position-independent code (PIC) by default to support dynamic linking.

Executable and Linking Format

The compiler system produces ELF object files. ELF is the format specified by the System V Release 4 Applications Binary Interface (the SVR4 ABI). ELF provides support for DSOs, described in the following section.

Types of ELF object files are as follows:

- Relocatable files, which contain code and data in a format suitable for linking with other object files to make a shared object or executable.

- DSOs, which contain code and data suitable for dynamic linking. Relocatable files may be linked with DSOs to create a dynamic executable. At run time, the run-time linker combines the executable and DSOs to produce a process image.

- Executable files ready for execution. They may or may not be dynamically linked.

You can use this version of the compiler system to construct ABI-compliant executables that run on any operating system supporting the MIPS ABI. Be careful to avoid referencing symbols that are not defined as part of the MIPS ABI specification. For more information, see the following publications:

- *System V Applications Binary Interface—Revised First Edition*. Prentice Hall, ISBN 0-13-880410-9

- *System V Application Binary Interface MIPS Processor Supplement*. Prentice Hall, ISBN 0-13-880170-3.

Dynamic Shared Objects

IRIX uses shared objects called Dynamic Shared Objects, or DSOs. The object code of a DSO is position-independent code (PIC), which can be mapped into the virtual address space of several different processes at once. DSOs are loaded at run time

instead of at linking time by the run-time loader, `rld`. As is true of static shared libraries, the code for DSOs is not included in executable files; thus, executables built with DSOs are smaller than those built with non-shared libraries, and multiple programs may use the same DSO at the same time. For more information on DSOs, see Chapter 3, "Using Dynamic Shared Objects", page 33.

Position-Independent Code

Dynamic linking requires that all object code used in the executable be position-independent code (PIC). For source files in high-level languages, you just need to recompile to produce PIC. Assembly language files must be modified to produce PIC; see the *MIPSpro Assembly Language Programmer's Guide* for details.

Position-independent code satisfies references indirectly by using a global offset table (GOT), which allows code to be relocated simply by updating the GOT. Each executable and each DSO has its own GOT. For more information on DSOs, see Chapter 3, "Using Dynamic Shared Objects", page 33.

The compiler system produces PIC by default when compiling higher-level language files. All of the standard libraries are provided as DSOs and therefore contain PIC code; if you compile a program into non-PIC, you will be unable to use those DSOs. One of the few reasons to compile non-PIC is to build a device driver, which does not rely on standard libraries.

Source File Considerations

This section describes conventions for naming source files and including header files.

Source File Naming Conventions

Each compiler driver recognizes the type of an input file by the suffix assigned to the file name. Table 2-2 describes the possible file name suffixes.

Table 2-2 Driver Input File Suffixes

Suffix	Description
.s	Assembly source
.i	Preprocessed source code in the language of the processing driver
.c	C source
.C, .c++, .CC, .cc, .CPP, .cpp, .CXX, .cxx	C++ source
.f.F.for.FOR .f.f90.F90	FORTRAN 77 source Fortran 90 source
.p	Pascal source
.o	Object file
.a	Object library archive
.so	DSO library

The following example compiles preprocessed source code:

```
f77 -c tickle.i
```

The f77 compiler also assumes the file has already been preprocessed (because the suffix is .i) and therefore does not invoke the preprocessor.

Header and Include Files

Header files, also called *include* files, contain code that is inserted into the program.

C header files contain information about the libraries with which they are associated. They define such things as data types, data structures, symbolic constants, and prototypes for functions exported by the library. To use those definitions without having to type them into each of your source files, you can use the #include directive to tell the macro preprocessor to include the complete text of the given header file in the current source file. When you include header files in your source files, you can specify definitions conveniently and consistently in each source file that uses any of the library routines.

Fortran include files are specified by the INCLUDE line, which names a file containing source text. That source text is substituted for the INCLUDE line during compilation. The source text can be any Fortran code that is valid in the context of its location in the program.

By convention, C header file names have a .h suffix. Each programming language handles these files the same way, via the macro preprocessor. For example, the stdio.h header file describes, among other things, the data types of the parameters required by the C language printf() function.

For detailed information about standard header files and libraries, see the International Standard ISO/IEC, *Programming languages—C, 9899*, 1990. Also see "Using Typedefs ", page 140, for information about the inttypes.h header file.

Specifying a Header File

The #include directive in C and C++ or the INCLUDE line in Fortran tells the preprocessor to replace the directive or line with the text of the indicated header file. The usual way to specify a header file in C is with the following line:

#include <*filename*>

The *filename* is the name of the header file to be included. The angle brackets (< >) surrounding the filename tell the macro preprocessor to search for the specified file only in directories specified by command-line options and in the default header file directory (/usr/include and /usr/include/CC for C++).

In another specification format, *filename* is given between double quotation marks (' ' ' '). In this case, the macro preprocessor searches for the specified header file in the current directory first (that is, the directory containing the main program file). If the preprocessor does not find the requested file, it searches the other directories as in the angle-bracket specification.

In Fortran, included text is specified as follows:

INCLUDE '*filename*'

In an f90(1) program, the directory containing *filename* can be specified on the command line with the -I option. In an f77(1) program, *filename* must be in the current directory.

Creating a Header File for Multiple Languages

A single header file can contain definitions for multiple languages; this setup allows you to use the same header file for all programs that use a given library, no matter what language those programs are in.

To set up a shareable header file, create a .h file and enter the definitions for the various languages as follows:

```
#ifdef _LANGUAGE_C
    C Definitions
#endif

#ifdef _LANGUAGE_C_PLUS_PLUS
    C++ definitions
#endif

#ifdef _LANGUAGE_FORTRAN
    Fortran definitions
#endif
```

Note: You must specify _LANGUAGE_ before the language name. To indicate C++ definitions, you must use _LANGUAGE_C_PLUS_PLUS, not _LANGUAGE_C++.

You can specify language definitions in any order.

Using Precompiled Headers in C and C++

This section describes the precompiled header mechanism that is available with the n32 and 64-bit C and C++ compilers. This mechanism is also available for C++ (but not C) in o32-bit mode.

This section contains the following topics:

- About precompiled headers, see the following section.

- Automatic precompiled header processing, see "Automatic Precompiled Header Processing", page 13.

- Other ways to control precompiled headers, see "Other Ways to Control Precompiled Headers", page 17.

- PCH performance issues, see "PCH Performance Issues", page 17.

About Precompiled Headers

The precompiled header (PCH) file mechanism is available through the C and C++ compilers front ends: `fec` and `fecc`. Use PCH to avoid recompiling a set of header files. This is particularly useful when your header files introduce many lines of code, and the primary source files that included them are relatively small.

In effect, `fec` and `fecc` take a snapshot of the state of the compilation at a particular point and write it to a file before completing the compilation. When you recompile the same source file or another file with the same set of header files, the PCH mechanism recognizes the snapshot point, verifies that the corresponding PCH file is usable, and reads it back in.

The PCH mechanism can give you a dramatic improvement in compile-time performance. The trade-off is that PCH files may take a lot of disk space.

Automatic Precompiled Header Processing

This section covers the following topics:

- PCH file requirements

- Reusing PCH files

- Obsolete file deletion mechanism

You can enable precompiled header processing by using the `-pch` option (`-Wf`, `-pch` in 32-bit mode) on the command line. With the PCH mechanism enabled, `fec` or `fecc` searches for a qualifying PCH file to read in or creates one for use on a subsequent compilation.

The PCH file contains a snapshot of all the code preceding the header stop point. The header stop point is typically the first token in the primary source file that does not belong to a preprocessing directive. The header stop point can also be specified directly by inserting a `#pragma hdrstop`. For example, consider the following C++ code:

```
#include ``xxx.h''
#include ``yyy.h''
int i;
```

In this case, the header stop point is `int i` (the first non-preprocessor token), and the PCH file will contain a snapshot reflecting the inclusion of `xxx.h` and `yyy.h`. If the first non-preprocessor token or the `#pragma hdrstop` appears within a `#if` block, the header stop point is the outermost enclosing `#if`. For example, consider the following C++ code:

```
#include ''xxx.h''
#ifndef YYY_H
#define YYY_H 1
#include ''yyy.h''
#endif
#if TEST
int i;
#endif
```

In this case, the first token that does not belong to a preprocessing directive is again `int i`, but the header stop point is the start of the `#if` block containing the `int`. The PCH file reflects the inclusion of `xxx.h` and conditionally the definition of `YYY_H` and inclusion of `yyy.h`. The file does not contain the state produced by `#if TEST`.

PCH File Requirements

A PCH file is produced only if the header stop point and the code preceding it (generally the header files themselves) meet the following requirements:

- The header stop point must appear at file scope; it may not be within an unclosed scope established by a header file. For example, a PCH file is not created in the following case:

  ```
  // xxx.h
  class A {

  // xxx.C
  #include "xxx.h"
  int i; };
  ```

- The header stop point cannot be inside a declaration started within a header file, and it cannot be part of a declaration list of a linkage specification. For example, a PCH file is not created in the following case:

  ```
  // yyy.h
  static

  // yyy.C
  ```

```
#include "yyy.h"
int i;
```

In this case, the header stop point is `int i`, but since it is not the start of a new declaration, a PCH file is not created

- The header stop point cannot be inside a `#if` block or a `#define` started within a header file.

- The processing preceding the header stop must not have produced any errors. (Note that warnings and other diagnostics are not reproduced when the PCH file is reused.)

- References to predefined macros `__DATE__` or `__TIME__` cannot be included.

- Use of the `#line` preprocessing directive cannot be included.

- `#pragma no_pch` cannot be included.

Reusing PCH Files

When a precompiled header file is produced, in addition to the snapshot of the compiler state, it contains some information that can be checked to determine under what circumstances it can be reused. This information includes the following:

- The compiler version, including the date and time the compiler was built.

- The current directory (in other words, the directory in which the compilation is occurring).

- The command–line options.

- The initial sequence of preprocessing directives from the primary source file, including `#include` directives.

- The date and time of the header files specified in `#include` directives.

This information comprises the PCH prefix. The prefix information of a given source file can be compared to the prefix information of a PCH file to determine whether or not the latter is applicable to the current compilation.

For example, consider the following C++ code:

```
// a.C
#include "xxx.h"
```

```
...                                 // Start of code

// b.C
#include "xxx.h"

...                                 // Start of code
```

When you compiled a.C with the -pch option, the PCH file a.pch is created. When you compile b.C (or recompile a.C), the prefix section of a.pch is read in for comparison with the current source file. If the command line options are identical and xxx.h has not been modified, fec or fecc reads in the rest of a.pch rather than opening xxx.h and processing it line by line. This establishes the state for the rest of the compilation.

It may be that more than one PCH file is applicable to a given compilation. If so, the largest (in other words, the one representing the most preprocessing directives from the primary source file) is used. For instance, consider a primary source file that begins with the following code:

```
#include "xxx.h"
#include "yyy.h"
#include "zzz.h"
```

If one PCH file exists for xxx.h and a second for xxx.h and yyy.h, the latter will be selected (assuming both are applicable to the current compilation). After the PCH file for the first two headers is read in and the third is compiled, a new PCH file for all three headers may be created.

When a precompiled header file is created, it takes the name of the primary source file, with the suffix replaced by pch. Unless -pch_dir is specified, the PCH file is created in the directory of the primary source file.

When a precompiled header file is created or used, a message similar to the following is issued:

```
"test.C": creating precompiled header file "test.pch"
```

Obsolete File Deletion Mechanism

In automatic mode (when -pch is used), the front end considers a PCH file obsolete and deletes it under the following circumstances:

- The file is based on at least one out-of-date header file but is otherwise applicable for the current compilation.

• The file has the same base name as the source file being compiled (for example, `xxx.pch` and `xxx.C`) but is not applicable for the current compilation (for example, because of different command-line options).

You must manually clean up any other PCH file.

Support for PCH processing is not available when multiple source files are specified in a single compilation. If the command line includes a request for precompiled header processing and specifies more than one primary source file, an error is issued and the compilation is aborted.

Other Ways to Control Precompiled Headers

You can use the following ways to control and tune how precompiled headers are created and used:

• You can insert a `#pragma hdrstop` in the primary source file at a point prior to the first token that does not belong to a preprocessing directive. Thus you can specify where the set of header files subject to precompilation ends, as in the following:

```
#include "xxx.h"
#include "yyy.h"
#pragma hdrstop
#include "zzz.h"
```

In this case, the precompiled header file includes the processing state for `xxx.h` and `yyy.h` but not `zzz.h`. This is useful if you decide that the information added by what follows the `#pragma hdrstop` does not justify the creation of another PCH file.

• You can use a `#pragma no_pch` to suppress the precompiled header processing for a given source file.

• You can use the command-line option `-pch_dir` *directoryname* to specify the directory in which to search for and create a PCH file.

PCH Performance Issues

The relative overhead incurred in writing out and reading in a precompiled header file is quite small for reasonably large header files.

In general, writing out a precompiled header file does not cost much, even if it does not end up being used, and, if it is used, it almost always produces a significant speedup in compilation. The problem is that the precompiled header files can be quite large (from a minimum of about 250 Kbytes to several Mbytes or more), and so you probably do not want many of them sitting around.

You can see that, despite the faster recompilations, precompiled header processing is not likely to be justified for an arbitrary set of files with nonuniform initial sequences of preprocessing directives. The greatest benefit occurs when a number of source files can share the same PCH file. The more sharing, the less disk space is consumed. With sharing, the disadvantage of large precompiled header files can be minimized without giving up the advantage of a significant speedup in compilation times.

To take full advantage of header file precompilation, you should reorder the #include sections of your source files and group the #include directives within a commonly used header file.

The fecc source provides an example of how this can be done. A common idiom is the following:

```
#include "fe_common.h"
#pragma hdrstop
#include ...
```

In this example, fe_common.h pulls in, directly and indirectly, a few dozen header files. The #pragma hdrstop is inserted to get better sharing with fewer PCH files. The PCH file produced for fe_common.h is slightly over a megabyte in size. Another example, used by the source files involved in declaration processing, is the following:

```
#include "fe_common.h"
#include "decl_hdrs.h"
#pragma hdrstop
#include ...
```

decl_hdrs.h pulls in another dozen header files, and a second, somewhat larger, PCH file is created. In all, the fifty-odd source files of fecc share just six precompiled header files. If disk space is at a premium, you can decide to make fe_common.h pull in all the header files used. In that case, a single PCH file can be used in building fecc.

Different environments and different projects have different needs. You should, however, be aware that making the best use of the precompiled header support will require some experimentation and probably some minor changes to your source code.

Compiler Drivers

The driver commands cc(1), CC(1), f90(1), and f77(1) call subsystems that compile, optimize, assemble, and link your programs. This section describes the default behavior for compiler drivers.

At compilation time, you can select one or more options that affect a variety of program development functions, including debugging, profiling, and optimizing. You can also specify the names assigned to output files. Note that some options have default values that apply if you do not specify them.

When you invoke a compiler driver with source files as arguments, the driver calls other commands that compile your source code into object code. It then optimizes the object code (if requested to do so) and links together the object files, the default libraries, and any other libraries you specify.

Given a source file foo.c, the default name for the object file is foo.o. The default name for an executable file is a.out. The following example compiles source files foo.c and bar.c with the default options:

% **cc foo.c bar.c**

This example produces two object files, foo.o and bar.o, and links them with the default C library, libc, to produce an executable called a.out.

Note: If you compile a single source directly to an executable, the compiler does not create an object file.

The command-line options for MIPSpro compiler drivers are listed and explained in the man page for your compiler.

Linking

The linker, ld, combines one or more object files and libraries (in the order specified) into one executable file, performing relocation, external symbol resolutions, and all other required processing. Unless directed otherwise, the linker names the executable file a.out. See the ld(1) man page for complete information on the linker.

This section summarizes the functions of the linker. It also covers how to link a program manually (without using a compiler driver) and how to compile multilanguage programs. Specifically, this section describes:

- Invoking the linker, see the following section.

- "Linking Assembly Language Programs", page 20.

- "Linking Libraries", page 21.

- "Linking to Previously Built Dynamic Shared Objects", page 23.

- "Linking Multilanguage Programs", page 23.

- "Finding an Unresolved Symbol with ld", page 26.

Invoking the Linker

Usually the compiler invokes the linker as the final step in compilation. If object files produced by previous compilations exist and you want to link them, invoke the linker by using a compiler driver instead of calling ld directly. Just pass the object file names to the compiler driver in place of source file names. If the original source files are in one language, invoke the associated driver and specify the list of object files.

In some cases you may need to invoke ld directly, such as when you are building a shared object or doing special linking not supported by compiler drivers (such as building an embedded system).

For information on the options available to the linker, see the ld man page.

Linker Example

The following command tells the linker to search for the DSO libcurses.so in the directory /usr/lib. If it does not find that DSO, the linker then looks for libcurses.a in /lib. The linker does not look for DSOs in /usr/local/lib, so do not put shared objects there.

```
% ld foiled.o again.o -lcurses
```

If found in any of these places, the DSO or library is linked with the objects foiled.o and again.o; if they are not found, an error is generated.

Linking Assembly Language Programs

The assembler driver, as, does not run the linker. To link a program written in assembly language, use one of these procedures:

- Assemble and link using one of the other driver commands (cc, for example). The .s suffix of the assembly language source file causes the driver to invoke the assembler.

- Assemble the file using as. Then link the resulting object file with the ld command.

Linking Libraries

The linker processes its arguments from left to right as they appear on the command line. Arguments to ld can be object files, DSOs, or libraries. Be sure to list object files before DSOs.

When ld reads a DSO, it adds all the symbols from that DSO to a cumulative symbol table. If it encounters a symbol that is already in the symbol table, it does not change the symbol table entry. If you define the same symbol in more than one DSO, only the first definition is used.

When ld reads an archive, usually denoted by a file name ending in .a, it uses only the object files from that archive that can resolve currently unresolved symbol references. When a symbol is referred to but not defined in any of the object files that have been loaded so far, it is called unresolved.

Once a library has been searched in this way, it is never searched again. Therefore, place libraries after object files on the command line in order to resolve as many references as possible. If a symbol is already in the cumulative symbol table from having been encountered in a DSO, its definition in any subsequent archive or DSO is ignored.

Specifying Libraries and DSOs

You can specify libraries and DSOs either by explicitly stating a path name or by using the library search rules. To specify a library or DSO by path (either relative to the current directory or absolute), simply include that path on the command line:

```
% ld myprog.o /usr/lib/libc.so.1 mylib.so
```

Note: libc.so.1 is the name of the standard C DSO, replacing the older libc.a. Similarly, libX11.so.1 is the X11 DSO. Most other DSOs are simply named *name*.so, without a .1 extension.

To use the linker's library search rules, specify the library with the -l*name* option:

```
% ld myprog.o -lmylib
```

When the -lmylib argument is processed, ld searches for a file called libmylib.so. If it cannot find libmylib.so in a given directory, it tries to find libmylib.a there; if it cannot find that either, it moves on to the next directory in its search order.

The default search order uses the path appropriate to the compilation mode:

- For -n32, the default search order is /usr/lib32:/lib32.

- For -64, the default search order is /usr/lib64:/lib64.

- For -o32, the default search order is /usr/lib:/lib.

If ld is invoked from one of the compiler drivers, all -L and -nostdlib options are moved up on the command line so that they appear before any -l*name* options. For example, consider the command:

```
% cc file1.o -lm -L mydir
```

This command invokes, at the linking stage of compilation, the following:

```
% ld -L mydir file1.o -lm
```

Note: There are three different kinds of files that contain object code files: non-shared libraries, PIC archives, and DSOs. Non-shared libraries are the old-style library. PIC archives are the default, built using ar from .o files compiled with -KPIC (the default option). They can be linked with other PIC files. DSOs are built from PIC .o files by using ld -shared; see Chapter 3, "Using Dynamic Shared Objects", page 33, for details.

If the linker tells you that a reference to a certain function is unresolved, check that function's man page to find out which library the function is in. If it is not in one of the standard libraries that ld links in by default, you may need to specify the appropriate library on the command line. For an alternative method of finding out where a function is defined, see "Finding an Unresolved Symbol with ld", page 26.

Note: Simply including the header file associated with a library routine is not enough; you must also specify the library when linking (unless it is a standard library). No automatic connection exists between header files and libraries. Header files only give prototypes for library routines, not the library code itself.

Examples of Linking DSOs

To link a sample program `foo.c` with the math DSO, `libm.so`, enter:

```
% cc foo.c -lm
```

To specify the appropriate DSOs for a graphics program `foogl.c`, enter:

```
% cc foogl.c -lgl -lX11
```

Note: When linking, you must specify the source file name before the linker options.

Linking to Previously Built Dynamic Shared Objects

This section describes how to link your source files with previously built DSOs; for more information about how to build your own DSOs, see Chapter 3, "Using Dynamic Shared Objects", page 33.

To build an executable that uses a DSO, simply call a compiler driver. For instance, the following command links the resulting object file, `needle.o`, with the previously built DSO, `libthread.so`, and the standard C DSO, `libc.so.1`, if available.

```
% cc needle.c -lthread
```

If no `libthread.so` exists, but a PIC archive named `libthread.a` exists, that archive is used with `libc.so.1`. So you still get dynamic (run-time) linking. Note that even `.a` libraries contain position-independent code by default, though it is also possible to build non-shared `.a` libraries that do not contain PIC.

Linking Multilanguage Programs

The source language of the main program may differ from that of a subprogram. Follow the steps below to link multilanguage programs. For an illustration of the process, see Figure 2-1, page 25.

1. Compile object files from the source files of each language separately by using the `-c` option.

   For example, if the source consists of a Fortran main program, `main.f`, and two files of C functions, `more.c` and `rest.c`, use the commands:

```
% cc -c more.c rest.c
% f77 -c main.f
```

These commands produce the object files `main.o`, `more.o`, and `rest.o`.

2. Use the compiler associated with the language of the main program to link the objects:

```
% f77 main.o more.o rest.o
```

The compiler drivers supply the default set of libraries necessary to produce an executable from the source of the associated language. However, when producing executables from source code in several languages, you may need to specify the default libraries explicitly for one or more of the languages. For instructions on specifying libraries, see "Linking Libraries", page 21.
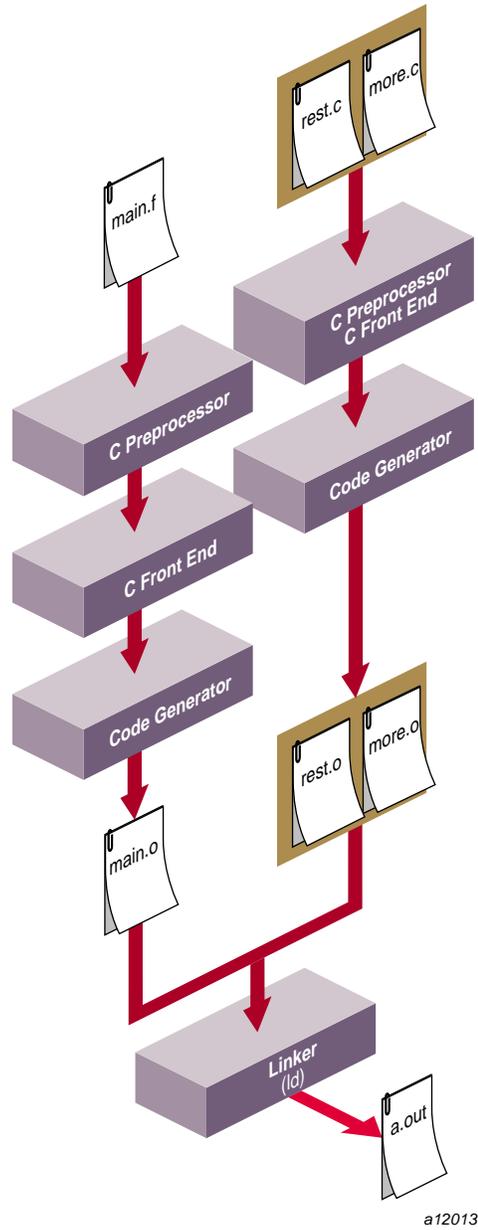
Figure 2-1 Compilation Control Flow for Multilanguage Programs

> **Note:** Use caution when passing pointers and longs between languages, since some languages use different type sizes and structures for data types.

For specific details about compiling multilanguage programs, see the programming guides for the appropriate languages.

Finding an Unresolved Symbol with `ld`

You can use `ld` to locate unresolved symbols. For example, suppose you are compiling a program, and `ld` tells you that you are using an unresolved symbol. You may not know where the unresolved symbol is referenced.

To find the unresolved symbol, enter:

% **ld -y***symbol file1 ... filen*

You can also enter:

% **cc prog.o -Wl,-y***symbol*

The output lists the source file that references *symbol*.

Getting Information About Object Files

Several tools provide information on object files and are described in the following sections:

- `dis` disassembles an object file into machine instructions.

- `dwarfdump` lists headers, tables, and other selected parts of a DWARF-format object file or archive file.

- `elfdump` lists the contents, including the symbol table and header information, of an ELF-format object file.

- `file` provides descriptive information on the properties of a file.

- `nm` lists symbol table information.

- `size` prints the size of each section of an object file.

- `strip` removes symbol table and relocation bits.

You can trace system call and scheduling activity by using the `par` command. For more information, see the `par`(1) man page.

Disassembling Object Files with `dis`

The `dis` tool disassembles object files into machine instructions. You can disassemble an object, archive library, or executable file.

See the `dis`(1) man page for descriptions of its options.

Listing Parts of DWARF Object Files with `dwarfdump`

The `dwarfdump` tool provides debugging information from selected parts of DWARF symbolic information in an ELF object file. For more information on DWARF, including option descriptions, see the `dwarfdump`(1) and `dwarf`(4) man pages.

Listing Parts of ELF Object Files and Libraries with `elfdump`

The `elfdump` tool lists headers, tables, and other selected parts of an ELF-format object file or archive file. See the `elfdump`(1) man page for option descriptions and other information.

Determining File Type with `file`

The `file` tool lists the properties of program source, text, object, and other files. This tool attempts to identify the contents of files using various heuristics. It is not exact and often erroneously recognizes command files as C programs. For more information, including option descriptions, see the `file`(1) man page.

Listing Symbol Table Information: `nm`

The `nm` tool lists symbol table information for object files and archive files. To get XPG4 (X/Open Portability Group) format, set the environment variable, `_XPG` in your environment.

For more information and option descriptions, see the `nm`(1) man page.

This example demonstrates how to obtain a symbol table listing. Consider the following program, tnm.c:

```
#include <stdio.h>
#include <math.h>
#define LIMIT 12
int unused_item = 14;
double mydata[LIMIT];

main()
{
     int i;
     for(i = 0; i < LIMIT; i++) {
         mydata[i] = sqrt((double)i);
     }
     return 0;
}
```

Compile the program into an object file by entering:

```
% cc -c tnm.c
```

To obtain symbol table information for the object file tnm.o in BSD format, use the nm -B command:

```
0000000000 T main
0000000000 B mydata
0000000000 U sqrt
0000000000 D unused_item
0000000000 N _bufendtab
```

To obtain symbol table information for the object file tnm.o in SVR4 format, use the nm command without any options:

```
Symbols from tnm.o:
```

[Index]	Value	Size	Class	Type	Section	Name
[0]	| 0|		|File	|ref=4	|Text	| tnm.c
[1]	| 0|		|Proc	|end=3 int	|Text	| main
[2]	| 116|		|End	|ref=1	|Text	| main
[3]	| 0|		|End	|ref=0	|Text	| tnm.c
[4]	| 0|		|File	|ref=6	|Text	| /usr/include/math.h

```
[5]    |         0|          |End     |ref=4       |Text     | /usr/include/math.h
[6]    |         0|          |Global  |            |Data     | unused_item
[7]    |         0|          |Global  |            |Bss      | mydata
[8]    |         0|          |Proc    |ref=1       |Text     | main
[9]    |         0|          |Proc    |            |Undefined| sqrt
[10]   |         0|          |Global  |            |Undefined| _gp_disp
```

Determining Section Sizes with `size`

The `size` tool prints information about the sections (such as text, rdata, and sbss) of the specified object or archive files. The `elf`(4) man page describes the format of these sections, and the `size`(1) man page describes the options accepted by the `size` command.

An example of the `size` command and the listings produced follows:

% **size a.out**

Section	Size	Physical Address	Virtual Address
.interp	21	268435856	268435856
.MIPS.options	104	268435880	268435880
.dynamic	464	268435984	268435984
.liblist	20	268436448	268436448
.MIPS.symlib	30	268436468	268436468
.msym	240	268436500	268436500
.dynstr	312	268436744	268436744
.dynsym	720	268437056	268437056
.hash	256	268437776	268437776
.MIPS.stubs	56	268438032	268438032
.text	460	268438088	268438088
.init	24	268438548	268438548
.data	17	268505088	268505088
.sdata	8	268505108	268505108
.got	112	268505120	268505120
.bss	36	268505232	268505232

Removing Symbol Table and Relocation Bits with `strip`

The `strip` tool removes symbol table and relocation bits that are attached to the assembler and loader. Use `strip` to save space after you debug a program. The effect of `strip` is the same as that of using the -s option to `ld`.

See the `strip`(1) man page for descriptions of the options.

Using the `.s` Assembly Language File

The MIPSpro compilers can produce a .s file rather than a .o file. The file is produced by specifying the -S option on the command line instead of the -c option.

The assembly language file that is produced contains exactly the same set of instructions that would have been produced in the .o object file, and inputting the .s file to the assembler produces an object file with the same instructions that the compiler would have produced. The .s file is a listing of the instructions, but does not contain all the object information that a .o file contains. Therefore, a .o file generated by a .s file will not be exactly the same as one generated directly by the compiler and they are not guaranteed to work identically (for example, `reorg_common` information is lost).

In addition to the program's instructions, the .s file contains comments indicating the effects of various optimization transformations that were made by the compiler.

For details about this file, see the *MIPSpro Assembly Language Programmer's Guide*.

Using the Archiver to Create Libraries

An archive library is a file that includes the contents of one or more object (.o) files. When the linker (`ld`) searches for a symbol in an archive library, it loads only that object file where that symbol was defined (not the entire library) and links it with the calling program.

The archiver (`ar`) creates and maintains archive libraries and has these main functions:

• Copying new objects into the library

• Replacing existing objects in the library

• Moving objects around within the library

- Extracting individual objects from the library

- Creating a symbol table for the linker to search symbols

The following section explains the syntax of the `ar` command and lists some options and examples of how to use it. See the `ar`(1) man page for details.

Note: `ar` simply strings together whatever object files you tell it to archive. For information about building DSOs and converting libraries to DSOs, see Chapter 3, "Using Dynamic Shared Objects", page 33.

`ar` Examples

To create a new library, `libtest.a`, and add object files to it, enter:

% **ar -cq libtest.a mcount.o mon1.o string.o**

The `-c` option suppresses an archiver message during the creation process. The `-q` option creates the library and puts `mcount.o`, `mon1.o`, and `string.o` into it.

To replace an object file in an existing library, enter:

% **ar -r libtest.a mon1.o**

The `-r` option replaces `mon1.o` in the library `libtest.a`. If `mon1.o` does not already exist in the library `libtest.a`, it is added.

Note: If you specify the same file twice in an argument list of files to be added to an archive, that file appears twice in the archive.

Debugging

The compiler system provides the `dbx`(1) debugging tool, which is described in detail in the *dbx User's Guide*, and `cvd`(1), which is part of the ProDev WorkShop suite of performance tools. For information about the WorkShop tools, see the *ProDev WorkShop: Overview*.

Before using one of the debuggers, specify the `-g` driver option to produce executables containing information that the debugger can use (see the `dbx`(1) or `cvd` man page).

Using Dynamic Shared Objects

A dynamic shared object (DSO) is an object file that is meant to be used simultaneously (or shared) by multiple applications (a.out files) while they are executing.

As you read this chapter, you will learn how to build and use DSOs. This chapter covers the following topics:

- "Benefits of Using DSOs", page 33, explains the benefits of DSOs.

- "Using DSOs", page 35, tells you how to obtain the most benefit from using DSOs when creating your executable.

- "Taking Advantage of QuickStart", page 40, discusses an optimization you can use to make sure that the DSOs you build load as quickly as possible.

- "Building DSOs", page 43, describes how to build a DSO.

- "Run-Time Linking", page 47, discusses the run-time linker, and how it locates DSOs at run time.

- "Dynamic Loading Under Program Control", page 53, explains the use of dlopen() and dlsym() to control run-time linking.

- "Versioning of DSOs", page 55, discusses a versioning mechanism for DSOs that allows binaries linked against different, incompatible versions of the same DSO to run correctly.

- "Loading Multiple DSOs", page 58, explains how to use the 'big page' options to load multiple DSOs on a single page.

You can use DSOs in place of archive libraries (they replace static shared libraries provided with earlier releases of IRIX).

Benefits of Using DSOs

Because DSOs contain shared components, using them provides several substantial benefits:

- **DSOs minimize overall memory use**: DSOs minimize overall memory usage because code is shared. Two executables that use the same DSO and that run

simultaneously have only one copy of the instruction from the shared component loaded into memory. For example, if executable A and executable B both link with the same DSO C, and if A and B are both running at the same time, the total memory used is what is required for A, B, and C, plus some small overhead. If C is an unshared library, the memory used is what is required for A, B, and two copies of C.

- **Executables linked with DSOs are smaller**: Executables linked with DSOs are smaller than those linked with unshared libraries because the shared objects are not part of the executable file image, so disk usage is minimized.

- **DSOs are easier to use, build, and debug**: DSOs are much easier to use, build, and debug than the static shared libraries (supplied in IRIX 4 and earlier). Most of the libraries supplied by SGI today are available as DSOs. In IRIX 4 and earlier, only a few static shared libraries were available; most libraries were unshared.

- **Executables using DSOs do not have to be relinked**: Executables that use a DSO do not have to be relinked if the DSO changes; when the new DSO is installed, the executable automatically starts using it. This feature makes it easier to update end users with new software versions. It also allows you to create hardware-independent software packages more easily.

  Suppose, for example, you want to build both MIPS IV and a MIPS III version of a shared object. You want your program to use the MIPS IV version when it is running on a Power Challenge (R8000) system, and also run correctly on another 64-bit platform. Suppose you want to do the above with the routines in a library named `libchange.so`. To do this, build one version of the routines in `libchange` using the `-mips4` option, and place it in `/usr/lib64/mips4` on a Power Challenge system. Next, build another version using the `-mips3` option, and place it in `/usr/lib64`. Then, when you build an executable that uses `libchange`, use the `-rpath` option to tell the run-time linker to look first for MIPS IV versions of the libraries. For example:

  ```
  % cc -mips3 -o prog prog.o -rpath /usr/lib64/mips4 -lchange
  ```

  As a result, `prog` runs on any IRIX 6 (and later) system, and it automatically takes advantage of any MIPS IV libraries whenever it runs on a Power Challenge system.

- **DSOs and executables are mapped into memory**: DSOs and the executables that use them are mapped into memory by a run-time loader, `rld`, which resolves external references between objects and relocates objects at run time. (DSOs contain only position-independent code (PIC), so they can be loaded at any virtual address at run time.) With `rld`, the binding of symbols can be changed at run

time at the request of the executing program. You could use this feature to dynamically change the feature set presented to a user of your application, for example, while minimizing start-up time. The application could be started quickly, with a subset of the features available and then, if the user needs other features, those can be loaded in under programmatic control.

Costs that are involved with using DSOs are explained in "Using DSOs", page 35. The sections after that explain how to build and optimize DSOs and how `rld` works. See the `rld`(1) man page for more information. The `dso`(5) man page also contains more information about DSOs.

Using DSOs

The linker command-line syntax is the same as for an archive (`.a`) library. This section explains how to use DSOs. Specific topics include:

- "DSOs vs. Archive Libraries" which describes differences between DSOs and archive libraries.

- "Using QuickStart", page 36, which briefly explains how QuickStart minimizes start-up times for executables.

- "Guidelines for Using Shared Libraries", page 36, which lists points to consider when you choose library members and tune shared library code.

DSOs vs. Archive Libraries

The following compile line creates the executable `yourApp` by linking with the DSOs `libyours.so` and with `libc.so.1`:

% **cc yourApp.c -o yourApp -lyours**

If `libyours.so` is not available, but the archive version `libyours.a` is available, that archive version is used along with `libc.so.1`.

A significant difference exists between DSOs and archive libraries in terms of what is mapped into the address space when an application is executing. With an archive library, only the text portion of the library that the application actually requires (and the data associated with that text) is mapped, not the entire library. In contrast, the entire DSO that is linked is mapped; in many cases, however, the DSO is shared and

already mapped into the address space. Thus, to conserve address space and save time at startup, do not link with DSOs unless your application actually needs them.

Avoid listing any archive libraries on the compile line after you list shared libraries; instead, list the archive libraries first and then the DSOs.

Using QuickStart

You may want to take advantage of the QuickStart optimization that minimizes start-up times for executables. You can use QuickStart when using or building DSOs. At link time, when an executable or a DSO is being created, the linker `ld` assigns initial addresses to the object and attempts to resolve all references. Since DSOs are relocatable, these initial address assignments are really only guesses about where the object will be really loaded. At run time, `rld` verifies that the DSO being used is the same one that was linked with and what the real addresses are. If the DSOs are the same and if the addresses match the initial assignments, `rld` does not have to perform any relocation work, and the application starts up very quickly (or QuickStarts). When an application QuickStarts, memory use is less since `rld` does not have to read in the information necessary to perform relocations.

To determine whether your application (or DSO) is able to do a QuickStart, use the `-quickstart_info` flag when building the executable (or DSO). If the application or DSO cannot do a QuickStart, you will be given information about what to do. The next section goes into more detail about why an executable may not be able to use QuickStart.

In summary, when you use DSOs to build an executable, remember the following:

* Link with only the DSOs that you need.

* Make sure that archive libraries precede DSOs on the compile line.

* Use the `-quickstart_info` flag.

Guidelines for Using Shared Libraries

When you are working with DSOs, you can avoid some common pitfalls if you adhere to the guidelines described in this section:

* "Choosing DSO Library Members", page 37, explains what routines to include and exclude when you choose library members.

- "Tuning Shared Library Code", page 38, covers how to tune shared library code by minimizing global data, improving locality, and aligning for paging.

Choosing DSO Library Members

This section covers some important considerations for choosing library members.

- **Include large, frequently used routines**. These routines are prime candidates for sharing. Placing them in a shared library saves code space for individual `a.out` files and saves memory, too, when several concurrent processes need the same code. `printf`(3S) and related C library routines are good examples of large, frequently used routines.

- **Exclude infrequently used routines**. Putting these routines in a shared library can degrade performance, particularly on paging systems. Traditional `a.out` files contain all code they need at run time. By definition, the code in an `a.out` file is (at least distantly) related to the process. Therefore, if a process calls a function, it may already be in memory because of its proximity to other text in the process.

  If the function is in the shared library, a page fault may be more likely to occur, because the surrounding library code may be unrelated to the calling process. Only rarely will any single `a.out` file use everything in the shared C library. If a shared library has unrelated functions, and unrelated processes make random calls to those functions, the locality of reference may be decreased. The decreased locality may cause more paging activity and, thereby, decrease performance.

- **Exclude routines that use much static data**. These modules increase the size of processes. Every process that uses a shared library gets its own private copy of the library's data, regardless of how much of the data is needed.

  Library data is static: it is not shared and cannot be loaded selectively with the provision that unreferenced pages may be removed from the working set.

  For example, `getgrent`(3C) is not used by many standard UNIX commands. Some versions of the module define over 1400 bytes of unshared, static data. So, do not include it in a shared library. You can import global data, if necessary, but not local, static data.

- **Make libraries self-contained**. It is best to make the library self-contained. You can do this by including routines in the shared object. For example, `printf`(3S) requires much of the standard I/O library. A shared library containing `printf`(3S), should also contain the rest of the standard I/O routines. This is done with `libc.so.1`.

If your shared object calls routines from a different shared object, it is best to build in this dependency by naming the needed shared objects on the link line in the usual way. For example:

```
% ld -shared -all mylib.a -o mylib.so -soname mylib.so -lfoo
```

This command line specifies that libfoo.so is needed by mylib.so. Thus, when an application is linked against mylib.so, it is not necessary to specify -lfoo.

This guideline should not take priority over the others in this section. If you exclude some routine that the library itself needs based on a previous guideline, consider leaving the symbol out of the library and importing it.

Tuning Shared Library Code

This section explains things to consider in tuning shared library code:

- **Minimize the number of symbols exported** (see "Controlling Symbols to Be Exported or Loaded", page 45) for details.

- **Minimize global data**. All external data symbols are, of course, visible to applications. This can make maintenance difficult. Therefore, you should try to reduce global data.

  - Try to use automatic (stack) variables. Do not use permanent storage if automatic variables work. Using automatic variables saves static data space and reduces the number of symbols visible to application processes.

  - Determine whether variables really must be external and exported. Static symbols and hidden symbols are not visible outside the library, so they may change meanings between library versions. Only exported external variables must retain the same meaning.

  - Allocate buffers at run time instead of defining them at compile time. Allocating buffers at run time reduces the size of the library's data region for all processes and, thus, saves memory. Only processes that actually need the buffers get them. It also allows the size of the buffer to change from one release to the next without affecting compatibility. Statically allocated buffers cannot change size without affecting the addresses of other symbols and, perhaps, breaking compatibility.

- **Organize to improve locality**. When a function is in a.out files, it typically resides in a page with other code that is used more often (see "Exclude Infrequently Used Routines"). Try to improve locality of reference by grouping

dynamically related functions. If every call of `funcA` generates calls to `funcB` and `funcC`, try to put them in the same page.

The `cord`(1) command rearranges procedures to reduce paging and achieve better instruction cache mapping. You can use `cord` to see the number of cycles spent in a procedure and the number of times the procedure was executed. Use profiling to see what is actually called, as opposed to what may be called.

- **Align for paging**. The key is to arrange the shared library target's object files so that frequently used functions do not unnecessarily cross page boundaries. When arranging object files within the target library, be sure to keep the text and data files separate. You can reorder text object files without breaking compatibility; the same is not true for object files that define global data.

  For example, the IRIX operating system uses 4Kbyte pages. Using name lists and disassemblies of the shared library target file, the library developers determined where the page boundaries fell.

  After grouping related functions, they broke them into page-sized chunks. Although some object files and functions are larger than a single page, most of them are smaller. Then the developers used the infrequently called functions as glue between the chunks. Because the glue between pages is referenced less frequently than the page contents, the probability of a page fault decreased.

  After determining the branch table, they rearranged the library's object files without breaking compatibility. The developers put frequently used, unrelated functions together, because they would be called randomly enough to keep the pages in memory. System calls went into another page as a group, and so on. For example, the order of the library's object files became:

```
Before          After

#objects        #objects
   ...  .          ...
   printf.o        strcmp.o
   fopen.o         malloc.o
   malloc.o        printf.o
   strcmp.o        fopen.o
   ....            ...
```

Taking Advantage of QuickStart

QuickStart is an optimization designed to reduce start-up times for applications that link with DSOs. Each time `ld` builds a DSO, it updates a registry of shared objects. The registry contains the preassigned QuickStart addresses of a group of DSOs that typically cooperate by having locations that do not overlap. If you compile your application by linking with registered DSOs, your application takes advantage of QuickStart: all the DSOs are mapped at their QuickStart addresses, and `rld` wiill not need to move any of them to an unused address and perform a relocation pass to resolve all references.

Suppose you compile your application using the `-quickstart_info` flag, and QuickStart fails. It may fail because:

*   Your application has directly or indirectly linked with two different versions of the same DSO, as shown in Figure 3-1, page 41. In this example, `yourApp` links with `libyours.so`, `libmotif.so`, and `libc.so.1` on the compile line. When the DSO `libyours.so` was built, however, it linked with `libmalloc.so`, which in turn linked with `libc.so.1` when it was created. If the two versions of `libc.so.1` were not identical, `yourApp` will not be able to use QuickStart.
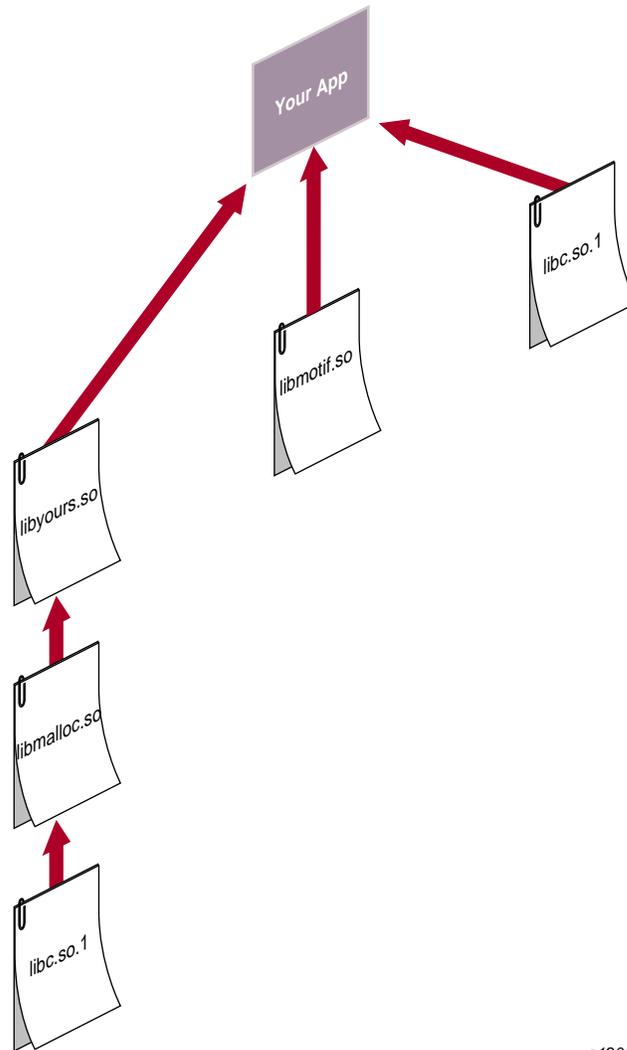
a12014

Figure 3-1 An Application Linked with DSOs

- You link with a DSO that cannot use QuickStart. This may occur because the DSO was not registered and therefore was assigned a location that overlaps with the location assigned to another DSO.

- Your application pulls in incompatible shared objects (in a manner similar to the example shown in Figure 3-1).

- Your application contains an unresolved reference to a function (where it takes the address of the function).

- The DSO links with another DSO that cannot use QuickStart.

Even if QuickStart officially succeeds, your application may have name space collisions and therefore may not start up as fast as it should. This is because `rld` has to bring in more information to resolve the conflicts. In general, you should avoid having conflicts both because of the detrimental effect on start-up time and because conflicts make it difficult to ensure the correctness of an application over time.

In the example shown in Figure 3-1, page 41, you may have written your own functions to allocate memory in `libmalloc.so` for `libyours.so` to use. If you did not use unique names for those functions (instead of `malloc()`, for example) the way this particular compile and link hierarchy is set up, the standard `malloc()` function defined in `libc.so.1` is used instead of the one defined in `libmalloc.so`.

Note: Conflicts are resolved by proceeding through the hierarchy from left to right and then moving to the next level (this is called breadth-first searching). "Searching for DSOs at Run Time", page 47, explains how the run-time linker searches for DSOs.

For example, suppose the diagram in Figure 3-1 corresponds to the following command:

```
% cc -lyours -lmotif -lc
```

Because shared objects mentioned on the command line always take precedence over those that are not mentioned, the preceding command uses the standard `malloc()` defined in `libc.so.1`.

To get your own version of `malloc()` defined in `libmalloc.so` for `libyours.so` to use, enter:

```
% cc -lyours -lmotif -lmalloc -lc
```

However, in both of the above examples, if -lyours contains `malloc()`, you will get that `malloc()`. (In the examples above, you do not need to specify -lc; it was added for clarity).

Thus, it is not a good idea to allow more than one DSO to define the same function. Even if the DSOs are synchronized for their first release, one of them may change the

definition of the function in a subsequent release. Of course, you can use conflicts to intentionally override function definitions; however, make sure you control what is overriding what.

If you use the `-quickstart_info` option, `ld` tells you if conflicts arise. It also tells you to run `elfdump` with the `-Dc` option to find the conflicts. See the `elfdump`(1) man page for more information about how to read the output produced by `elfdump`.

Building DSOs

In most cases, you can build DSOs as easily as archive libraries. If your library is written in a high-level language, such as C or Fortran, you will not have to make any changes to the source code. If your code is written in assembly language, you must modify it to produce PIC. This is described in the *MIPSpro Assembly Language Programmer's Guide*.

This section covers procedures to use when you build DSOs, and includes these topics:

- "Creating DSOs"

- "Making DSOs Self-Contained", page 44

- "Controlling Symbols to Be Exported or Loaded", page 45

- "Building DSOs with C++", page 46

Creating DSOs

To create a DSO from a set of object files, use `ld` with the `-shared` option:

```
% ld -shared stuff.o nonsense.o -soname libdada.so -o libdada.so
```

The preceding example creates a DSO, `libdada.so`, from two object files, `stuff.o` and `nonsense.o`. Note that DSO names should begin with `lib` and end with `.so`, for ease of use with the compiler driver's `-l`*lib* argument. If you are already building an archive library (`.a` file), you can create a DSO from the library by using the `-shared` and `-all` arguments to `ld`:

```
ld -shared -all libdada.a -soname libdada.so -o libdada.so
```

The `-all` argument specifies that all of the object files from the library, `libdada.a`, should be included in the DSO.

Note: It is best to use the -soname option. For example, if the -o *name* has an explicit path such as -o ../a/libdada.so, typically you want the -soname to be libdada.so.

Warning: It is essential that the *soname* of a DSO and its file name be congruent according to the versioning rules. (*Congruent* means that if the file name is, for example, xxxx.so.1, the *soname* must be *optional-path*/xxxx.so or xxxx.so.1). Usually, *optional-path* should be empty. A full path *soname* is used when a DSO must be accessed (at run time) from that specific location in the file system. A partial-path is used when the DSO must be accessed at that location from the current working directory. As such, a partial path is almost always a mistake.

The consequences of having a DSO file name and *soname* that are not congruent range from not being able to use the DSO at all to having all of multiple dlopen commands load a fresh copy, even though only one will actually be used.

Making DSOs Self-Contained

When building a DSO, be sure to include any archives required by the DSO on the link line so that the DSO is self-contained (that is, it has no unresolved symbols). If the DSO depends on libraries not explicitly named on the link line, subsequent changes to any of those libraries may result in name space collisions or other incompatibilities that can prevent any applications that use the DSO from doing a QuickStart. Such incompatibilities can also lead to unpredictable results over time as the libraries change asynchronously. Suppose you want to make the archive libmine.a into a DSO, and libmine.a depends on routines in another archive, libutil.a. In this case, include libutil.a on the link line:

```
% ld -shared -all -no_unresolved libmine.a -soname libmine.so \
-o libmine.so -none libutil.a
```

This causes the modules in libutil.a that are referenced in libmine.a to be included in the DSO, but these modules will not be exported. (For more information about exported symbols, see "Controlling Symbols to Be Exported or Loaded", page 45.) The -no_unresolved option causes a list of unresolved symbols to be created; generally, this list should be empty to enable using QuickStart.

Similarly, if a DSO relies on another DSO, be sure to include that DSO on the link line. For example:

```
% ld -shared -all -no_unresolved libbtree.a -soname libtree.so \
-o libtree.so -lyours
```

This example places `libyours.so` in the `liblist` of the new DSO, `libtree.so`. This ensures that `libyours.so` is loaded whenever an executable that uses `libtree.so` is launched. Again, symbols from `libyours.so` will not be exported for use by other libraries. (You can use the `-exports` flag to reverse this exporting behavior; the `-hides` flag specifies the default exporting behavior.)

Controlling Symbols to Be Exported or Loaded

Limiting the number of symbols exported has two effects that are important to DSO performance. These effects apply mainly to DSO startup costs rather than to the time spent executing the code in the application or DSO:

- Hidden symbols always resolve very quickly, but references to exported symbols can take longer to resolve. If there are n DSOs during execution, name lookup will take more than $n/2$ times as long. Hidden symbols have an update cost but do not have a real lookup cost so even if n is only 3 or 4, the hidden symbols are handled much faster than exported symbols.

- Hidden symbols do not conflict with other symbols. Conflicts slow down the startup of DSOs that are quickstarted. Conflicts are irrelevant in any DSO that does not quickstart and after any `dlopen()` is done, nothing will quickstart in that execution.

By default, to help avoid conflicts, symbols defined in an archive or a DSO that is used to build another DSO are not externally visible. You can explicitly export or hide symbols with the `-exported_symbol` (or `-exports`) and `-hidden_symbol` (or `-hides`) options :

```
-exported_symbol name1, name2, name3
-hidden_symbol name4, name5
```

By default, if you explicitly export any symbols, all other symbols are hidden. If you both explicitly export and explicitly hide the same symbol on the link line, the first occurrence determines the behavior. You can also create a file of symbol names (delimited by white space) that you want explicitly exported or hidden, and then refer to the file on the link line with either the `-exports_file` or `-hiddens_file` option:

```
-exports_file file
-hiddens_file file2
```

These files can be used in addition to explicitly naming symbols on the link line.

The -exports option is used in conjunction with the -shared or -call_shared options. It specifies that symbols from the next object, archive, or DSO be exported by the object being created. Similarly, the hides option specifies that symbols from the next object, archive, or DSO be hidden by the object being created. This is the default behavior for linking in archives or DSOs, but it is not for relocatable objects.

Another useful option, -delay_load, prevents a library from being loaded until it is actually referenced. Suppose, for example, that your DSO contains several functions that are likely to be used in only a few instances. Furthermore, those functions rely on another library (archive or DSO). If you specify -delay_load for this other library when you build your DSO, the run-time linker loads that library only when those few functions that require it are used. Note that if you explicitly export any symbols defined in a library that the run-time linker is supposed to delay loading, the export behavior takes precedence and the library is automatically loaded at run time.

Delay-loaded shared objects are not delay-loaded if direct references to data symbols exist in the delay-loaded object, or if the address of a function in the delay-loaded object is taken. That is, -delay_load is only effective with objects that have a purely functional interface (ld(1) will only set the -delay-load flag in the library list if delay-load will work properly).

Delay-loaded shared objects do not function properly if direct references to data symbols exist in the delay-loaded object, or if the address of the function in the delay-loaded object is used. Therefore, only use -delay_load to load shared objects that have a purely functional interface.

Note: You can build DSOs using cc. However, if you want to export symbols/files or use -delay_load, use ld to build DSOs.

Building DSOs with C++

It is recommended that you use the CC command rather than the ld command to build DSOs from C++ programs. The driver generates a lot of C++ specific arguments to ld, without which the DSO does not work. If you use templates, using CC to build your DSO also guarantees that templates get instantiated properly. For example:

```
% CC -shared -o libmylib.so   object file list
```

For example:

```
% CC -shared -o libmylib.so a.o b.o c.o
```

CC recognizes many of the `ld` options such as `-l` and `-L`; hence these options to `ld` work. However, most `ld` options do not work. If you want to specify other options, refer to the `CC`(1) and the `ld`(1) man pages. If the option is not described in the `CC` page, you may need to use the `-Wl,`*ld _option* syntax to tell the `CC` driver to pass *ld_option* to `ld`. See the `CC`(1) man page for details.

Run-Time Linking

This section explains the search path followed by the run-time linker and how you can cause symbols to be resolved at run time rather than link time. Specifically, this section describes:

- "Searching for DSOs at Run Time"

- "Run-Time Symbol Resolution", page 50

Searching for DSOs at Run Time

When you run a dynamically linked executable, the run-time linker, `rld`(1), identifies the DSOs required by the executable and loads the required DSOs. If necessary the IRIX kernel relocates DSOs within the process' virtual address space, so that no two DSOs occupy the same location. The program header of a dynamically linked executable contains a field, the **liblist**, which lists the DSOs required by the executable.

When looking for a DSO, `rld` searches directories in a specific sequence. This section covers run-time searching for the o32-bit, n32-bit, and 64-bit ABIs.

This section also describes environment variables that let you customize the search on your system. Each ABI has its own environment variable set. The o32 set usually applies to the other ABIs unless the other ABI environment variable is set. Only the `_RLD_ARGS` environment variable, which is not often used, is shared by all three ABIs.

Searching for DSOs at Run Time under the o32-Bit ABI

The (old) o32-bit ABI rules use the following sequence when searching for DSOs at run time:

1. `/usr/lib`

2. `/usr/lib/internal`

3. `/lib`

4. `/lib/cmplrs/cc`

5. `/usr/lib/cmplrs/cc`

6. `/opt/lib`

`RPATH` is a colon-separated list of directories stored in the main executable. You can set `RPATH` by using the `-rpath` argument to `ld`:

```
% ld -o myprog myprog.c -rpath /d/src/mylib -soname libmylib.so \
libmylib.so -lc
```

This example links the program against `libmylib.so` in the current directory and configures the executable such that `rld` searches the directory `/d/src/mylib` when searching for DSOs.

The `LD_LIBRARY_PATH` environment variable is a colon-separated list of directories to search for DSOs. This can be very useful for testing new versions of DSOs before installing them in their final location.

You can set the environment variable, `_RLD_ROOT` for the old 32-bit ABI, to a colon-separated list of directories. The run-time linker prepends these to the paths in `RPATH` and the paths in the default search path.

In all of the colon-separated directory lists, an empty field is interpreted as the current directory. A leading or trailing colon counts as an empty field. For example, if an application using the old 32-bit ABI sets `LD_LIBRARY_PATH` to the following:

`/d/src/lib1:/d/src/lib2:`

In this example, the run-time linker searches the directory `/d/src/lib1`, then the directory `/d/src/lib2`, and then the current directory.

> **Note:** The security policy is implemented in the IRIX kernel in IRIX 6.5 and later versions; for earlier versions of IRIX, it is implemented in `rld`. The current policy for honoring `rld` environment variables is as follows:
>
> Most `rld` environment variables are ignored for executables with no capabilities set (see the `capabilities`(4) man page) if both of the following are true:
>
> - The real user ID is not 0 (root).
> - One of the following is true:
>   - The real and effective (those active for the process) user IDs do not match.
>   - The real and effective group IDs do not match.
>
> If the environment or an executable has capabilities set, that executable will be treated as if it were a `setuid`(2) application. To check if your shell has capabilities set, use `id -P`. Use `su -C all=` to get a shell with no capabilities.

Searching for DSOs at Run Time under the n32-Bit ABI

The (new) n32-bit ABI rules use the following sequence when searching for DSOs at run time:

1. `/usr/lib32`

2. `/usr/lib32/internal`

3. `/lib32`

4. `/opt/lib32`

Setting the `_RLD32_ROOT` or the `LD_LIBRARYN32_PATH` environment variable overrides the default settings.

If `LD_LIBRARYN32_PATH` is not specified, `rld` honors `LD_LIBRARY_PATH`, if specified. As a result, if `LD_LIBRARY_PATH` is set for an old 32-bit program, it is recommended that you also set `LD_LIBRARYN32_PATH` to something (`""`, for example) to avoid having `LD_LIBRARY_PATH` apply accidentally to new 32-bit applications in that environment.

Searching for DSOs at Run Time under the 64-Bit ABI

The 64-bit ABI rules use the following sequence when searching for DSOs at run time:

1. `/usr/lib64`

2. `/usr/lib64/internal`

3. `/lib64`

4. `/opt/lib64`

Setting the `_RLD64_ROOT` or the `LD_LIBRARY64_PATH` environment variable overrides the default settings.

If `LD_LIBRARY64_PATH` is not specified, `rld` honors `LD_LIBRARY_PATH`, if specified. As a result, if `LD_LIBRARY_PATH` is set for an old 32-bit program, it is recommended that you also set `LD_LIBRARY64_PATH` to something (`""`, for example) to avoid having `LD_LIBRARY_PATH` apply accidentally to 64-bit applications in that environment.

Run-Time Symbol Resolution

Dynamically linked executables can contain symbol references that are not resolved before run time. Any symbol references in your main program or in an archive must be resolved at link time, unless you specify the `-ignore_unresolved` argument to `cc`.

DSOs may contain references that are not resolved at link time. All data symbols must be resolved at run time. If `rld` finds an unresolvable data symbol at run time, the executable exits with an error. Text symbols are resolved only when they are used, so a program can run with unresolved text symbols, as long as the unresolved symbols are not used.

You can force `rld` to resolve text symbols at run time by setting the environment variable `LD_BIND_NOW`. If unresolvable text symbols exist in your executable and you set `LD_BIND_NOW`, the executable exits with an error, as if there were unresolvable data symbols.

Building a DSO with `-Bsymbolic`

When you build a DSO with `-Bsymbolic`, the dynamic linker resolves referenced symbols from itself first. If the shared object fails to supply the referenced symbol, then the dynamic linker searches the executable file and other shared objects. For example:

`main` defines *x*

`x.so` defines and uses *x*

If you build `x.so` with `-Bsymbolic` on, the linker tries to resolve the use of `x` by looking first for the definition in `x.so` and then by looking in `main`.

In FORTRAN programs, the linker allocates space for `COMMON` symbols and the compiler allocates space for `BLOCK DATA`. The first kind of symbol (with `COMMON` blocks present) appears in the symbol table as `SHN_MIPS_ACOMMON` (uninitialized `DATA`) whereas the second kind of symbol (with `BLOCK DATA` present) appears as `SHN_DATA` (initialized `DATA`). In general, initialized data takes precedence when the dynamic linker tries to resolve a symbol. However, with `-Bsymbolic`, whatever is defined in the current object takes precedence, whether it is initialized or uninitialized.

Variables that are declared at file scope in C with `-cckr` are also treated this way. For example:

```
int foo[100];
```

is `COMMON` if `-cckr` is used and `DATA` if `-xansi` or `-ansi` is used.

For example:

In `main`:

```
COMMON i, j /* definition of i, j with initial values */
DATA i/1/, j/1/
CALL junk
END
```

In `x.so`:

```
SUBROUTINE junk
COMMON i, j
/* definition of i, j with NO initial values */
/* initialized by kernel to all zeros */
PRINT *, i, j
END
```

When you build `x.so` using `-Bsymbolic`, this program prints `0 0`. When you build `x.so` without `-Bsymbolic`, the program prints `1 1`.

Converting Archive Libraries to DSOs

When you link a program with a DSO, all of the symbols in the DSO become associated with the executable. This can cause unexpected results if archives that contain unresolved externals are converted to DSOs. When linking with a PIC archive, the linker links in only those object files that satisfy unresolved references.

If an object file in an archive contains an unresolved external reference, the linker tries to resolve the reference only when that object file is linked in to your program. In contrast, a DSO containing an external data reference that cannot be resolved at run time causes the program to fail. Therefore, use caution when converting archives with external data references to DSOs.

For example, suppose you have an archive, `mylib.a`, and one of the object files in the archive, `has_extern.o`, references an external variable, *foo*. As long as your program does not reference any symbols in `has_extern.o`, the program will link and run properly. If your program references a symbol in `has_extern.o` and does not define *foo*, then the link will fail. However, if you convert `mylib.a` to a DSO, then any program that uses the DSO and does not define *foo* will fail at run time, regardless of whether the program references any symbols from `has_extern.o`.

Two possible solutions exist for this problem.

- Add a "dummy" definition of the data to the DSO. A data definition appearing in the main executable preempts one appearing in the DSO itself. This may, however, be misleading for executables that use the portion of the DSO that needs the data, but that failed to define it in the main program.

- Separate the routines that use the data definition into a second DSO, and place dummy functions for them in the first DSO. The second DSO can then be loaded dynamically the first time any of the dummy functions is accessed. Each of the dummy functions must verify that the second DSO was loaded before calling the real function (which must have a unique name). This way, programs run whether or not they supply the missing external data, as long as they do not call any of the functions that require the data. The first time one of the dummy functions is called, it tries to dynamically load the second DSO. Programs that do not supply the missing data fail at this point.

For more information on dynamic loading, see "Dynamic Loading Under Program Control", page 53.

Dynamic Loading Under Program Control

IRIX provides a library interface to the run-time linker that allows programs to load and unload DSOs dynamically. The functions in this interface are part of `libc` (see Table 3-1).

Table 3-1 Functions to Load and Unload DSOs

Function	Action
dlopen()	Loads a DSO
dlsym()	Finds a symbol in a loaded DSO
dlclose()	Unloads a DSO
dlerror()	Reports errors
sgidlopen_version()	Loads a DSO
sgidladd_version()	Loads a DSO

You can dynamically load shared objects by using `sgidladd()`, which is similar to `dlopen(...,RTLD_LOCAL|...)`. However, unlike `dlopen()`, all the names in the shared object become available to satisfy references in shared objects during lazy text resolution. Furthermore, it is not necessary to use `dlsym()` to gain access to the symbols in the shared object. `sgidladd()` is available as part of `libc`. For more information, see the `sgidladd(3)` man page.

To load a DSO, call `dlopen()`:

```
include <dlfcn.h>
void *dlhandle;
     ..
dlhandle = dlopen("/usr/lib/mylib.so", RTLD_LAZY | RTLD_LOCAL);
if (dlhandle == NULL) {
    /* couldn't open DSO */
    printf("Error: %s\n", dlerror());
}
```

The first argument to `dlopen()` is the pathname of the DSO to be loaded. This may be either an absolute or a relative pathname. When you call this routine, the run-time linker tries to load the specified DSO. If any unresolved references exist in the executable that are defined in the DSO, the run-time linker resolves these references

on demand. You can also use dlsym() to access symbols in the DSO, whether or not the symbols are referenced in your executable.

When a DSO is brought into the address space of a process, it may contain references to symbols whose addresses are not known until the object is loaded. These references must be relocated before the symbols can be accessed. The second argument to dlopen() governs when these relocations take place.

This argument can have the following values:

- RTLD_LAZY: Under this mode, only references to data symbols are relocated when the object is loaded. References to functions are not relocated until a given function is invoked for the first time. This mode may result in better performance, since a process may not reference all of the functions in any given shared object.

- RTLD_NOW: Under this mode, all necessary relocations are performed when the object is first loaded. This may result in some wasted effort if relocations are performed for functions that are never referenced. However, this option is useful for applications that need to know as soon as an object is loaded that all symbols referenced during execution will be available.

- RTLD_GLOBAL: This mode modifies the treatment of the symbols in the DSO being opened to be identical to those of sgidladd(). RTLD_GLOBAL may be ORed with either RTLD_NOW or RTLD_LAZY (RTLD_GLOBAL cannot be the mode value on its own). See dlopen(3c) for details.

- RTLD_LOCAL: With this mode, the symbols in the dlopen DSO can only be referenced by dlsym; they cannot be accessed by symbol name. This is the default.

To access symbols that are not referenced in your program, use dlsym():

```
#include <dlfcn.h>
void *dlhandle;
int (*funcptr)(int);
int i,j;
     .. load DSO ...
funcptr = (int (*)(int)) dlsym(dlhandle, "factorial");
if (funcptr == NULL) {
    /* couldn't locate the symbol */
    exit();
}
i = (*funcptr)(j);
```

> **Note:** The cast to `(int (*) (int))` may produce a compiler warning about converting data pointers to function pointers. The warning is honoring the ANSI/ISO C standard; the cast and subsequent call work fine.

This example looks up the address of the function `factorial()` and assigns it to the function pointer `funcptr`.

If you encounter an error (`dlopen()` or `dlsym()` returns NULL), you can get diagnostic information by calling `dlerror()`. The `dlerror()` function returns a string describing the cause of the latest error. You should call `dlerror()` only after an error has occurred; at other times, its return value is undefined.

An application with multiple threads that calls these functions must provide its own locking because `dlerror()` is not thread specific.

To unload a DSO, call `dlclose()`:

```
#include <dlfcn.h>
void *dlhandle;
... load DSO, use DSO symbols ...
dlclose(dlhandle);
```

The `dlclose` function frees up the virtual address space that has been `mmap`ed by the `dlopen` call of that file (similar to a `munmap` call). The difference, however, is that a `dlclose` on a file that has been opened multiple times (either through `dlopen` or program startup) does not cause the file to be `munmap`ed until the file is no longer needed by the process.

Versioning of DSOs

This section describes the DSO versioning mechanism of SGI and includes the following topics:

- "The Versioning Mechanism"

- "What Is a Version?"

- "Building a Shared Library Using Versioning", page 56

- "Example of Versioning", page 57

The Versioning Mechanism

SGI uses a mechanism for the versioning of shared objects and executables. Note that this mechanism is outside the scope of the MIPS ABI, and, thus, must not be relied on for code that must be MIPS ABI-compliant and run on other vendors' platforms. Currently, all executables produced on SGI systems have a bit set that marks them as SGI_ONLY to allow use of the versioning mechanism.

Versioning is of interest mainly to developers of shared objects. It may not be of interest to you if you simply use shared objects. Versioning allows a developer to update a shared object in a way that may be incompatible with executables previously linked against the shared object. You can accomplish this by renaming the original shared object and providing it along with the (incompatible) new version.

What Is a Version?

A version is part or all of an identifying *version_string* that can be associated with a shared object by using the -set_version *version_string* option to ld when the shared object is created.

A *version_string* consists of one or more versions separated by colons (:). A single version has the form:

[*comment*#]sgi*major.minor*

where:

- *comment*: Specifies a comment string, which is ignored by the versioning mechanism. It consists of any sequence of characters followed by a pound sign (#). The comment is optional.

- sgi: Specifies the literal string sgi.

- *major*: Specifies the major version number, which is a string of digits [0-9].

- .: Specifies a literal period.

- *minor*: Specifies the minor version number, which is a string of digits [0-9].

Building a Shared Library Using Versioning

Follow these instructions when building your shared library:

When you first build your shared library, give it an initial version, for example, sgi1.0. Add the option –set_version sgi1.0 to the command to build your shared library (cc -shared, ld -shared).

Whenever you make a compatible change to the shared object, create another version by changing the minor version number (for example, sgi1.1) and add it to the end of the *version_string*. The command to set the version of the shared library now looks like -set_version ''sgi1.0:sgi1.1''.

When you make an incompatible change to the shared object:

1. Change the filename of the old shared object by adding a dot followed by the major number of one of the versions to the filename of the shared object. Do not change the *soname* of the shared object or its contents. Simply rename the file.

2. Update the major version number and set the *version_string* of the shared object (when you create it) to this new version; for example, -set_version sgi2.0.

This versioning mechanism affects executables in the following ways:

- When an executable is linked against a shared object, the last version in the shared object's *version_string* is recorded in the executable as part of the liblist. You can examine this using elfdump -Dl.

- When you run an executable, rld looks for the proper filename in its usual search routine.

- If a file is found with the correct name, the version specified in the executable for this shared object is compared to each of the versions in the *version_string* in the shared object. If one of the versions in the *version_string* matches the executable's version exactly (ignoring comments), then that library is used.

- If no proper match is found, a new filename for the shared object is built by combining the *soname* specified in the executable for this shared object and the *major* number found in the version specified in the executable for this shared object (*soname.major*). Remember that you did not change the *soname* of the object, only the filename. The new file is searched for using rld's usual search procedure.

Example of Versioning

For example, suppose you have a shared object foo.so with initial version sgi10.0. Over time, you make two compatible changes for foo.so that result in the following final *version_string* for foo.so:

```
initial_version#sgi10.0:upgrade#sgi10.1:new_devices#sgi10.2
```

You then link an executable that uses this shared object, useoldfoo. This executable specifies version sgi10.2 for *soname* foo.so. (Remember that the executable inherits the last version in the *version_string* of the shared object.)

The time comes to upgrade foo.so in an incompatible way. Note that the *major* version of foo.so is 10, so you move the existing foo.so to the filename foo.so.10 and create a new foo.so with the *version_string*:

```
efficient_interfaces#sgi11.0
```

New executables linked with foo.so use it directly. Older executables, like useoldfoo, attempt to use foo.so, but find that its version (sgi11.0) is not the version they need (sgi10.2). They then attempt to find a foo.so in the file name foo.so.10 with version sgi10.2.

Note: When a needed DSO has its interface changed, then a new version is created. If the interface change is not compatible with older versions, then a consuming shared object needs incompatible versions in order to use the new version, even if it does not use that part of the interface that is changed.

Loading Multiple DSOs

Options to the rld(5) command are available beginning with the MIPSpro 7.4.1 release that allow users to load multiple DSOs on a single page.

By default, every text and data segment must be on a separate IRIX page when loaded. Loading multiple DSOs onto a single page can reduce cache misses and may substantially improve startup and execution of some applications.

Loading multiple DSOs also gives users control over the areas of virtual memory that are used to load DSOs. This is true whether a big page size is requested or not.

The following example demonstrates memory layout with the new rld options. Assume that 1MB pages are used and three DSOs are loaded using u ordering. The memory layout will be as follows:

```
dsoa-text  4K beginning at beginning of page
dsoa-data  4K
dsob-test 16K
dsob-data  4K
```

```
dsoc-text  8K
dsoc-data  4K
        end-of-page space
```

A single page is needed for the 3 DSOs. IRIX 6.5.14 would use a minimum of 6 pages for the kernel to load DSOs.

Using `d` ordering, the following layout occurs:

```
beginning of page unused space
dsoc-text  8K
dsoc-data  4K
dsob-text 16K
dsob-data  4K
dsoa-text  4K
dsoa-data  4K ending at end of page on a single page
```

It is useful to experiment with different options to determine what is the best set of values for a given application. While experiments have shown as much as a 25% speedup in wall clock time for one particular application (gong from 16K to 256K pages), your results may vary.

Using `rld` 'Big Page' Options

To set up DSOs to use this feature, first decide the page size and the page allocation size. The page allocation size must be a multiple of the page size.

No special linking needs to be done. Text and data should be allocated sequentially near each other (this is the normal `ld` default and no special options are required.) Defaults are noted in the description of each option.

Options are passed to `rld` by changing the `_RLD_ARGS_BIG_PAGES` environment variable. Once set, the values apply to all child processes unless explicitly revoked by an application or by a shell script in the chain of processes.

Many of these options take a *number* as an argument, which is interpreted as follows:

nnnn: taken as decimal (for example, `1024`)
0X*nnn...*: taken as hexidecimal (for example, `0x1000`)
0*nnnn...*: taken as octal (for example, `01000`)

A trailing `K` indicates that the *number* should be multiplied by 1024. A trailing `M` means to multiply by 1024×1024. A trailing `G` means to multiply by 1024×1024×1024. For example, to specify a page size of 64K, the following are equivalent:

```
-big_page_page_pagesize 64K
-big_page_pagesize 65536
-big_page_pagesize 0x10000
```

The octal form, `-big_page_pagesize 0200000` can be used but it is not the ideal form to use because IRIX addresses and sizes are not normally given in octal.

If any option is repeated, the last one seen is used. However, duplicating options can have unforeseen effects and should be avoided.

The `-big_page_load` option

This is a required option and it indicates that big page loading is to be allowed. For `setuid` applications or when IRIX capabilities are in use, big page loading is not allowed and this option is ignored.

The other options are only honored if `-big_page_load` is in `_RLD_ARGS_BIG_PAGES` (that is, if it is not ignored due to `setuid` applications or IRIX capabilities).

The `-big_page_pagesize` *number* option

The *number* argument indicates the desired page size. Separate sizes for text and data are not provided because the text and data will be on the same page.

If not supplied, the default is 16K pages. Even if the normal machine default is different, if `-big_page_load` is given, `rld` will use the *number* provided with this option.

Any legal page size for the machine can be used. For IRIX64 machines, the minimum legal page size is 16K. See "Kernel-related Tuning Variables", page 63 for more information.

If this option is used, the `-big_page_mmap_size` option must also be used.

The `-big_page_mmap_size` *number* option

The *number* argument indicates, in bytes (not pages), the space that `rld` should mmap when it creates text and data pages. If provided, this must be greater than or equal to the `-big_page_pagesize` argument and it must be a multiple of that argument. If not set, the value used for `-big_page_pagesize` *number* is used.

If this is not large enough for a specific DSO to be loaded, `rld` adjusts the size that is mmapped for a DSO to be large enough.

See the `mmap`(2) man page for more details about memory mapping.

The `-big_page_allocation_preference` *set-of-triples* option

> **Note:** If this option is not specified, the appropriate default for each ABI is used.

This option specifies a set of triples that indicate how you want DSOs loaded into memory. Triples are separated by a colon (:) and are of the following form:

lowaddress–highaddress,{u|d}

The *highaddress* must be 1 past the address in the region. u or d specifies to `rld` that the addresses be taken from the bottom-up or from the top-down in a region.

No spaces are allowed in a triple or between triples. The following is a valid example:

```
0x400000-0x10000000,u:0x40000000-0x78000000,d
```

The following is an **invalid** example:

```
0x400000-0x10000000,u: 0x40000000-0x78000000,d
```

The following examples demonstrate this option; for ease of viewing, the option has been split into separate lines.

The default `-o32/-n32` allocation is as follows:

```
-big_page_allocation_preference
0x400000-0x0f000000,u:
0x60000000-0x78000000,u:
0x40000000-0x60000000,u:
0x20000000-0x30000000,u
```

One strategy for `-n32` use (not the default) is to keep DSOs high, as in this example:

```
-big_page_allocation_preference
0x400000-0x0f000000,u:
0x40000000-0x78000000,d:
0x20000000-0x30000000,u
```

This example allocates DSOs in the upper area second and moving down from the stack area. If the application stack will not grow too much, the upper end of 0x78000000 could be made higher in this example. There is no limit in the number of areas selected this way, though there is a cost in rld CPU cycles if a long list is used. It is best to keep the list short.

The default for -64 is as follows:

```
-big_page_allocation_preference
0x400000-0x0f000000,u:
0x8000000000-0xfff8000000,u:
0x40000000-0x8000000000,u:
0x20000000-0x30000000,u
```

For Origin/Onyx 300 and 3000 and Fuel workstations, the default shown above is not the widest possible DSO loading map. There are more bits of address space than this default shows.

The region 0x30000000 to 0x4000000 is reserved by the o32 ABI for mmap space by default, but it can be used for DSO loading.

When allocating space, the rld command is aware of what is used for malloc and other system allocations and does not use space that is already in use.

If ranges overlap, big page allocation is ignored and a message is issued.

Tuning the big_page_allocation_preference setting is less important for applications compiled with -64 than for -n32 or -32 because the address space is large enough with -64 so that very few applications will need DSO placement. Tuning this setting may be useful, though, to avoid problems of interference with shared memory or mmap fixed allocation regions that an application might use.

The -big_page_cache_misalign option

This option specifies that rld should attempt to make small misalignments of the address where the DSO is loaded so that bits 13, 14, and 15 of successive DSOs that rld loads will have different values. The default is OFF.

The existing _RLD_ARGS options are changed as follows:

• The -v option reports when the new loading was successful and where the loading took place. This allows you to differentiate the loading as a big page loading and not the default style of loading.

- The `-debug_map` option is supported only in `rld.debug` and reports the details of what is done and what is attempted (mmap, DSO loading, region changes, and details of the derivation of load addresses). See the `rld`(5) man page for details about `rld.debug`.

Interaction with Other IRIX Facilities

Big page loading can affect how other IRIX facilities and executables work. This section describes some of those effects.

Interaction with `dplace`

Because `rld` uses the same kernel facilities as `dplace`(1) and because `rld` sets preferences after `dplace` runs, the `rld` settings supercede the `dplace` settings for data. The `rld` command uses `pm_setdefault(pm_data, MEM_DATA)` to set the page size. The big page options to `rld` affect both the text and data of the DSOs that `rld` is able to load with big pages. See the `pm`(3) man page for information.

If the big page options are not used, DSO text is not put on big pages but is always in base system page size units because that is how IRIX loads and shares DSO text.

If big page options are used, the `rld` big page options take effect, the `dplace` data settings are overwritten, and the requested page sizes are used for DSOs.

In no instance will the `rld` big page options affect the size of the pages used for an executable or `lib.so.1` or for `rld` itself.

Monitoring Page Usage

The `dlook`(1) command can be used to examine a running executable. The result shows the page sizes that are actually used in a process. See the `dlook`(1) man page for details.

The `numa_view` command can also be used to view large page use. See the `numa_view`(1) man page for details.

Kernel-related Tuning Variables

The following `systune` IRIX tuning parameters must be set appropriately to allow IRIX to supply the large pages as requested by `rld`:

```
percent_totalmem_16m_pages
percent_totalmem_4m_pages
percent_totalmem_1m_pages
percent_totalmem_256k_pages
percent_totalmem_64km_pages
percent_totalmem_16k_pages
```

Another set of kernel-related tuning variables are `coalesced_run_period` and `coalesced_precoalesce`. The setting of `coalesced_precoalesce=1` and `coalesced_run_period=60`, in addition to setting `percent_totalmem_4m_pages`, gave test runs that generated useful 4M pages, improved TLB misses (as reported by `perfex`) and better wall clock times.

Tuning and `rld`

Note: Tuning with the `_RLD_ARGS_BIG_PAGES` and `DT_MIPS_DIRECT` environment variables is only useful for codes in MIPSpro 7.4.1 or patch 4711 and later MIPSpro versions. Earlier versions of `rld` ignore these tuning features, so there is no adverse compatibility issue.

It is an important part of tuning to hide all symbols that are referred to only within a given DSO. Marking symbols by using `STO_HIDDEN`(/usr/include/elf.h, /usr/include/sys/elf.h) can significantly reduce the effort that `rld` expends at run time. It is not unusual that 80% or more of the so-called global symbols in a DSO are, in fact, never referred to from outside the DSO.

Use the following linking options to assist in tuning:

```
-exports
-exports_file
-exported_symbol
-hiddens-file
-hidden_symbol
-hides
```

Two other linker options used at DSO creation can assist better `rld` performance:

- `-Bsymbolic`: this option directs `rld` to search within this DSO first to resolve symbols. If the symbol is not found, the search continues using the standard symbol lookup, starting with the executable's global symbols. The dynamic section `DT_SYMBOLIC` flag is set in the DSO.

- -Bdirect: this option directs `rld` to assume that the DSO where the symbol was found (if found) at link time is the correct resolution of the symbol. For this symbol, the static-link-time table that records the resolution DSO is used directly. The dynamic section `DT_MIPS_DIRECT` flag is set in the DSO. Symbol preemption is therefore not possible for items called from the DSO marked `DT_MIPS_DIRECT`.

If both -Bsymbolic and -Bdirect are given on a single DSO, -Bdirect is applied because using a single DSO is harmless.

If lookup fails using the rule for -Bsymbolic or -Bdirect, normal `rld` lookup rules are invoked.

Using these options with hidden or exported symbols is appropriate; the symbolic and direct options apply to non-hidden symbols.

Optimizing Program Performance

This chapter describes the compiler optimization facilities and their benefits, and explains the major optimizing techniques. This chapter includes the following topics:

- "Optimization Overview", page 68, provides an overview of optimization benefits and debugging.

- "Performance Tuning with Interprocedural Analysis (IPA)", page 68, describes performance tuning with interprocedural analysis.

- "Controlling Loop Nest Optimizations (LNO)", page 75, discusses ways of controlling loop nest optimizations.

- "Controlling Floating-Point Optimization", page 95, explains methods of controlling floating-point optimization.

- "Controlling Other Optimizations with the -OPT Option", page 101 describes how to control miscellaneous optimizations.

- "Controlling Execution Frequency", page 104 describes how to control execution frequency.

- "The Code Generator ", page 105, describes the code generator.

- "Reordering Code Regions", page 117, describes methods of optimizing the location of code regions within your program

- "Programming Hints for Improving Optimization", page 120, lists programming hints for improving optimization

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

See your compiler manual for information about the optional parallelizer, apo, and the OpenMP directives and routines for a method of using a portable method to code parallelization into your program. You can find additional information about optimization in the *MIPSpro 64-Bit Porting and Transition Guide*. For information about writing code for 64-bit programs, see Chapter 5, "Coding for 64-Bit Programs", page 127. For information about porting code to -n32 and -64, see Chapter 6, "Porting Code to N32 and 64-Bit SGI Systems", page 133.

Optimization Overview

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

Optimize your program only when it is fully developed and debugged. To debug a program, you can use the -g option. Note that you can also use -DEBUG:*options* to debug run-time code and generate compile, link, and run-time warning messages.

Debug a program before optimizing it, because the optimizer may move operations around so that the object code does not correspond in an obvious way to the source code. These changed sequences of code can create confusion when using a debugger. For information on the debugger, see *dbx User's Guide* and *ProDev WorkShop: Debugger User's Guide*.

The compiler optimization options -O0 through -O3 determine the optimization level to be applied to your program. The -O0 option applies no optimizations, and the -O3 option performs the most aggressive optimizations. See your compiler manual and man page for more information.

Performance Tuning with Interprocedural Analysis (IPA)

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

Note: If you are using the automatic parallelizer (-apo), run it after IPA. If you apply parallelization to subroutines in separate modules, and then apply inlining to those modules using -IPA, you inline parallelized code into a main routine that is not compiled to initialize parallel execution. Therefore, you must use the parallelizer when compiling the main module as well as any submodules.

Currently IPA optimizes code by performing the following functions:

• Procedure inlining

• Interprocedural constant propagation

- Dead function elimination

- Identification of global constants

- Dead variable elimination

- PIC optimization

- Automatic selection of candidates for the gp-relative area (autognum)

- Dead call elimination

- Automatic internal padding of `COMMON` arrays in Fortran

- Interprocedural alias analysis

Figure 4-1, page 70, shows the interprocedural analysis and interprocedural optimization phase of the compilation process.
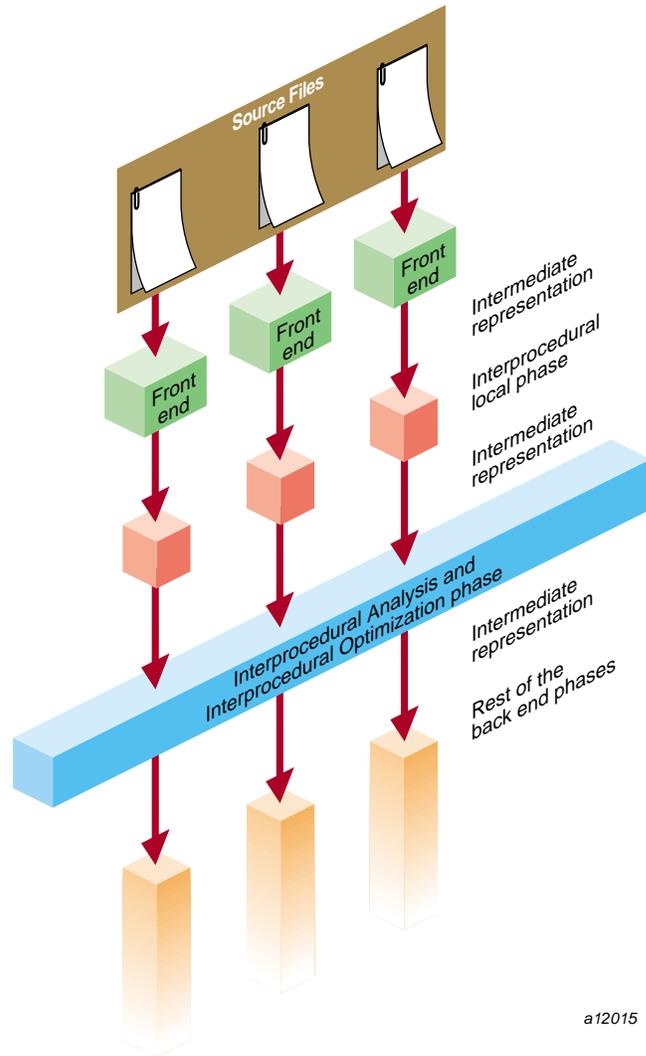
a12015

Figure 4-1 Compilation Process Showing Interprocedural Analysis

Typically, you invoke IPA with the -IPA: option group to f77, f90, cc, CC, and ld. Its inlining decisions are also controlled by the -INLINE: option group. Up-to-date information on IPA and its options is in the ipa(5) man page.

This section covers some IPA options including:

- "Inlining"

- "Common Block Padding", page 73

- "Alias and Address Taken Analysis", page 75

Inlining

IPA performs across and within file inlining. A default inlining heuristic determines which calls to inline.

Your code may benefit from inlining for the following reasons:

- Inlining exposes a larger context to the scalar and loop-nest optimizers, thereby allowing more optimizations to occur.

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

Inlining Options for Routines

You may wish to select certain procedures to be inlined or not to be inlined by using the inlining options.

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

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

- **The -INLINE:none and -INLINE:all Options**

  Changes the default inlining heuristic.

  The `-INLINE:all` option. Attempts to inline all routines that are not excluded by a `never` option or a routine pragma suppressing inlining, either for the routine or for a specific callsite.

  The `-INLINE:none` option. Does not attempt to inline any routines that are not specified by a `must` option or a pragma requesting inlining, either for the routine or for a specific callsite.

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

- **The -INLINE:must and -INLINE:never Options**

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

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

  The `-INLINE:never=`*routine_name<,routine_name>** option. Does not inline the specified routines at call sites not marked by inlining pragmas; it observes site inlining pragmas.

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

- **The -INLINE:**_file=<filename>_ **Option**

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

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

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

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

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

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

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

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

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

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

Common Block Padding

Power of two arrays can lead to degenerate behavior on cache-based machines. The IPA phases try, when possible, to pad the leading dimension of arrays to avoid cache conflicts. Several restrictions exist that limit IPA padding of common arrays. If the restrictions are not met, the arrays are not padded. The current restrictions are as follows:

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

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

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

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

```
        ....
        ....
    end
```

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

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

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

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

4. Restricted types of equivalences to arrays that may be padded are allowed. Equivalences that do not intersect with any column of the array are allowed. This implies an equivalencing that will not cause the equivalenced array to access invalid locations. In the following example, the arrays in common /a/ will not be padded since z is equivalenced to x(2,1), and hence z(1024) is equivalenced to x(1,2).

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

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

6. The common block symbol can not be referenced by regular object files that have been linked with the program.

Alias and Address Taken Analysis

The optimizations that are performed later in the compiler are often constrained by the possibility that two variable references may be *aliased*, meaning they may be aliased to the same address. This possibility is increased by calls to procedures that are not visible to the optimizer, and by taking the addresses of variables and saving them for possible use later (for example, in pointers).

Furthermore, the compiler must normally assume that a global (external) datum may have its address taken in another file, or may be referenced or modified by any procedure call. The IPA alias and address-taken analyses are designed to identify the actual global variable addressing and reference behavior so that such worst-case assumptions are not necessary.

The `-IPA:alias=ON` Option

This option performs IPA alias analysis. That is, it determines which global variables and formal parameters are referenced or modified by each call, and which global variables are passed to reference formal parameters. This analysis is used for other IPA analyses, including constant propagation and address-taken analysis. This option is `ON` by default.

The `-IPA:addressing=ON` Option

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

Controlling Loop Nest Optimizations (LNO)

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

This section covers the following topics:

- "Running LNO", page 76

- "LNO Optimizations ", page 78

- "Compiler Options for LNO ", page 84

- "Pragmas and Directives for LNO ", page 85

For complete information on options, see the lno(5) man page.

Running LNO

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

After LNO performs high-level transformations, it may be desirable to view the transformed code in the original source language. Two translators that are integrated into the back end translate the compiler internal code representation back into the original source language after the LNO transformation (and IPA inlining). You can invoke either one of these translators by using the Fortran option -FLIST:=on or the cc option -CLIST:=on. For example, f77 -O3 -FLIST:=on x.f creates an a.out as well as a Fortran file x.w2f.f. The .w2f.f file is a readable file, and it is usually a compilable SGI Fortran representation of the original program after the LNO phase (see Figure 4-2, page 77). LNO is not a preprocessor, which means that recompiling the .w2f.f file directly may result in an executable that is different from the original compilation of the .f file.

Use the -CLIST:=on option to cc to translate compiler internal code to C. No translator exists to translate compiler internal code to C++. When the original source language is C++, the generated C code may not be compilable.
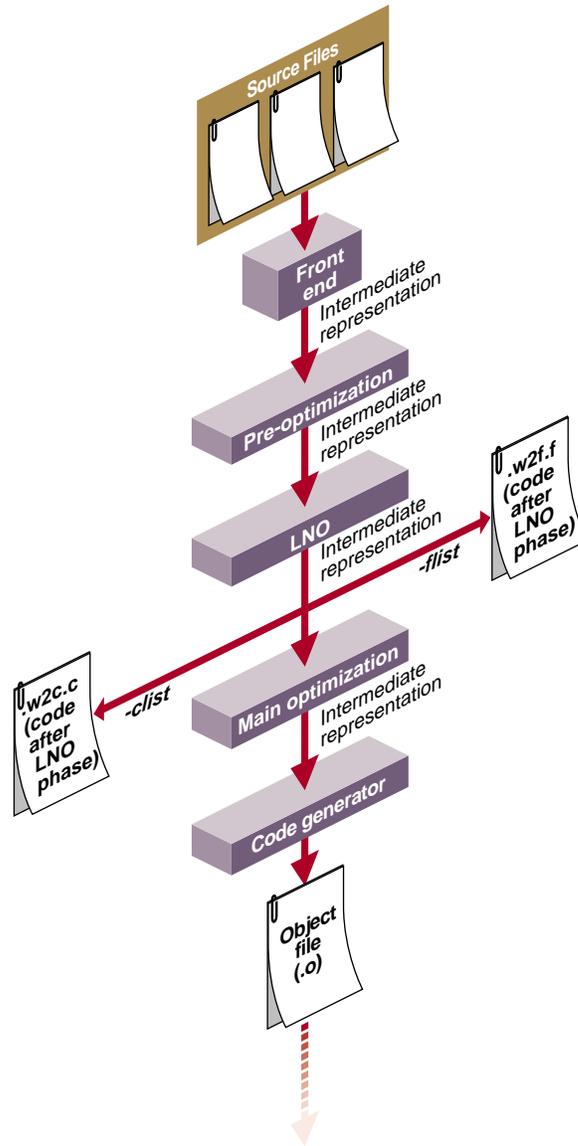
Figure 4-2 Compilation Process Showing LNO Transformations

LNO Optimizations

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

- "Loop Interchange"

- "Blocking and Outer Loop Unrolling ", page 79

- "Loop Fusion", page 80

- "Loop Fission/Distribution", page 81

- "Prefetching", page 83

- "Gather-Scatter Optimization", page 83

Loop Interchange

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

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

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

Interchanging the loops improves performance.

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

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

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

Blocking and Outer Loop Unrolling

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

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

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

```
do i=1,10000
   do j=1,10000,2
    do k=1,10000
      c(i,j) = c(i,j) + a(i,k)*b(k,j)
      c(i,j+1) = c(i,j+1) + a(i,k)*b(k,j+1)
```

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

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

```
do tilej=1,10000,Bj
   do tilek=1,10000,Bk
```

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

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

LNO automatically caches tile loops with block sizes appropriate for the target machine. When compiling for an SGI R8000, LNO uses a single level of blocking. When compiling for an SGI systems (such as R4000, R5000, R10000, or R12000) that contain multi-level caches, LNO uses multiple levels of blocking where appropriate.

Loop Fusion

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

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

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

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

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

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

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

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

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

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

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

To fuse aggressively, use -LNO:fusion=2.

Loop Fission/Distribution

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

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

Using loop fission, as shown in the following example, also enables loop interchange and blocking.

```
do i1
    do j1
      S1
    end do
end do
do i2
    do j2
      S2
    end do
end do
```

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

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

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

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

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

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

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

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

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

Prefetching

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

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

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

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

Prefetching can improve performance in compute-intensive operations where data is too large to fit in the cache. Conversely, prefetching will not help performance in a memory-bound loop where data fits in the cache.

Gather-Scatter Optimization

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

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

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

```
inc = 0 do i = 1,n
   tmp(inc) = i
   if (t(i) .gt. 0.0) then
     inc = inc + 1
   end do
end do
do i = 1,inc
   a(tmp(i)) = 2.0*b((tmp(i)-1)
end do
```

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

Compiler Options for LNO

The `lno`(5) man page describes the compiler options for LNO. Specifically, topics include:

- Controlling LNO Optimization Levels (described below)

- Controlling Fission and Fusion (described below)

- Controlling Gather-Scatter

- Controlling Cache Parameters

- Controlling Permutation Transformations and Cache Optimization

- Controlling Prefetch

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

The options `-r5000`, `-r8000`, `-r10000`, and `-r12000` set a series of default cache characteristics. To override a default setting, use one or more of the options below.

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

For example, on a machine with 128 Mbytes of main memory, you can block for it by using parameters, for example, `-LNO:cs3=128M:ls3=....` In this case, `assoc3` is ignored and does not have to be specified. Instead, you must specify `is_mem3..`, since virtual memory is fully associative.

Pragmas and Directives for LNO

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

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

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

This section covers:

* Fission/Fusion

* Blocking and Permutation Transformations

* Prefetch

- Fill/Align Symbol

- Dependence Analysis

See the `lno`(5) man page, or your compiler manual, for more information on these pragmas and directives.

Fission/Fusion

The following pragmas/directives control fission and fusion.

```
C*$* AGGRESSIVE INNER LOOP FISSION
#pragma aggressive inner loop fission
```

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

```
C*$* FISSION
#pragma fission
C*$* FISSIONABLE
#pragma fissionable
```

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

```
C*$* FUSE
#pragma fuse
C*$* FUSABLE
#pragma fusable
```

> Fuse the following loops, which must be immediately adjacent. The default is 2,level. Fusion is attempted on each pair of adjacent loops and the level, by default, is the determined by the maximal perfectly nested loop levels of the fused loops although partial fusion is allowed. Iterations may be peeled as needed during fusion and the limit of this peeling is 5 or the number specified by the -LNO:fusion_peeling_limit option. No fusion is done for non-adjacent outer loops. When the FUSABLE pragma is present, no validity test is done and the fusion is done up to the maximal common levels.

```
C*$* NO FISSION
#pragma no fission
```

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

```
C*$* NO FUSION
#pragma no fusion
```

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

Blocking and Permutation Transformations

The following pragmas/directives control blocking and permutation transformations.

```
C*$* INTERCHANGE
#pragma interchange
```

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

```
C*$* NO INTERCHANGE
#pragma no interchange
```

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

```
C*$* BLOCKING SIZE [(n1,n2)]
#pragma blocking size (n1,n2)
```

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

For example:

```
subroutine amat(x,y,z,n,m,mm)
real*8 x(100,100), y(100,100), z(100,100)
```

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

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

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

This pragma suggests that the compiler generates a nest like:

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

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

C*$* NO BLOCKING
#pragma no blocking

> Prevent the compiler from involving this loop in cache blocking.

```
C*$* UNROLL
#pragma unroll
```

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

For example, the following code:

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

becomes:

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

and not:

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

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

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

```
C*$* BLOCKABLE(I,J,K)
#pragma blockable (i,j,k)
```

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

Prefetch

The following pragmas/directives control prefetch operations.

```
C*$* PREFETCH
#pragma prefetch
```

> Specify prefetching for each level of the cache. Scope: entire function containing the pragma.
>
> 0 prefetching off (default for all processors except R10000 and R12000)
>
> 1 prefetching on, but conservative (default at –03 when prefetch is on)
>
> 2 prefetching on, and aggressive

```
C*$* PREFETCH_MANUAL
#pragma prefetch_manual
```

> Specify whether manual prefetches (through pragmas) should be respected or ignored. Scope: entire function containing the pragma.
>
> 0 ignore manual prefetches (default for all processors except R10000 and R12000)
>
> 1 respect manual prefetches (default at –03 for R10000 and beyond)

```
C*$* PREFETCH_REF
#pragma prefetch_ref
```

> The object specified is the reference itself; for example, an array element A(i, j). The object can also be a scalar, for example, x, an integer. Must be specified.
>
> A specified stride prefetches every stride iterations of this loop. Optional; the default is 1.
>
> The specified level specifies the level in memory hierarchy to prefetch. Optional; the default is 2.
>
> lev=1: prefetch from L2 to L1 cache.
>
> lev=2: prefetch from memory to L1 cache.
>
> kind specifies read/write. Optional; the default is write.
>
> size specifies the size (in Kbytes) of this object referenced in this loop. Must be a constant. Optional.

The effect of this pragma is:

- Generate a prefetch and connect to the specified reference (if possible).

- Search for array references that match the supplied reference in the current loop-nest. If such a reference is found, then that reference is connected to this prefetch node with the specified latency. If no such reference is found, then this prefetch node stays free-floating and is scheduled "loosely."

- Ignore all references to this array in this loop-nest by the automatic prefetcher (if enabled).

- If the size is specified, then the auto-prefetcher (if enabled) uses that number in its volume analysis for this array.

- No scope, just generate a prefetch.

```
C*$* PREFETCH_REF_DISABLE
#pragma prefetch_ref_disable
```

> This explicitly disables prefetching all references to the specified array in the current loop nest. The auto-prefetcher runs (if enabled), ignoring the array.

Fill/Align Symbol

The following pragmas and/or directives control fill and/or alignment of symbols. This section uses the term pragma when describing either a pragma or a directive.

The `align_symbol` pragma aligns the start of the named symbol at the specified alignment. The `fill_symbol` pragma pads the named symbol.

```
C*$* FILL_SYMBOL
#pragma fill_symbol
C*$* ALIGN_SYMBOL
#pragma align_symbol
```

> The `fill_symbol/align_symbol` pragmas take a symbol, that is, a variable that is a Fortran COMMON, a C/C++ global, or an automatic variable (but not a formal and not an element of a structured type like a struct or an array).

> The second argument in the pragma may be one of the keywords:

> - *L1cacheline* (machine specific first-level cache line size, typically 32 bytes)

> - *L2cacheline* (machine specific second-level cache line size, typically 128 bytes)

> - *page* (machine specific page size, typically 16 Kbytes)

> - *a user-specified power-of-two value*

> The `align_symbol` pragma aligns the start of the named symbol at the specified alignment, that is, the symbol "s" will start at the specified alignment boundary.

> The `fill_symbol` pragma pads the named symbol with additional storage so that the symbol is assured not to overlap with any other data item within the storage of the specified size. The additional padding required is heuristically divided between each end of the

specified variable. For instance, a `fill_symbol` pragma for the *L1cacheline* guarantees that the specified symbol does not suffer from false-sharing for the L1 cache line.

For global variables, these pragmas must be specified at the variable definition, and are optional at the declarations of the variable.

For COMMON block variables, these pragmas are required at each declaration of the COMMON block. Since the pragmas modify the allocated storage and its alignment for the named symbol, inconsistent pragmas can lead to undefined results.

The `align_symbol` pragma is ineffective for local variables of fixed-size symbols, such as simple scalars or arrays of known size. The pragma continues to be effective for stack-allocated arrays of dynamically determined size.

A variable cannot have both `fill_symbol` and `align_symbol` pragmas applied to it.

Examples:

```
int x;  /* x is a global or a common block variable */
#pragma align_symbol (x, 32)

/* x will start at a 32-byte boundary */

#pragma align_symbol (x, 2)
/* Error: cannot request alignment lower than the natural
   * alignment of the symbol.
   */

double y; /* y is a global, common, or local */
#pragma fill_symbol (y, L2cacheline)
/* allocate extra storage both before and after ``y'' so
 * that ``y'' is within an L2cacheline (128 bytes) all by
 * itself. Can be useful to avoid false-sharing between
 * multipleprocessors for cacheline containing ``y''.
 */
```

Dependence Analysis

The following pragmas/directives control dependence analysis.

```
CDIR$ IVDEP
#pragma ivdep
```

> Liberalize dependence analysis. This applies only to inner loops.
> Given two memory references, where at least one is loop variant,
> ignore any loop-carried dependences between the two references.
>
> For example:
>
> ```
> CDIR$ IVDEP
>    do i = 1,n
>       b(k) = b(k) + a(i)
>    enddo
> ```
>
> ivdep does not break the dependence since b(k) is not loop variant.
>
> ```
> CDIR$ IVDEP
>    do i=1,n
>       a(i) = a(i-1) + 3.0
>    enddo
> ```
>
> ivdep does break the dependence, but the compiler warns the user
> that it is breaking an obvious dependence.
>
> ```
> CDIR$ IVDEP
>    do i=1,n
>       a(b(i)) = a(b(i)) + 3.0
>    enddo
> ```
>
> ivdep does break the dependence.
>
> ```
> CDIR$ IVDEP
>    do i = 1,n
>       a(i) = b(i)
>       c(i) = a(i) + 3.0
>    enddo
> ```
>
> ivdep does not break the dependence on a(i) since it is within an
> iteration.
>
> If -OPT:cray_ivdep=TRUE, use Cray semantics. Break all lexically
> backward dependences. For example:
>
> ```
> CDIR$ IVDEP
>    do i=1,n
> ```

```
              a(i) = a(i-1) + 3.0
        enddo
```

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

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

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

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

-OPT:cray_ivdep and -OPT:liberal_ivdep are OFF (FALSE) by default.

Controlling Floating-Point Optimization

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

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

Two of these options are the preferred controls:

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

- -OPT:IEEE_arithmetic=*n* deals with the extent to which the generated code conforms to IEEE 754 standards for discrete IEEE-specified operations (for example, a divide or a square root).

-OPT:roundoff=*n*

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

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

roundoff=0

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

roundoff=1

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

roundoff=2

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

Reassociation can have a substantial effect on the performance of software pipelined loops by breaking recurrences. This is therefore the default for optimization level -O3.

roundoff=3

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

-OPT:IEEE_arithmetic=*n*

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

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

-OPT:IEEE_arithmetic=1

No degradation: do no transformations that degrade floating-point accuracy from IEEE requirements. The generated code may use instructions such as madd, which provides greater accuracy than required by IEEE 754. This is the default.

-OPT:IEEE_arithmetic=2

Minor degradation: allow transformations with limited effects on floating point results, as long as exact results remain exact. This option allows use of the MIPS4 recip and rsqrt operations.

-OPT:IEEE_arithmetic=3

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

As an example, consider optimizing the following 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)`. For example, on the MIPS R8000 and R10000, 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 and R10000.

Using `roundoff=2`, the loop may be reordered. For example, 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 is mathematically equal to `SQRT(r*r + r*r)`. However, calculating it this way causes an overflow if `2*r*r` is greater than the maximum REAL value, even though a representable result can be calculated for a much wider range of values of *r* (at greater cost). Both of these transformations are forbidden for `roundoff=2`, but enabled for `roundoff=3`.

Other Options to Control Floating Point Behavior

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

-OPT:div_split

> Enable/disable the calculation of x/y as x*(1.0/y), normally enabled by IEEE_arithmetic=3. Simplifies expressions by determining if (A/B) should be turned into (1/B)*A. This can be useful if B is a loop-invariant, as it replaces the divide with a multiply. For example, X = A/B becomes X = A*(1/B). See -OPT:recip.

-OPT:fast_complex

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

-OPT:fast_exp

> 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 labeled intrinsic in <math.h> (the default in -xansi and -cckr modes).

-OPT:fast_io

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

-OPT:fast_sqrt

> Enable/disable the calculation of square root as x*rsqrt(x) for -mips4, normally enabled by IEEE_arithmetic=3. This option is ignored for the R10000.

-OPT:fold_reassociate

> Enable/disable transformations that reassociate or distribute floating point expressions. This option is on at -O3, or if roundoff >= 2. For example, X + 1. X can be turned into X - X + 1.0, which will then simplify to 1. This can cause problems is X is large compared to 1, so that X+1 is X due to roundoff.

-OPT:IEEE_comparisons

> Force comparisons to yield results conforming to the IEEE 754 standard for NaN and Inf operands, normally disabled. Setting this option disables certain optimizations like assuming that a comparison x==x is always TRUE (since it is FALSE if x is a NaN). It also disables optimizations that reverse the sense of a comparison, for example, turning ``x < y'' into ``! (x >= y)'', since both ``x<y'' and ``x>=y'' may be FALSE if one of the operands is a NaN.

-OPT:recip

> Allow use of the MIPS IV reciprocal instruction for 1.0/y, normally enabled by -O3 or IEEE_arithmetic>=2. See -OPT:div_split. For example, X = 1./Y generates the recip instruction instead of a divide instruction. This may change the results slightly.

-OPT:rsqrt

> Allow use of the MIPS IV reciprocal square root instruction for 1.0/sqrt(y), normally enabled by -O3 or IEEE_arithmetic>=2. For example, X = 1./SQRT(Y) generates the rsqrt instruction instead of a divide and a square root. This may change the results slightly. This option is ignored for the R10000.

-TARG:madd

> The MIPS IV architecture supports fused multiply-add instructions, which add the product of two operands to a third, with a single roundoff step at the end. Because the product is not separately rounded, this can produce slightly different (but more accurate)

results than a separate multiply and add pair of instructions. This is normally enabled for −mips4.

Debugging Floating-Point Problems

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

If you encounter such effects when using these options (including −O3, which enables -OPT:roundoff=2 by default), first attempt to identify the cause by forcing the safe levels of the options: -OPT:IEEE_arithmetic=1:roundoff=0. When you do this, do not have the following options explicitly enabled:

```
-OPT:div_split
-OPT:fast_complex
-OPT:fast_exp
-OPT:fast_sqrt
-OPT:fold_reassociate
-OPT:recip
-OPT:rsqrt
```

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

Controlling Other Optimizations with the **–OPT** Option

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

- "Using the -OPT:Olimit Option", page 102
- "Using the -OPT:alias Option", page 102
- "Simplifying Code with the -OPT Option", page 104

Using the **-OPT:Olimit** Option

-OPT:Olimit

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

Using the **-OPT:alias** Option

-OPT:alias=*name*

> The compilers must typically be very conservative in optimization of memory references involving pointers (especially in C), since aliases (that is, different ways of accessing the same memory) may be very hard to detect. This option may be used to specify that the program being compiled avoids aliasing in various ways. The -OPT:alias options are listed below.

-OPT:alias=any

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

-OPT:alias=typed

> The compiler assumes that any pair of memory references that reference distinct types in fact reference distinct data. For example, consider the code:

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

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

`-OPT:alias=unnamed`

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

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

The compiler assumes that `f` cannot point to `g`, and produces an overlapped schedule for the two calculations.

This option also implies the `alias=typed` assumption. Note that this is the default assumption for the pointers implicit in Fortran dummy arguments according to the ANSI standard.

`-OPT:alias=restrict` and `-OPT:alias=disjoint`

The compiler assumes a very restrictive (`restrict`) model of aliasing: memory operations dereferencing different named pointers in the program are assumed not to alias with each other, nor with any named scalar in the program. For example, if `p` and `q` are pointers, `*p` does not alias with `*q`; `*p` does not alias with `p`; and `*p` does not alias with any named scalar variable.

Use `-OPT:alias=no_restrict` when distinct pointer variables may point to overlapping storage.

Use `-OPT:alias=disjoint` for memory operations dereferencing different named pointers in the program that are assumed not to alias with each other, or with any named scalar in the program. For example, if `p` and `q` are pointers, `*p` does not alias with `*q`; `*p` does not alias with `**p`; and `*p` does not alias with `**q`.

Use `-OPT:alias=no_disjoint` when distinct pointer expressions may point to overlapping storage.

Although these options are very dangerous to use, they may produce significantly better code when used for specific, well-controlled cases where they are known to be valid.

Simplifying Code with the **-OPT** Option

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

-OPT:fold_unsafe_relops

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

-OPT:fold_unsigned_relops

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

Controlling Execution Frequency

The #pragma mips_frequency_hint provides information about execution frequency for certain regions in the code. The format of the pragma is:

#pragma mips_frequency_hint {NEVER|INIT} [*function_name*]

You can provide the following indications: NEVER or INIT.

NEVER: This region of code is never or rarely executed. The compiler may move this region of the code away from the normal path. This movement may either be to the end of the procedure or at some point to an entirely separate section.

INIT: This region of code is executed only during initialization or startup of the program. The compiler may try to put all regions under INIT together to provide better locality during the startup of a program.

You can specify this pragma in two ways:

- You can specify it with a function declaration. It then applies everywhere the function is called. For example:

```
extern void Error_Routine ();
#pragma mips_frequency_hint NEVER Error_Routine
```

Note: The pragma must appear after the function declaration.

- You can place the pragma anywhere in the body of a procedure. It then applies to the next statement that follows the pragma. For example:

```
        if (some_condition) {
#pragma mips_frequency_hint NEVER
            Error_Routine ();
        }
```

The Code Generator

This section describes the part of the compiler that generates code.

The code generator processes an input program unit (PU) in intermediate representation form to produce an output object file (.o) or assembly file (.s).

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

This section covers the following topics:

- "Code Generator and Optimization Levels ", page 106
- "Code Generator and Optimization Levels -O2 and -O3", page 107
- "Modifying Code Generator Defaults", page 115
- "Other Code Generator Performance Topics", page 116

Code Generator and Optimization Levels

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

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

An Example of Local Optimization for Fortran

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

The .s file for -O0:

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

The .s file for -O1:

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

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

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

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

Code Generator and Optimization Levels -O2 and -O3

This section provides additional information about the -O2 and -O3 optimization
levels.

if Conversion

The if conversion transformation converts control-flow into conditional assignments.
For example, consider the following code before if conversion. Note that expr1 and
expr2 are arbitrary expressions without calls or possible side effects. For example, if
expr1 is i++, the following example would be wrong because 'i' would not be
updated.

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

After if conversion, the code looks like this:

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

Performing `if` conversion results in the following benefits:

- It exposes more instruction-level parallelism. This is almost always valuable on hardware platforms such as R10000.

- It eliminates branches. Some platforms (for example, the R10000) have a penalty for taken branches. There can be substantial costs associated with branches that are not correctly predicted by branch prediction hardware. For example, a mispredicted branch on R10000 has an average cost of about 8 cycles.

- It enables other compiler optimizations. Currently, cross-iteration optimizations and software pipelining both require single basic block loops. The `if` conversion changes multiple basic block innermost loops into single basic block innermost loops.

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

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

Notice that with `-TENV:X=2`, it is legal to speculate FLOPs, but not legal to speculate memory references. So `expr1 = *p + y;` can be speculated to `tmp1 = *p_safe + y;`. If `*known_safe_address` is uninitialized, there can be spurious floating point exceptions associated with this code. In particular, on some MIPS platforms (for example, R10000) if the input to a FLOP is a denormalized number, then a trap will occur. Therefore, by default, the code generator initializes `*known_safe_address` to 1.0.

Cross-Iteration Optimizations

Four main types of cross-iteration optimizations include:

- Read-Read Elimination: Consider the following example:

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

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

- Read-Write Elimination: Sometimes a value written in one iteration is read in a later iteration. For example:

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

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

- Write-Write Elimination: Consider the following example:

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

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

- Common Sub-Expression Elimination: Consider the following example:

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

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

Loop Unrolling

In this example, unrolling 4 times converts this code:

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

to this code:

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

Loop unrolling:

- Exposes more instruction-level parallelism. This may be valuable even on execution platforms such as R10000 or R12000 systems.

- Eliminates branches.

- Amortizes loop overhead. For example, unrolling replaces four increments i+=1 with one increment i+=4.

- Enables some cross-iteration optimizations such as read/write elimination over the unrolled iterations.

Recurrence Breaking

Recurrence breaking offers multiple benefits. Before the recurrences are broken (see both of the following examples), the compiler waits for the prior iteration's add to complete (four cycles on the R8000) before starting the next one, so four cycles per iteration occur.

When the compiler interleaves the reduction, each add must still wait for the prior iteration's add to complete, but four of these are done at one time, then partial sums

are combined on exiting the loop. The four iterations are done in four cycles, or one cycle per iteration, quadruple the speed.

With back substitution, each iteration depends on the result from two iterations back (not the prior iteration), so four cycles per two iterations occur, or two cycles per iteration (double the speed).

Note: The compiler actually interleaves and back-substitutes these examples even more than shown below, for even greater benefit (3 cycles/4 iterations for the R8000 in both cases). These examples are simple for purposes of exposition.

Two types of recurrence breaking are reduction interleaving and back substitution:

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

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

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

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

- Back substitution. For example:

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

  The code is converted to:

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

Software Pipelining

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

Global Code Motion

The global code motion phase performs various code motion transformations in order to reduce the overall execution time of a program. The global code motion phase is useful because it does the following:

- It moves instructions in nonloop code. In the following code example, assume `expr1` and `expr2` are arbitrary expressions that cannot be `if`-converted. The `cond` is a Boolean expression that evaluates to either true or false and has no side effects with either `expr1` or `expr2`.

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

  After global code motion, the code looks like this:

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

  Note, that `expr1` is arbitrarily chosen to speculate above the branch. The decision to select candidates for code movement are based on several factors, including resource availability, critical length, basic block characteristics, and so forth.

- It moves instructions in loops with control-flow. In the following code example, assume that `p` is a pointer and `expr1` and `expr2` are arbitrary expressions

involving `p`. Also, assume that `cond` is a Boolean expression that uses `p` and has no side effects with `expr2` and `expr1`.

```
while (p != NULL) {
    if (cond)
        sum += expr1(p);
    else
        sum += expr2(p);
    p = p->next;
}
```

After global code motion, the code looks like this:

```
while (p != NULL) {
    t1 = expr1(p);
    t2 = expr2(p);
    if (cond)
            sum += t1;
    else
        sum += t2;
    p = p->next;
}
```

Note, that `t1` and `t2` temporaries are created to evaluate the respective expressions and conditionally executed. The increment of the pointer, `p=p->next`, cannot move above the branch because of side effects with the condition.

- It moves instructions across procedure calls. In the following code example, assume that `expr1` has no side effects with the procedure call to `foo` (that is, procedure `foo` does not use and/or modify the expression `expr1`).

```
...
foo();
expr1;
...
```

After global code motion, the code looks like this:

```
...
expr1;
foo();
...
```

Benefits of Global Code Motion

The benefits of global code motion include the following:

- It exposes more instruction-level parallelism. Global code motion identifies regions and/or blocks that have excessive and/or insufficient parallelism than that provided by the target architecture. Global code motion effectively redistributes (or load balances) the regions/blocks by selectively performing code movements between them. This can effectively reduce their respective schedule lengths and the overall execution time of the program.

- It provides branch delay slot filling. Global code motion fills branch delay slots and converts most frequently executed branches to branch-likely form (for example, `beql, rs, rt, L1`).

- It enables other compiler optimizations. As a result of performing global code motion, some branches are either removed or transformed to a more effective form.

Steps Performed by the Code Generator at Levels `–O2` and `–O3`

The steps performed by the code generator at `–O2` and `–O3` include:

1. Nonloop `if` conversion. This also works in loops by performing any `if`-conversion that produces faster code.

2. Find innermost loop candidates for further optimization. Loops are rejected for any of the following reasons:

   - Marked `UNIMPORTANT` (for example, LNO cleanup loop)

   - Strange control flow (for example, branch into the middle)

3. `if` convert (`–O3` only). This transforms a multi-basic block loop into a single basic block containing operations with "guards." The `if` conversion of loop bodies containing branches can fail for any of the following reasons:

   - Cross-iteration read/write (read/read, and write/write) elimination

   - Cross-iteration CSE (common subexpression elimination)

   - Recurrence fixing

   - Software pipelining

4. Perform cross-iteration optimizations (except write/write elimination on loops without trip counts; for example, most "while" loops).

5. Unroll loops.

6. Fix recurrences.

7. If still a loop, and there is a trip count, and –O3, invoke software pipelining.

8. If not software pipelined, reverse if convert.

9. Reorder basic blocks to minimize (dynamically) the number of taken branches. Also eliminate branches to branches when possible, and remove unreachable basic blocks. This step also happens at –O1.

10. Invoke global code motion phase.

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

- At –O3, the loop nest optimizer, LNO, provides a dependence graph for each innermost loop.

- The scalar optimizer provides information on aliasing at the level of symbols. That is, it can tell whether arrays A and B are independent, but it does not have information about the relationship of different references to a single array.

- The code generator can sometimes tell that two memory references are identical or distinct. For example, if two references use the same register, and there are no definitions of that register between the two references, then the two references are identical.

Modifying Code Generator Defaults

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

You can control the code generator by:

- Increasing or decreasing the unroll amount.

  A heuristic is controlled by -OPT:unroll_analysis (on by default), which generally tries to minimize unrolling. Less unrolling leads to smaller code size and

faster compilation. You can change the upper bound for the amount of unrolling with -OPT:unroll_times (default is 8) or -OPT:unroll_size (the number of instructions in the unrolled body, current default is 80).

You can look at the .s file for notes (starting with #<loop>) that indicate how the decision to limit unrolling was made. For example, loops are not unrolled with recurrences that cannot be broken (since unrolling cannot possibly help in these cases), so the .s file now tells why unrolling was limited and how to change it. For example:

```
#<loop> Loop body line 7, nesting depth:1, estimated iterations: 100
#<loop> Not unrolled: limited by recurrence of 4 cycles
#<loop> Not unrolled: disable analysis w/-CG:unroll_analysis=off
```

- Disabling software pipelining with -OPT:swp=off.

  As far as the code generator is concerned, -O3 --OPT:swp=off is the same as -O2. Since LNO does not run at -O2, however, the input to the code generator can be very different, and the available aliasing information can be very different. In particular, cross-iteration loop optimizations are much more effective at -O3 even with -OPT:swp=off, due to the improved alias information.

Other Code Generator Performance Topics

This section explains a few miscellaneous topics including:

- Prefetch and Load Latency
- Frequency and Feedback

Prefetch and Load Latency

At the -O3 level of optimization, with -r10000, LNO generates prefetches for memory references that are likely to miss either the L1 (primary) or the L2 (secondary) cache. The code generator generates prefetch operations for L2 prefetches, and implements L1 prefetches as follows: makes sure that loads that had associated L1 prefetches are issued at least 8 cycles before their results are used.

It is often possible to reduce prefetch overhead by eliminating some of the corresponding prefetches from different replications. For example, suppose a prefetch is only required on every fourth iteration of a loop, because four consecutive iterations will load from the same cache line. If the loop is replicated four times by

software pipelining, then there is no need for a prefetch in each replication, so three of the four corresponding prefetches are pruned away.

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

Frequency and Feedback

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

The frequency guesses are replaced with the measured frequencies. Currently the information guides if-conversion, some loop unrolling decisions (unrelated to the trip count estimate), global code motion, control flow optimizations, global spill and restore placement, global register allocation, instruction alignment, and delay slot filling. Average loop trip-counts can be derived from feedback information. Trip count estimates are used to guide decisions about how much to unroll and whether or not to software pipeline.

Reordering Code Regions

Cording is an optimization technique for reordering parts of your program to achieve better locality of reference and reduce instruction fetch overhead based on dynamically collected data. The following areas are influenced by code region reordering:

- Page faults and translation lookaside buffer (TLB) misses
- Instruction cache misses

Both of these events contribute to instruction fetch overhead, which can be alleviated with better locality of reference. Retrieving an instruction from cache is always faster than retrieving it from memory; so the idea is to keep instructions in cache as much as possible. The frequencies and costs associated with each of those events differ

significantly. The size of the program in memory and text-resident set sizes can also be reduced as a result of cording.

Programs can be reordered using either the cord(1) command (see the following section) or the ld(1) linker command (see "Reordering with ld", page 119). The SpeedShop prof(1) command and the WorkShop cvperf(1) user interface are alternative methods of provided feedback files to cord and ld (see "Using prof or cvperf", page 119).

Reordering with `cord`

Use the following procedure to optimize your application by reordering its text areas with the cord(1) command:

1. Run one or more SpeedShop bbcounts experiments to collect performance data, setting caliper points to better identify phases of execution.

   ```
   % ssrun -bbcounts a.out
   ```

2. Use sswsextr to extract working set files related to the interval between each pair of calipers in the experiment file.

   ```
   % sswsextr a.out a.out.bbcounts.m20683
   ```

   A working set list file (in this case, a.out.a.out.bbcounts.m20683.wslist) is also generated. It assigns a number to the working set files and the weight for each one (the default weight is 1).

3. Use ssorder(1) or sscord(1) to generate a cord feedback file combining the working set files for the binary.

```
% ssorder -wsl a.out.a.out.bbcounts.m20683.wslist -gray -o a.out.fb a.out
```

```
% sscord -wsl a.out.a.out.bbcounts.m20683.wslist a.out
```

4. Use the cord command to reorder the procedures in the binary.

   ```
   % cord a.out a.out.fb
   ```

Reordering with `ld`

The following procedure uses the `ld` linker to reorder routines in a source file. The procedure shows how to reorder routines in an executable file (`a.out`), but you can also reorder a DSO.

1. Compile the application as follows:

   ```
   % f90 -OPT:procedure_reorder verge.f
   ```

   You can turn reordering on or off for a given compilation by setting the `procedure_reorder` argument to an optional Boolean value. Doing so is convenient if you are compiling the application in a makefile. Setting a Boolean value of 1 enables reordering, while a value of 0 disables reordering.

   ```
   % f90 -OPT:procedure_reorder=1 verge.f
   ```

2. Run a SpeedShop `bbcounts` experiment to collect performance data, setting caliper points to better identify phases of execution.

   ```
   % ssrun -bbcounts a.out
   ```

3. Use `sswsextr` to extract working set files for the binary.

   ```
   % sswsextr a.out a.out.bbcounts.m20683
   ```

4. Use `ssorder` or `sscord` to generate a cord feedback file, combining multiple working set files for the binary.

```
% ssorder -wsl a.out.a.out.bbcounts.m20683.wslist -gray -o a.out.fb a.out
```

```
% sscord -wsl a.out.a.out.bbcounts.m20683.wslist a.out
```

5. Use `ld` to reorder the procedures in the binary as follows:

   ```
   % ld -LD_LAYOUT:reorder_file=a.out.fb
   ```

Using `prof` or `cvperf`

Once the experiment file is generated, you can generate a cord feedback file using either the SpeedShop `prof` command or the WorkShop `cvperf` user interface.

Enter the `prof` command as follows:

```
% prof -cordfb a.out.bbcounts.m20683
```

The `-cordfb` option generates cord feedback for the executable and all DSOs. Along with its usual output, this command writes a cord feedback file named `a.out.fb`.

While using `prof` will give you what you want, using either the `ssorder` or `sscord` method described in the previous subsections or the `cvperf` method described in the following paragraphs produces more efficient results.

If you are using the WorkShop performance analyzer, first enter the `cvperf` command with the experiment file as an argument:

```
% cvperf a.out.bbcounts.m20683
```

Select the **Working Set View** from the **Views** menu. Once the new window appears, choose **Save Cord Map File** from the **Admin** menu. By default, the name of the cord feedback file will be `a.out.fb`.

Specify the cord feedback file on the `cord` or `ld` commands to reorder the procedures in the binary:

```
% cord a.out a.out.fb
% ld -LD_LAYOUT:reorder_file=a.out.fb
```

Programming Hints for Improving Optimization

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

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

- Hints for Writing Programs

- Coding Hints for Improving Other Optimization

- Using SpeedShop to optimize your code.

Hints for Writing Programs

Use the following hints when writing your programs:

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

- Use functions that return values instead of pointer parameters.

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

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

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

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

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

  Pointers in particular tend to cause aliasing problems, since it is often impossible for the compiler to identify their target objects. Therefore, you can help the compiler avoid possible aliases by introducing local variables to store the values obtained from dereferenced pointers. Indirect operations and calls affect dereferenced values, but do not affect local variables. Therefore, local variables can be kept in registers. The following example shows how the proper placement of pointers and the elimination of aliasing produces better code.

In the following example, the optimizer does not know if `*p++ = 0` will eventually modify `len`. Therefore, the compiler cannot place `len` in a register for optimal performance. Instead, the compiler must load it from memory on each pass through the loop.

```
int len = 10;
void
zero(char *p)
{
    int i;
    for (i= 0; i!= len; i++) *p++ = 0;
}
```

Increase the efficiency of this example by not using global or common variables to store unchanging values.

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

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

  ```
  extern int ext1;
  static int stat1;

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

- Write straightforward code. For example, do not use ++ and - - operators within an expression. Using these operators produces side-effects (requires the use of extra temporaries, which increases register pressure).

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

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

Coding Hints for Improving Other Optimization

The global optimizer processes programs only when you specify the –O2 or –O3 option at compilation. The code generator phase of the compiler performs certain optimizations. This section has coding hints that increase optimization for other passes of the compiler.

Use Tables Rather Than `if-then-else` or `switch` Statements

In your programs, use tables rather than if-then-else or switch statements. For example, consider this code:

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

Instead of:

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

Use:

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

Declare Variables Most Frequently Manipulated

As an optimizing technique, the compiler puts the first eight parameters of a parameter list into registers where they may remain during execution of the called routine. Therefore, always declare, as the first eight parameters, those variables that are most frequently manipulated in the called routine.

Use 32-Bit or 64-Bit Scalar Variables

Use 32-bit or 64-bit scalar variables instead of smaller ones. This practice can take more data space. However, it produces more efficient code because the MIPS instruction set is optimized for 32-bit and 64-bit data.

Suggestions for C and C++ Programs

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

- Rely on `libc.so` functions (for example, `strcpy`, `strlen`, `strcmp`, `bcopy`, `bzero`, `memset`, and `memcpy`). These functions are carefully coded for efficiency.

- Use a signed data type, or cast to a signed data type, for any variable that does not require the full unsigned range and must be converted to floating-point. For example:

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

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

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

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

- Use `const` and `restrict` qualifiers. The `__restrict` keyword tells the compiler to assume that dereferencing the qualified pointer is the only way the program can access the memory pointed to by that pointer. Hence loads and stores through such a pointer are assumed not to alias with any other loads and

stores in the program, except other loads and stores through the same pointer variable. For example:

```
float x[ARRAY_SIZE];
float *c = x;

void f4_opt(int n, float * __restrict a, float * __restrict b)
{
    int i;
    /* No data dependence across iterations because of __restrict */
    for (i = 0; i < n; i++)
      a[i] = b[i] + c[i];
}
```

Suggestions for C++ Programs Only

The following suggestions apply to C++ programs:

- Use the `inline` keyword whenever possible. Functions calls in loops that are not inlined prevent loop-nest optimizations and software pipelining.

- Use a direct calls rather than indiscriminate use of virtual function calls. The penalty is in method lookup and the inability to inline them.

- For scalars only, avoid the creation of unnecessary temporaries, that is, `Aa = 1` is better than `Aa = A(1)`.

- For `structs` and `class`, pass by `const ref` to avoid the overhead of copying.

- If your code does not use exception handing, use `-LANG:exceptions=off` when compiling.

Using SpeedShop

SpeedShop is an integrated package of performance tools that you can use to gather performance data and generate reports. To record the experiments, use the `ssrun(1)` command, which runs the experiment and captures data from an executable (or instrumented version). You can compare experiment results with the `sscompare(1)` command. To examine data you can use either `prof(1)` or display the data in the WorkShop graphical user interface with the `cvperf(1)` command. Speedshop also lets you start a process and attach a debugger to it.

For detailed information about SpeedShop, `ssrun`, `sscompare` and `prof`, see the *SpeedShop User's Guide*.

Coding for 64-Bit Programs

This chapter provides information about ways to write your code so that you can take advantage of the SGI implementation of the IRIX 64-bit operating system. Specifically, this chapter describes the following:

- "Coding Assumptions to Avoid", page 127

- "Solving Porting Problems", page 130

- "Guidelines for Writing Code for 64-Bit SGI Platforms", page 130

Also, refer to Chapter 6, "Porting Code to N32 and 64-Bit SGI Systems", page 133, for information about compatibility, porting guidelines, and details on data types, predefined types, typedefs, memory allocation, and so forth. The *MIPSpro N32 ABI Handbook* and the *MIPSpro 64-Bit Porting and Transition Guide* provide further information.

Coding Assumptions to Avoid

Most porting problems come from assumptions, implicit or explicit, about either absolute or relative sizes of the int, long int, or pointer types in code.

To avoid porting problems, examine code that assumes any of the following:

- sizeof(int) == sizeof(void *)

- sizeof(int) == sizeof(long)

- sizeof(long) == 4

- sizeof(void *) == 4

- Implicitly declared functions

- Constants with the high-order bit set

- Arithmetic with long types (including shifts involving mixed types and code that may overflow 32 bits)

Note: When compiling using –64-bit mode, avoid using unsigned 32-bit integers. In the MIPS architecture, when a 32-bit integer (signed or unsigned) is stored in 64-bit registers, the high order 32 bits are sign-extended.

sizeof(int) == sizeof(void *)

An assumption may arise from casting pointers to int types to do arithmetic, from unions that implicitly identify int and pointer, or from passing pointer types as actual arguments to functions where the corresponding formal arguments are declared as int. Any of these practices may result in inadvertently truncating the high-order part of an address.

The compilers generally detect the first case and provide warnings. Also given ANSI C function prototypes, the compilers generally detect the last case. No diagnostic messages are provided for unions that implicitly identify ints and pointers.

You can declare an integer variable that is required to be the size of a pointer with the type ptrdiff_t in the standard header file stddef.h, or with the types _psint_t and _psunsigned_t in the header file inttypes.h.

Also note that a cast of an int to a pointer may result in sign-extension, if the sign bit of the int is set when a --64 compilation occurs.

sizeof(int) == sizeof(long)

Data that fits in an int or long on 32-bit systems will fit in an int on 64-bit systems. Expansion, in this case, has no visible effect. Problems may occur, however, where an unsigned int actual parameter is passed to a long (signed or unsigned) formal parameter without benefit of an ANSI prototype. In this case, the unsigned value is implicitly sign-extended in the register, and therefore is misinterpreted in the callee if the sign bit was set.

sizeof(long) == 4

A problem may occur in cases where long int is used to map fields in data structures defined externally to be 32 bits, or where union attempts to identify a long with four char.

sizeof(void *) == 4

Problems with this code are similar to those encountered with `sizeof(long)==4`. However, mappings to external data structures are seldom a problem, since the external definition also assumes 64-bit pointers.

Implicitly Declared Functions

It is always risky to call a function without an explicit declaration in scope. Furthermore, be sure to declare with a compatible prototype any function defined with a prototype. Problems arise when mixing prototype and nonprototype declarations for the same function. For example, suppose you call a function (defined with a prototype to take a variable number of arguments) in a scope without a prototyped declaration. You may get unexpected results if a floating point argument is passed to it. This is a typical problem with calls to `printf` and after `stdio.h` routines. Therefore, always include `stdio.h` in any context where you use `stdio.h` facilities.

Constants with the High-Order Bit Set

A change in type sizes may yield some problems related to constants. Be 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 modes, the constant is interpreted as a 32-bit unsigned `int`, with value 4,294,967,295. In 32-bit mode, the addition results in a 32-bit unsigned `long`, which is cast to type `long` and has value $x-1$ because of the truncation to 32 bits. In 64-bit mode, the addition results in a 64-bit `long` with value $x+4,294,967,295$, and the cast is redundant.

Arithmetic with `long` Types

Code that does arithmetic (including shifting), and code that may overflow 32 bits and assumes particular treatment of the overflow (for example, truncation), can exhibit different behavior, depending on the mix of types involved (including how it is signed).

Similarly, implicit casting in expressions that mix int and long values may produce unexpected results due to sign/zero extension. An int constant is sign- or zero-extended when it occurs in an expression with long values.

Solving Porting Problems

Once you identify porting problems, solve them by:

- Changing the relevant declaration to one that has the desired characteristics in both target environments

- Adding explicit type casts to force the correct conversions

- Using function prototypes or using type suffixes (such as l or u) on constants to force the correct type

Guidelines for Writing Code for 64-Bit SGI Platforms

The key to revising existing code and writing new code that is compatible with all of the major C data models is to avoid the assumptions described previously in "Coding Assumptions to Avoid", page 127. Since all of the assumptions described sometimes represent legitimate attributes of data objects, you need to tailor declarations to the target machines' data models.

The following guidelines help you to produce portable code. Use these guidelines when you are developing new code or as you identify portability problems in existing code.

1. In a header file that you include in each of the program's source files, define (typedef) a type for each scalar integer type:

   - 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 that 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. The typedefs referred to above exist in the file inttypes.h (see "Using Typedefs ", page 140).

After you construct the above header file, use the new typedef types instead of the standard C type names. You may need a distinct copy of this header file (or conditional code) for each target platform supported.

If you provide libraries or interfaces to be used by others, be careful to use these types (or similar application-specific types) chosen to match the specific requirements of the interface. Also, carefully choose the actual names used to avoid name space conflicts with other libraries. Thus, your clients should be able to use a single set of header files on all targets. However, you will always need to provide distinct libraries (binaries) for the 32-bit compatibility mode and the 64-bit native mode on 64-bit SGI platforms, although the sources can be identical.

2. Be sure to specify constants with an appropriate type specifier so that they will have the size required by the context with the values expected. 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 may be interpreted as 4,294,967,295 (signed or unsigned, depending on the mode and context) on most 64-bit systems. The inttypes.h header file provides cpp macros to facilitate this conversion. Defining constants that are sensitive to type sizes in a central header file may help in modifying them when a new port is done.

3. Where printf/scanf are used for objects whose types are typedefed 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 ''%lld''
#define _fmt64u ''%llu''
```

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

4. Code that has a variable number of floating point arguments or doubles should be prototyped. `printf` is used to print a variable floating point in this example:

```
#include <stdio.h>

main()

{

     float d,e;
     d = 3.14;
     printf(''%e\n'',d);
}
```

Porting Code to N32 and 64-Bit SGI Systems

This chapter explains the levels of compatibility between the new 32-bit compilation mode (n32), the old 32-bit mode, and 64-bit programs. It also describes the porting procedure to follow and the changes you must make to port your application from old 32-bit mode to n32-bit mode. For more details, see the *MIPSpro 64-Bit Porting and Transition Guide*

Specifically, this chapter discusses the following topics:

- "Compatibility", page 133, describes compatibility between 32, n32, and 64-bit programs.

- "N32 Porting Guidelines", page 135, explains guidelines for porting high-level languages.

- "Porting Code to 64-Bit SGI Systems", page 137, describes data types, typedefs, maximum memory allocation, and use of large files on XFS.

This chapter uses the following terminology:

o32   The old 32-bit ABI generated by the ucode compiler; that is, 32-bit compilers prior to IRIX 6.1 operating system. For information about this compiler, see the *MIPS O32 Compiling and Performance Tuning Guide*

n32   The new 32-bit ABI generated by the MIPSPro 64-bit compiler (for a list of n32 features, see Chapter 1, "About the MIPSpro Compiler System", page 1). For information about the n32 ABI, see the *MIPSpro N32 ABI Handbook.*

Compatibility

In order to execute different ABIs, support must exist at three levels:

- The operating system must support the ABI.

- The libraries must support the ABI.

- The application must be recompiled with a compiler that supports the ABI.

Figure 6-1, page 134, shows how applications rely on library support to use the operating system resources that they need.

Note: Each o32, n32, and n64 application must be linked against unique libraries that conform to its respective ABI. As a result, you **cannot** mix and match object files or libraries from any of the different ABIs.
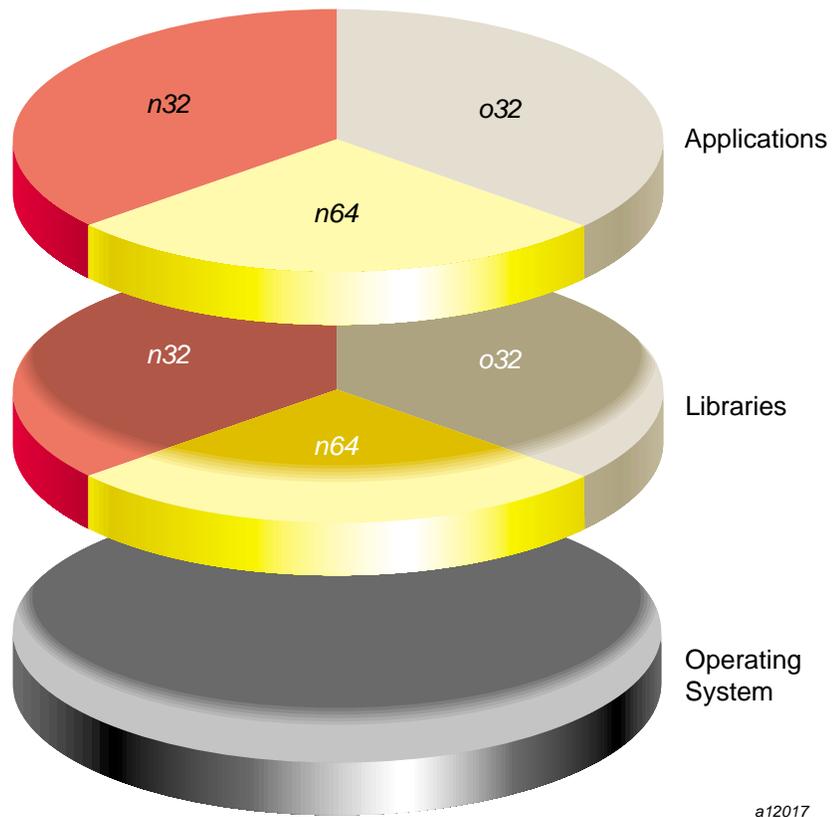
Figure 6-1 Application Support under Different ABIs

Figure 6-2, page 135, illustrates the library locations for different ABIs.
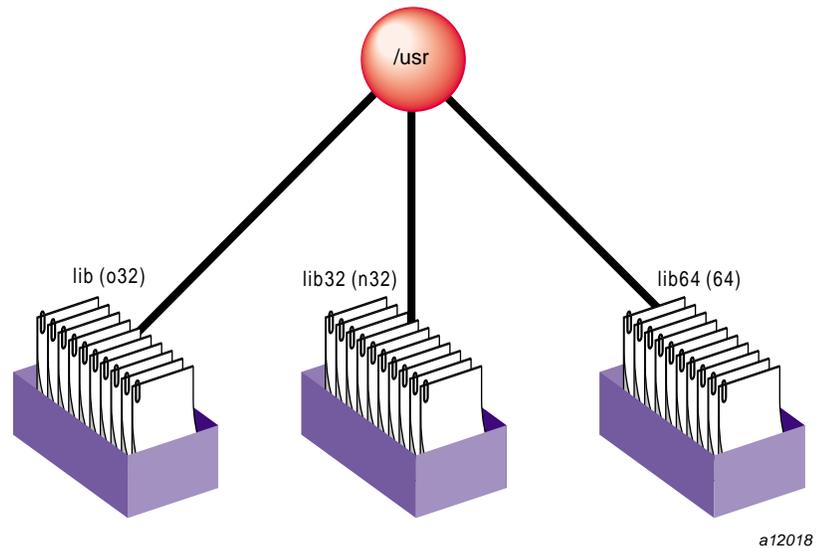
Figure 6-2 Library Locations for Different ABIs

An operating system that supports all three ABIs is also needed for running the application. Consequently, all applications that want to use the features of n32 must be ported. The next section covers the steps in porting an application to the N32 ABI.

N32 Porting Guidelines

This section describes the guidelines/steps necessary to port IRIX 5.*x* 32-bit applications to n32. Typically, any porting project can be divided into the following tasks:

- Identifying and creating the necessary porting environment (see "Porting Environment", page 136)

- Identifying and making the necessary source code changes (see "Source Code Changes", page 136)

- Rebuilding the application for the target machine (see "Build Procedure", page 136)

- Analyzing and debugging run-time issues (see "Run-time Issues", page 137)

Each of these tasks is described in the following sections. You can also find additional information about n32 in the *MIPSpro N32 ABI Handbook*.

Porting Environment

The porting environment consists of a compiler and associated tools, `include` files, libraries, and makefiles, all of which are necessary to compile and build your application. To generate n32 code, you must:

* Check all libraries needed by your application to make sure they are recompiled using n32. The default root location for n32 libraries is */usr*/`lib32`. If the n32 library needed by your application does not exist, recompile the library for n32.

* Modify existing Makefiles (or set environment variables) to reflect the locations of these n32 libraries.

Source Code Changes

Since no differences exist in the sizes of fundamental types between the old 32-bit mode and n32, porting to n32 requires no source code changes for applications written in high-level languages such as C, C++, and Fortran. The only exception to this is that C functions that accept variable numbers of floating point arguments must be prototyped.

Assembly language code, however, must be modified to reflect the new subprogram interface. Guidelines for following this interface are described in Chapter 3 of the *MIPSpro N32 ABI Handbook* in the section titled "Assembly Language Programming Guidelines."

Build Procedure

Recompiling for n32 involves either setting the `-n32` argument in the compiler invocation or running the compiler with the environment variable `SGI_ABI` set to `-n32`. That's all you must do after you set up a native n32 compilation environment (that is, all necessary libraries and `include` files reside on the host system).

Run-time Issues

Applications that are ported to n32 may get different results than their o32 counterparts. Reasons for this include:

- Differences in algorithms used by n32 libraries and o32 libraries.

- Operand reassociation or reduction performed by the optimizer for n32.

- Hardware differences of the R8000 and R1000 (`madd` instructions round slightly differently than a multiply instruction followed by an add instruction).

Porting Code to 64-Bit SGI Systems

This section covers porting code to 64-bit SGI systems, including:

- Using Data Types

- Using Predefined Types

- Using Typedefs

- Maximum Memory Allocation

- Using Large Files with XFS

You can find additional information about porting to 64-bit SGI systems in the *MIPSpro 64-Bit Porting and Transition Guide*.

Using Data Types

Data types and sizes are listed in Table 6-1.

Table 6-1 Data Types and Sizes

Data Type	(old) 32 Bit	n32 Bit	64 Bit
char	8	8	8
short	16	16	16
int	32	32	32

Data Type	(old) 32 Bit	n32 Bit	64 Bit
`long`	32	32	64
`long long`	64 (emulated with 32-bit integer operations)	64 (native 64-bit integer operations)	64
`pointer`	32	32	64
`float`	32	32	32
`double`	64	64	64
`long double`	64	128	128
`void`	32	32	64

Note that in 64-bit mode, types `long` and `int` have different sizes and ranges; a `long` always has the same size as a `pointer`. A `pointer` (or address) has 64-bit representation in 64-bit mode and 32-bit representation in 32-bit mode. An `int` has a smaller range than a `pointer` in 64-bit mode. On 32-bit compiles, the `long double` generates a warning message indicating that the `long` qualifier is not supported.

Characteristics of integral types and floating point types are defined in the standard files `limits.h` and `float.h`.

Using Predefined Types

The `cc`, `CC`, and `as` compiler drivers produce predefined macros listed in Table 6-2. These macros are used in `sys/asm.h`, `sys/regdef.h`, and `sys/fpregdef.h`.

Table 6-2 Predefined Macros

32-Bit Executables	64-Bit Executables
`-D_MIPS_FPSET=16`	`-D_MIPS_FPSET=32`
`-D_MIPS_ISA=_MIPS_ISA_MIPS1`	`-D_MIPS_ISA=_MIPS_ISA_MIPS3`
`-D_MIPS_SIM=_MIPS_SIM_ABI32`	`-D_MIPS_SIM=_MIPS_SIM_ABI64`
`-D_MIPS_SZINT=32`	`-D_MIPS_SZINT=32`

32-Bit Executables	64-Bit Executables
`-D_MIPS_SZLONG=32`	`-D_MIPS_SZLONG=64`
`-D_MIPS_SZPTR=32`	`-D_MIPS_SZPTR=64`

`_MIPS_FPSET` describes the number of floating point registers. The 64-bit compilation mode makes use of the extended floating point registers.

`MIPS_ISA` determines the MIPS Instruction Set Architecture. `MIPS_ISA_MIPS1` and `MIPS_ISA_MIPS3` are the defaults for 32 bits and 64 bits, respectively. For example:

```
/* Define a parameter for the integer register size: */
#if (_MIPS_ISA == _MIPS_ISA_MIPS1 || _MIPS_ISA == _MIPS_ISA_MIPS2)
#define SZREG          4
#else
#define SZREG          8
#endif
```

`MIPS_SIM` determines the MIPS Subprogram Interface Model, which describes the subroutine linkage convention and register naming/usage convention.

`_MIPS_SZINT`, `_MIPS_SZLONG`, and `_MIPS_SZPTR` define the size of types `int`, `long`, and `pointer`, respectively.

The 64-bit MIPSpro compiler drivers generate 64-bit `pointer` and `long` and 32-bit `int`. Therefore, assembler code that uses either `pointer` or `long` types must be converted to use `double-word` instructions for MIPS III code (`-64`), and must continue to use `word` instructions for MIPS I and MIPS II code (`-32`).

Also, new subroutine linkage conventions and register naming conventions exist. The compiler predefined macro `_MIPS_SIM` enables macros in `sys/asm.h` and `sys/regdef.h`.

Eight argument registers exist: `$4` through `$11`. Four additional argument registers replace the `temp` registers in `sys/regdef.h`. These `temp` registers are not lost, however, as the argument registers can serve also as scratch registers, with certain constraints.

In the `_MIPS_SIM_ABI64` model, registers `t4` through `t7` are not available, so any code using these registers does not compile. Similarly, registers `a4` through `a7` are not available under the `_MIPS_SIM_ABI32` model.

If you are converting assembler code, the new registers ta0, ta1, ta2, and ta3 are available under both _MIPS_SIM models. These alias with registers t4 through t7 in 32-bit mode, and with registers a4 through a7 in 64-bit mode.

Note that the caller no longer has to reserve space for a called function in which to store its arguments. The called routine allocates space for storing its arguments on its own stack, if desired. The NARGSAVE macro in sys/asm.h facilitates this.

Using Typedefs

This section describes typedefs that you can use to write portable code for a range of target environments, including 32- and 64-bit workstations as well as 16- and 32-bit PCs. These typedefs are enabled by compiler-predefined macros (listed in Table 6-2, page 138), and are in the file inttypes.h. (This discussion applies to C, although the same macros are predefined by the C++ compiler.)

Portability problems exist because an int (32 bits) is no longer the same size as a pointer (64 bits) and a long (64 bits) in 64-bit programs. Typedefs free you from having to know the underlying compilation model or worry about type sizes. In the future, if that model changes, the code should still work.

Typically, you want source code that you can compile either in 32- or 64-bit mode. (In this discussion, 32-bit mode implies -mips1/2; 64-bit mode implies -mips3/4.)

The following typedefs are defined in inttypes.h:

```
typedef signed char    int8_t;
typedef unsigned char    uint8_t;
typedef signed short   int16_t;
typedef unsigned short   uint16_t;
typedef signed int   int32_t;
typedef unsigned int   uint32_t;
typedef signed long long int   int64_t;
typedef unsigned long long int   uint64_t;
typedef signed long long int   intmax_t;
typedef unsigned long long int   uintmax_t;
typedef signed long int   intptr_t;
typedef unsigned long int   uintptr_t;
```

The intmax_t and uintmax_t types are guaranteed to be the largest integer type supported by this implementation. Use them in code that must be able to deal with

any integer value. `intptr_t` and `uintptr_t` are guaranteed to be exactly the size of a `pointer`.

Maximum Memory Allocation

The total memory allocation for a program, and individual arrays, can exceed 2 gigabytes (2 Gbytes, or 2,048 Mbytes).

Previous implementations of Fortran, C, and C++ limited the total program size, as well as the size of any single array, to 2 GBytes. The current release allows the total memory in use by the program to exceed 2 gigabytes.

Arrays Larger Than 2 Gigabytes

The IRIX 6.2 (MIPSPro 7.1) compilers (and above) support arrays that are larger than 2 gigabytes for programs compiled under the -64 ABI. The arrays can be local, global, and dynamically created as the following example demonstrates. (Initializers are not provided for these arrays.) Large array support is limited to Fortran, C, and C++.

Example of Arrays Larger Than 2 Gigabytes

The following code shows an example of arrays larger than 2 gigabytes.

```
#include <stdlib.h>

int i[0x100000008];

void foo()
{
int k[0x100000008];
   k[0x100000007] = 9;
   printf(``%d \n'', k[0x100000007]);
}

main()
{
char *j;
j = malloc(0x100000008);
   i[0x100000007] = 7;
   j[0x100000007] = 8;
   printf(``%d \n'', i[0x100000007]);
```

```
        printf(''%d \n'', j[0x100000007]);
        foo();

}
```

You must run this program on a 64-bit operating system with IRIX version 6.2 (or higher). You can verify the system you have by typing **uname -a**. You must have enough swap space to support the working set size and you must have your shell limit *datasize*, *stacksize*, and *vmemory*use variables set to values large enough to support the sizes of the arrays (see sh(1) man page).

The following example compiles and runs the preceding code after setting the stack size to a correct value:

```
% uname -a
IRIX64 cydrome 6.2 03131016 IP19
$cc -64 -mips3 a2.c
$limit
cputime         unlimited
filesize        unlimited
datasize        unlimited
stacksize       65536 kbytes
coredumpsize    unlimited
memoryuse       754544 kbytes
descriptors     200
vmemoryuse      unlimited
$limit stacksize unlimited
$limit
cputime         unlimited
filesize        unlimited
datasize        unlimited
stacksize       unlimited
coredumpsize    unlimited
memoryuse       754544 kbytes
descriptors     200
vmemoryuse      unlimited
$a.out
7
8
9
```

Using Large Files with XFS

An application may create or encounter files greater than 2 gigabytes with the XFS file system. If a program is doing sequential I/O and does not maintain internal byte counters, files greater than 2 gigabytes will not encounter problems.

However, if an application uses internal byte counters, then modifications are required. Table 6-3, page 143, lists potential problems and modifications required to enable files greater than 2 gigbytes to run on XFS.

Table 6-3 Modifications for Applications on XFS

Application	Modification
Uses an internal byte counter while reading	Change to type `long long`
Uses certain system calls such as `lseek()` and `stat()` that use 32-bit `off_t`	Use `lseek64()`, `stat64()`, and so forth
Relies on internal features of EFS (such as reads the raw disk)	Rewrite the application (so it does not read the raw disk)

For more information about XFS, see *Getting Started with XFS Filesystems*.

Index

D

prefetching, 83
data type
  signed, 124
data types
  sizes, 137
debugging
  driver options, 31
  floating points, 101
defaults
  compilation modes, 5
  specification file, 6
definition, 117
dependence analysis, 93
directives
  LNO, 85
dis command, 26, 27
disassemble object file, 26
dlclose(), 55
dlerror(), 55
dlopen(), 53
dlsym(), 54
double, 138
drivers
  as assembler, 21
  bypassing, 2
  defaults, 6, 19
  –c option, 3
  fec preprocessor, 2
  file name suffixes, 9
  input file suffixes, 10
  linking, 3
  omit linking, 3
  optimizing programs, 96
  options, 19, 31
  -show option, 3
  stages of compilation, 3
DSO
  benefits, 33
DSOs, 1, 8, 33
  building new dsos, 43
  converting libraries, 52
  creating dsos, 43

dlclose(), 55
dlerror(), 55
dlopen(), 53
dlsym(), 54
dynamic loading diagnostics, 55
exporting symbols, 45
guidelines, 36
hiding symbols, 45
libraries, shared, 36
linking, 23
loading dynamically, 53
mmap() system call, 55
munmap() system call, 55
naming conventions, 43
QuickStart, 40
search path, 47
sgidladd(), 53
shared libraries, 36
starting quickly, 40
unloading dynamically, 55
versioning, 56
dump command. See elfdump, 27
DWARF symbolic information, 27
dwarfdump command, 26, 27
dynamic linking, 1, 8, 53
Dynamic shared objects. See DSOs, 23

E

Elf object file, 27
ELF. See executable and linking format, 8
elfdump command, 26, 27
  command syntax, 27
elimination
  branches, 108
  read-read, 109
  read-write, 109
  sub-expression, 109
  write-write, 109
–Bsymbolic, compiling, 50

N

n32, 136
  assembly language programs, 136
  build procedure, 136
  include files, 136
  libraries, 133, 136
  porting environment, 136
  porting guidelines, 135
  runtime issues, 137
  source code changes, 136
n32-bit mode, 5
naming source files, 9
NARGSAVE macro, 140
negative values
  problems, 131
nm command, 26, 27
  command syntax, 27
  example, 28
  example of undefined symbol, 26
  undefined symbol, 26

O

object file information
  disassemble, 26
  format, 1, 8
  listing file properties, 26
  listing section sizes, , 29
  symbol table information, 26, 27
  tools, 26
  using, 26
  using dwarfdump, 26
  using elfdump, 26, 27
operating system
  64 bit, 127
operations
  relational, 104
  unsigned relational, 104
optimization, 67
  addresses, 122

Also see optimizing programs, 75
C programs, 121
C++ programs, 121
Fortran, 121
function return values, 121
global, 120
if-then-else statements, 123
libc.so functions, 124
pointer placement, 121
signed data types, 124
STDARG, 123
stdarg.h, 123
switch statements, 123
tables, 123
tips for improving, 120
unions, 121
value parameters, 121
VARARG, 123
varargs.h, 123
variables, global vs. local, 121
optimizer, 2
optimizing programs
  alias analysis, 75
  Also see code generator, 105
  benefits, 68
  cache, 83
  code generator, 105
  common block padding, 73
    restrictions, 73
  debugging, 68
  dependence analysis, 93
  –INLINE, 71
  –LNO option, 75
    blocking, 79
    code transformation, 76
    controlling cache parameters, 85
    directives, 85
    fission, 86
    fusion, 86
    gather-scatter, 83
    loop fission, 81

type, determining for files, 27
typedefs, 131, 140
types
  assumptions, 127
  change in size, 129
  char, 137
  constants, 129
  double, 138
  float, 138
  int, 128, 140, 137
  largest integer type, 141
  long, 140, 138
  long double, 138
  long long, 138
  pointer, 128, 140, 138
  problems, 127
  scaling integer, 130
  short, 137
  sizes, 137

U

unions
  optimization, 121
unrolling loops, 79, 89, 110, 115
unsigned ints
  32-bit, 128
unsigned relational operations, 104

V

VARARG. See optimization, 123
varargs.h, 123
variables
  scalar, 124
virtual address space, 55

W

word-size scalar variables, 124
working set
  list file, 118
write-write elimination, 109

X

XFS
  file size, 143

Z

zero extension, 130
zero-extension code, 124