MIPSpro™ Compiling and Performance Tuning Guide

Document Number 007-2360-007

Contributors
Written by Arthur Evans, Wendy Ferguson, Jed Hartman, Jackie Neider
Production by Cindy Stief
Engineering contributions by Dave Anderson, Zaineb Asaf, Dave Babcock, Greg Boyd, Jack Carter, Ann Mei Chang, Wei-Chau Chang, David Ciemiewicz, Rune Dahl, Jim Dehnert, David Frederick, Sanjoy Ghosh, Jay Gischer, Bob Green, Seema Hiranandani, W. Wilson Ho, Marty Itzkowitz, Bhaskar Janakiraman, Woody Lichtenstein, Dror Maydan, Ajit Mayya, Ray Milkey, Michael Murphy, Bron Nelson, Andy Palay, Ron Price, John Wilkinson
St Peter's Basilica image courtesy of ENEL SpA and InfoByte SpA. Disk Thrower image courtesy of Xavier Berenguer, Animatica.

© Copyright 1997 Silicon Graphics, Inc.— All Rights Reserved
This document contains proprietary and confidential information of Silicon Graphics, Inc. The contents of this document may not be disclosed to third parties, copied, or duplicated in any form, in whole or in part, without the prior written permission of Silicon Graphics, Inc.

Restricted Rights Legend
Use, duplication, or disclosure of the technical data contained in this document by the Government is subject to restrictions as set forth in subdivision (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARS 52.227-7013 and/or in similar or successor clauses in the FAR, or in the DOD or NASA FAR Supplement. Unpublished rights reserved under the Copyright Laws of the United States. Contractor/manufacturer is Silicon Graphics, Inc., 2011 N. Shoreline Blvd., Mountain View, CA 94039-7311.

Silicon Graphics, the Silicon Graphics logo, and IRIS are registered trademarks and IRIX, CASEVision, IRIS IM, IRIS Showcase, Impressario, Indigo Magic, Inventor, IRIS-4D, POWER Series, RealityEngine, CHALLENGE, Onyx, Origin2000, and WorkShop are trademarks of Silicon Graphics, Inc. MIPS, R4000, and R8000 are registered trademarks and MIPSpro, R5000, and R10000 are trademarks of MIPS Technologies, Inc. OSF/Motif is a trademark of Open Software Foundation, Inc. PostScript is a registered trademark and Display PostScript is a trademark of Adobe Systems, Inc. UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company, Ltd. X Window System is a trademark of the Massachusetts Institute of Technology.

MIPSpro™ Compiling and Performance Tuning Guide
Document Number 007-2360-007

Contents

List of Figures xi

List of Tables xiii

About This Guide xv
What This Guide Contains xv
What You Should Know Before Reading This Guide xvi
Suggestions for Further Reading xvi
Conventions Used in This Guide xviii

1. About the MIPSpro Compiler System 3

2. Using the MIPSpro Compiler System 9
Selecting a Compiler 10
Using a Defaults Specification File 10
Using Command-Line Options 12
Setting an Environment Variable 12
When to Use –n32 or –64 12
Object File Format and Dynamic Linking 13
Executable and Linking Format 13
Dynamic Shared Objects 14
Position-Independent Code 14
Source File Considerations 15
Source File Naming Conventions 15
Header Files 16
Specifying a Header File 16
Creating a Header File for Multiple Languages 17

Using Precompiled Headers in C and C++ 17
About Precompiled Headers 18
Automatic Precompiled Header Processing 18
Other Ways to Control Precompiled Headers 22
PCH Performance Issues 23
Compiler Drivers 24
Default Behavior for Compiler Drivers 24
General Options for Compiler Drivers 25
Linking 30
Invoking the Linker Manually 30
Linker Syntax 31
32
Linker Example 32
Linking Assembly Language Programs 33
Linking Libraries 33
Specifying Libraries and DSOs 33
Examples of Linking DSOs 35
Linking to Previously Built Dynamic Shared Objects 35
Linking Multilanguage Programs 35
Finding an Unresolved Symbol With *ld* 38
Debugging 38
Getting Information About Object Files 39
Disassembling Object Files with *dis* 40
dis Syntax 40
dis Options 40
Listing Parts of DWARF Object Files With *dwarfdump* 41
dwarfdump Syntax 41
dwarfdump Options 42
Listing Parts of ELF Object Files and Libraries with *elfdump* 43
elfdump Syntax 43
elfump Options 43

Determining File Type with *file* 45

file Syntax 45

file Example 45

Listing Symbol Table Information: *nm* 46

nm Syntax 46

nm Symbol Table Options 46

nm Example of Obtaining a Symbol Table Listing 48

Determining Section Sizes with *size* 49

size Syntax 50

size Options 50

size Example 51

Removing Symbol Table and Relocation Bits with *strip* 51

strip Syntax 51

Using the Archiver to Create Libraries 52

ar Syntax 53

ar Options 53

ar Examples 55

3. Using Dynamic Shared Objects 59

Benefits of Using DSOs 60

DSOs Minimize Overall Memory Use 60

Executables Linked with DSOs Are Smaller 60

DSOs Are Easier To Use, Build, and Debug 60

Executables Using DSOs Don’t Have to be Relinked 61

DSOs and Executables Are Mapped Into Memory 61

Using DSOs 62

DSOs vs. Archive Libraries 62

Using QuickStart 62

Guidelines for Using Shared Libraries 63

Choosing DSO Library Members 63

Tuning Shared Library Code 65

Taking Advantage of QuickStart 66

Building DSOs 69
Creating DSOs 69
Making DSOs Self-Contained 70
Controlling Symbols to Be Exported or Loaded 71
Building DSOs With C++ 71
Run-Time Linking 72
Searching for DSOs at Run Time 72
Searching for DSOs at Run Time Under the o32-Bit ABI 72
Searching for DSOs at Run Time Under the n32-Bit ABI 73
Searching for DSOs at Run Time Under the 64-Bit ABI 74
Run-Time Symbol Resolution 74
Building a DSO with **–Bsymbolic** 74
Converting Archive Libraries to DSOs 76
Dynamic Loading Under Program Control 77
Versioning of DSOs 79
The Versioning Mechanism 79
What Is a Version? 80
Building a Shared Library Using Versioning 81
Example of Versioning 82

4. Optimizing Program Performance 85
Optimization Overview 86
Benefits of Optimization 86
Optimization and Debugging 86
Using the Optimization Options 86
Performance Tuning with Interprocedural Analysis 88
Inlining 90
Benefits of Inlining 90
Inlining Options for Routines 91
Options To Control Inlining Heuristics 93
Common Block Padding 93
Alias and Address Taken Analysis 95
The –IPA:alias=ON Option 95
The –IPA:addressing=ON Option 96

Controlling Loop Nest Optimizations 96
Running LNO 96
LNO Optimizations 99
Loop Interchange 99
Blocking and Outer Loop Unrolling 100
Loop Fusion 101
Loop Fission/Distribution 102
Prefetching 104
Gather-Scatter Optimization 104
Compiler Options for LNO 105
Controlling LNO Optimization Levels 106
Controlling Fission and Fusion 106
Controlling Gather-Scatter 107
Controlling Cache Parameters 107
Controlling Permutation Transformations and Cache Optimization 109
Controlling Prefetch 110
Dependence Analysis 111
Pragmas and Directives for LNO 111
Fission/Fusion 112
Blocking and Permutation Transformations 113
Prefetch 116
Fill/Align Symbol 117
Dependence Analysis 120
Controlling Floating Point Optimization 121
–OPT:roundoff=n 122
–OPT:IEEE_arithmetic=n 123
Other Options to Control Floating Point Behavior 124
Debugging Floating-Point Problems 126
Controlling Miscellaneous Optimizations With the –OPT Option 127
Using the –OPT:Olimit=n Option 127
Using the –OPT:alias Option 127
Simplifying Code With the –OPT Option 129
Controlling Execution Frequency 129

The Code Generator 130
Overview of the Code Generator 130
Code Generator and Optimization Levels 131
An Example of Local Optimization for Fortran 131
Code Generator and Optimization Levels –O2 and –O3 132
If Conversion 132
Cross-Iteration Optimizations 134
Read-Read Elimination 134
Read-Write Elimination 134
Write-Write Elimination 135
Common Sub-expression Elimination 135
Loop Unrolling 135
Recurrence Breaking 136
Software Pipelining 137
Global Code Motion 137
Benefits of GCM 139
Steps Performed By the Code Generator at Levels **–O2** and **–O3** 139
Modifying Code Generator Defaults 141
Miscellaneous Code Generator Performance Topics 141
Prefetch and Load Latency 142
Frequency and Feedback 142
Controlling the Target Architecture 143
Controlling the Target Environment 143
Programming Hints for Improving Optimization 145
Hints for Writing Programs 145
Coding Hints for Improving Other Optimization 147
Use Tables Rather Than if-then-else or switch Statements 147
Declare Variables Most Frequently Manipulated 148
Use 32-Bit or 64-Bit Scalar Variables 148
Suggestions for C and C++ Programs 148
Suggestions for C++ Programs Only 149
const reference Parameter Optimization With –Lang:alias_const 150
Using SpeedShop 151

5. Coding for 64-Bit Programs 155
Coding Assumptions to Avoid 155
sizeof(int) == sizeof(void *) 156
sizeof(int) == sizeof(long) 156
sizeof(long) == 4 156
sizeof(void *) == 4 156
Implicitly Declared Functions 157
Constants With the High-Order Bit Set 157
Arithmetic with **long** Types 157
Solving Porting Problems 158
Guidelines for Writing Code for 64-Bit Silicon Graphics Platforms 158

6. Porting Code to N32 and 64-Bit Silicon Graphics Systems 163
Compatibility 164
N32 Porting Guidelines 165
Porting Environment 166
Source Code Changes 166
Build Procedure 166
Runtime Issues 167
Porting Code to 64-Bit Silicon Graphics Systems 167
Using Data Types 168
Using Predefined Types 169
Using Typedefs 170
Maximum Memory Allocation 171
Arrays Larger Than 2 Gigabytes 171
Example of Arrays Larger Than 2 Gigabytes 171
Using Large Files With XFS 173

Index 175

List of Figures

Figure 1-1 Compiler System Flowchart 6
Figure 2-1 Compilation Control Flow for Multilanguage Programs 37
Figure 3-1 An Application Linked with DSOs 67
Figure 4-1 Compilation Process Showing Interprocedural Analysis 89
Figure 4-2 Compilation Process Showing LNO Transformations 98
Figure 6-1 Application Support Under Different ABIs 164
Figure 6-2 Library Locations for Different ABIs 165

List of Tables

Table i Topics and Manuals xvi
Table 1-1 Compiler System Functional Components 4
Table 1-2 Compiler Mode and Default Library Search Path 5
Table 1-3 Compilers and Default Libraries 5
Table 2-1 The *compiler.defaults* File Specifications 10
Table 2-2 Compilation Mode Command-Line Options 12
Table 2-3 Compilation Mode Environment Variable Specifications 12
Table 2-4 Driver Input File Suffixes 15
Table 2-5 General Driver Options 25
Table 2-6 Linker Options 31
Table 2-7 Driver Options for Debugging 39
Table 2-8 *dis* Options 40
Table 2-9 *dwarfdump* Options 42
Table 2-10 *elfdump* Options 43
Table 2-11 Symbol Table *nm* Options 46
Table 2-12 Character Code Meanings 47
Table 2-13 *size* Options 50
Table 2-14 *strip* Options 52
Table 2-15 Archiver Options 53
Table 2-16 Archiver Modifiers 54
Table 3-1 Functions to Load and Unload DSOs 77
Table 4-1 Optimization Options 87
Table 4-2 Inlining Options for Routines 91
Table 4-3 Options to Control Inlining Heuristics 93
Table 4-4 LNO Options to Control Optimization Levels 106
Table 4-5 LNO Options to Control Fission and Fusion 106
Table 4-6 LNO Option to Control Gather-Scatter 107

Table 4-7 LNO Options to Control Cache Parameters 108
Table 4-8 LNO Options to Control Transformations 109
Table 4-9 LNO Option to Control Cache Optimization 109
Table 4-10 LNO Option to Control Illegal Transformation 110
Table 4-11 LNO Options to Control Prefetch 110
Table 4-12 Options to Control Dependence Analysis 111
Table 6-1 Data Types and Sizes 168
Table 6-2 Predefined Macros 169
Table 6-3 Modifications for Applications on XFS 173

About This Guide

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

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

What This Guide Contains

This guide contains the following chapters:

- Chapter 1, “About the MIPSpro Compiler System,” provides an overview of the MIPSpro compiler system.
- Chapter 2, “Using the MIPSpro Compiler System,” describes the components and related tools of the MIPSpro compiler system and explains how to use them.
- Chapter 3, “Using Dynamic Shared Objects,” explains how to build and use dynamic shared objects.
- Chapter 4, “Optimizing Program Performance,” explains how to reduce program execution time by using optimization options and techniques.
- Chapter 5, “Coding for 64-Bit Programs” describes how to write or update code that is portable to 64-bit systems.
- Chapter 6, “Porting Code to N32 and 64-Bit Silicon Graphics Systems” explains how to port code from the old 32-bit mode to the new 32-bit mode (n32).

For an overview of the IRIX programming environment and tools available for application programming, see *Programming on Silicon Graphics Computer Systems: An Overview*.

What You Should Know Before Reading This Guide

This guide is for anyone who wants to program effectively using the MIPSpro compilers. We assume you are 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 the differences between n32-bit, 64-bit, and o32-bit compilation modes. Refer to *MIPSpro Application Porting and Transition Guide* and *MIPSpro N32 ABI Handbook* 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, which contain important information about this release of the MIPSpro compiler system.

Suggestions for Further Reading

Some online and printed documents that may be of interest to you are listed in Table i.

Table i Topics and Manuals

Topic	Document
Compiler information	Release Notes for your compiler
IRIX programming	*Topics in IRIX Programming*
Automatic parallelization	*MIPSpro Automatic Parallelizer Programmer's Guide*
Debugging a program	*dbx User's Guide*
MIPS ABI	See http://www.mipsabi.org/
Multiprocessing	Appropriate language manual, for example, *MIPSpro Fortran 77 Programmer's Guide* or *C Language Reference Manual*

Table i (continued) Topics and Manuals

Topic	Document
N32 ABI	MIPSpro N32 ABI Handbook
prof, *pixie*, and *ssrun*	*SpeedShop User's Guide*
Porting code	*MIPSpro Porting and Transition Guide*
Assembly language	*MIPSpro Assembly Language Programmer's Guide*
C language	*C Language Reference Manual*
C++ language	*C++ Programming Guide*
Fortran language	*MIPSpro Fortran 77 Programmer's Guide* *MIPSpro Fortran 90 Commands and Directives Reference*
Parallel programming	Appropriate language manual, for example, *MIPSpro Fortran 77 Programmer's Guide*
Real-time programming	*REACT Real-Time Programmer's Guide*

Silicon Graphics also provides manuals online, on the Web or in IRIS InSight. To read an online manual after installing it, type **insight** or double-click the InSight icon. It's easy to print sections and chapters of the online manuals from InSight. You can also order printed manuals from Silicon Graphics by calling SGI Direct at 1-800-800-7441. Outside the U.S. and Canada, contact your local sales office or distributor.

To read an online manual on the Web, use this URL:

http://techpubs.sgi.com/library/lib/display.cgi?4097

Silicon Graphics offers software options to assist in your software development. The compiler options include languages such as *Fortran77*, *C, C++*, and the automatic parallelizers: *pfa and pca*. *CASEVision/Workshop* provides the WorkShop toolset: the Debugger, Static Analyzer, Performance Analyzer, Tester, and Build Manager.

As a developer, you are eligible to become a member of the Silicon Graphics Developer Program at SGI. Call 1-800 -770-3033 for details. If you are developing a MIPS ABI-compliant application, you may want to consult the *MIPS ABI Frequently Asked Questions*.

You may also want to learn more about standard UNIX and ANSI C topics. For this information, consult a computer bookstore or manuals such as:

- AT&T. *UNIX System V Release 4 Programmer's Guide: ANSI C and Programming Support Tools*. Englewood Cliffs, NJ: Prentice Hall, 1990.
- Levine, Mason, and Brown. *lex & yacc*. Sebastopol. CA: O'Reilly & Associates, Inc., 1992.
- Oram and Talbott. *Managing Projects with make*. Sebastopol. CA: O'Reilly & Associates, Inc., 1991.
- American National Standards Institute, Inc. *American National Standard, Programming Language*—*C, ANSI C Standard*. ANSI X3.159-1989.
- International Standard ISO/IEC. *Programming languages—C, 9899*. 1990(E).

Conventions Used in This Guide

This guide uses these conventions and symbols:

Links	Links to other sections and chapters in this guide appear in blue. For example, Chapter 1 describes the components of the compiler system. Links to applications, files, and reference pages appear in red. For example, for information about the *cc* compiler, see the cc(1) reference page.
`Courier`	In text, the Courier font represents function names, file names, and keywords. It is also used for command syntax, output, and program listings.
`bold`	Boldface is used along with Courier font to represent user input.
italics	Words in italics represent characters or numerical values that you define. Replace the abbreviation with the defined value. Also, italics are used for manual titles, file and path names, and commands.
_ _	A double underline appears as __ in text.
[ ]	Brackets enclose optional items.
{ }	Braces enclose two or more items; you must specify at least one of the items.
|	The OR symbol separates two or more optional items.

…	A horizontal ellipsis in a syntax statement indicates that the preceding optional items can appear more than once in succession.
( )	Parentheses enclose entities and must be typed.

The following two examples illustrate the syntax conventions:

```
DIMENSION a(d) [,a(d)] …
```

indicates that you must type the Fortran keyword DIMENSION as shown, that the user-defined entity `a(d)` is required, and that you can specify one or more of `a(d)`. The parentheses ( ) enclosing `d` are required. The following example:

```
{STATIC | AUTOMATIC} v [,v] …
```

indicates that you must type either the STATIC or AUTOMATIC keyword as shown, that the user-defined entity `v` is required, and that you can specify one or more `v` items.

Chapter 1

About the MIPSpro Compiler System

This chapter presents a brief overview of the MIPSpro compiler system.

Chapter 1

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 higher performance than the old 32-bit mode available in IRIX releases prior to 6.1. When you compile **–n32**, 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" 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."
- creates *Position-Independent Code*, (PIC) by default, to support dynamic linking. See "Position-Independent Code,"for additional information.

Table 1-1 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*
Profiler	Analyze program performance	*prof*, *pixie*, *ssrun*
Archiver	Produce object-file libraries	*ar*
Linker	Link object files	*ld*
Runtime linker	Link Dynamic Shared Objects at runtime	*rld*
Debugger	Debug programs	*dbx*

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

- Macro preprocessor (*cpp*)
- Parallel analyzer (*pca*, *fef77p*, *fef90p*)
- Scalar optimizer (*copt*)
- Compiler front end
- Compiler back end
- Linker (*ld*)

You can invoke a compiler driver with various options (described in "General Options for Compiler Drivers" on page 25) 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/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.")

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*).

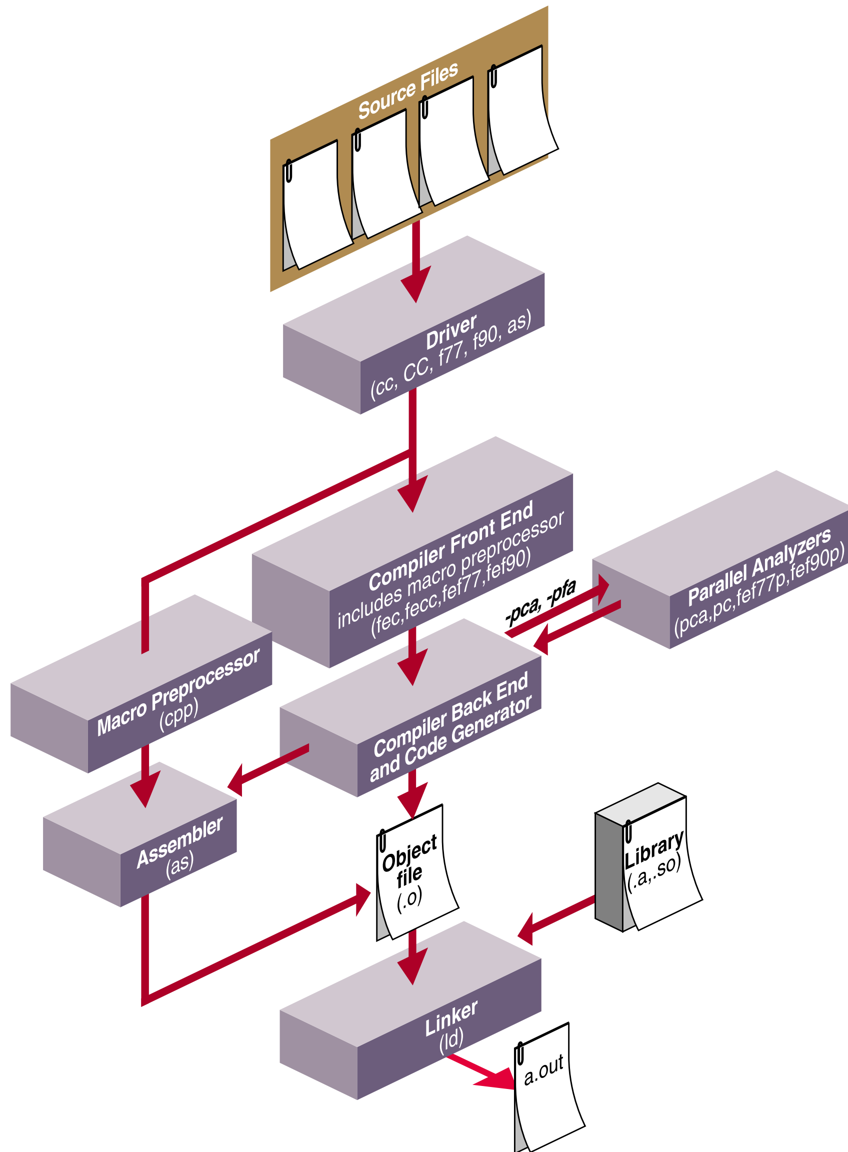

Figure 1-1 Compiler System Flowchart

Chapter 2

Using the MIPSpro Compiler System

This chapter describes the components of the MIPSpro compiler system, and explains how to use them.

Chapter 2

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 a Compiler" explains how to specify n32-bit, 64-bit, or o32-bit compilation mode and how to set up a *compiler.defaults* file.
- "Object File Format and Dynamic Linking" discusses object files including executable and linking format, dynamic shared objects, and position-independent code.
- "Source File Considerations" explains source file naming conventions, and the procedure for including header files.
- "Using Precompiled Headers in C and C++" describes automatic and manual precompiled header processing.
- "Compiler Drivers" lists and explains general compiler-driver options.
- "Linking" 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.
- "Debugging" describes the compiler-driver options for debugging.
- "Getting Information About Object Files" provides information on how to use the object file tools to analyze object files.
- "Using the Archiver to Create Libraries" explains how to use the archiver, *ar*.

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

Selecting a Compiler

You can select a compiler by explicitly specifying a command-line option, an environment variable, and by specifying the defaults in a specification file. This section covers the following topics:

- "Using a Defaults Specification File"
- "Using Command-Line Options"
- "Setting an Environment Variable"
- "When to Use –n32 or –64"

Using a Defaults Specification File

You can set the Application Binary Interface (ABI), instruction set architecture (ISA), and processor type without explicitly specifying them. Just set the environment variable COMPILER_DEFAULTS_PATH to a colon separated list of paths designating where the compiler is to look for a *compiler.defaults* file. If no *compiler.defaults* file is found or if the environment variable is not set, the compiler looks for */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 specifier is as follows:

```
-DEFAULT:[abi={o32|n32|64}] [:isa=mipsn] [:proc={r4k|r5k|r8k|r10k}] [:opt={0-3}]
[arith={1-3}]
```

This format of the *compiler.defaults* file is described in Table 2-1.

Table 2-1 The *compiler.defaults* File Specifications

Specifier	Description
abi=o32	Compiles 32-bit objects.
abi=n32	Compiles "new" 32-bit objects (high performance).
abi=64	Compiles 64-bit objects.

Table 2-1 (continued) The *compiler.defaults* File Specifications

Specifier	Description
isa=mips2	Generates code using the MIPS II instruction set (MIPS I plus R4000 extensions, for R4000 and above).
isa=mips3	Generates code using the full MIPS III instruction set (for R4000 and above).
isa=mips4	Generates code using the MIPS IV instruction set (for R5000, R8000, and R10000).
proc=r4k	Schedules code for the R4000 processor and adds the appropriate paths to the head of the library search path.
proc=r5k	Schedules code for the R5000 processor and adds the appropriate paths to the head of the library search path.
proc=r8k	Schedules code for the R8000 processor and adds the appropriate paths to the head of the library search path.
proc=r10k	Schedules code for the R10000 processor and adds the appropriate paths to the head of the library search path.
opt={0-3}	Defines an optimization level: **–O0** (default), **–O1**, **–O2**, **–O3**, or **–O4**.
arith={1-3}	Indicates which **–OPT:IEEE_arith=** value to use as the default.

Use the **–show_defaults** option to print the *compiler.defaults* being used (if any) and the 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 command:

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

overrides a *compiler.defaults* file that sets **–DEFAULT:abi=n32:isa=mips4:proc=r10k**. and compiles **–64 –mips4 –r10000**.

The following command overrides the *compiler.defaults* file and sets the ABI to **–o32** and the ISA to **–mips2** (**–o32** 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 reference (man) pages for your compiler for information about default settings.

Using Command-Line Options

You can specify command-line options to override a *compiler.defaults* file. Table 2-2 lists the compilation mode options.

Table 2-2 Compilation Mode Command-Line Options

Option	Description
–n32	Compiles the source code to new 32-bit mode (high performance, native 32-bit integers; **long long** provides 64-bit integers). The default is **–mips3**, if you do not specify **–mips4**.
–64	Compiles the source code to 64-bit mode (the default is **–mips4** if you do not specify **–mips3**.
–o32	Compiles the source code to 32-bit mode (the default is **–mips2**, if you do not specify **–mips1**).

Setting an Environment Variable

You can set an environment variable (shown in Table 2-3) to specify the compilation mode to use.

Table 2-3 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 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 4 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 Dynamic Shared Objects, described below.

Types of ELF object files are as follows:

- Relocatable files contain code and data in a format suitable for linking with other object files to make a shared object or executable.
- Dynamic Shared Objects 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 are programs ready for execution. They may or may not be dynamically linked.

Note: The current compiler system has no facility for creating or linking COFF executables; therefore, you must recompile COFF executables.

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

- *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 for 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."

Note: COFF static shared libraries are not supported under this release. The current compiler system has no facilities for generating static shared libraries.

Position-Independent Code

Dynamic linking requires that all object code used in the executable be position-independent code. 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."

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 doesn't rely on standard libraries. In this case, you should use the **–non_shared** option to the compiler

to negate the default option, **–KPIC**. For convenience, the C library and math library are provided in non-shared format as well as in DSO format (although the non-shared versions are not installed by default). You can link these libraries **–non_shared** with other non-PIC files.

Source File Considerations

This section describes conventions for naming source files and including header files. Topics covered include:

- "Source File Naming Conventions"
- "Header Files"
- "Using Precompiled Headers in C and C++"

Source File Naming Conventions

Each compiler driver recognizes the type of an input file by the suffix assigned to the filename. Table 2-4 describes the possible filename suffixes.

Table 2-4 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	Dynamic shared object library

The following example compiles preprocessed source code:

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

The Fortran 77 compiler, *f77,* assumes the file *tickle.i* contains Fortran statements (because the Fortran driver is specified). *f77* also assumes the file has already been preprocessed (because the suffix is *.i*), and therefore does not invoke the preprocessor.

Header Files

Header files, also called *include* files, contain information about the libraries with which they're 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 such definitions conveniently and consistently in each source file that uses any of the library routines.

By convention, header filenames 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 **printf()**.

For detailed information about standard header files and libraries, see the International Standard ISO/IEC. *Programming languages—C, 9899*. 1990. Also see "Using Typedefs" on page 170 for information about the *inttypes.h* header file.

Specifying a Header File

The *#include* directive tells the preprocessor to replace the *#include* line with the text of the indicated header file. The usual way to specify a header file is with the line

```
#include <filename>
```

where *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 *including* file). If the preprocessor doesn't find the requested file, it searches the other directories as in the angle-bracket specification.

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-bit 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"
- "Automatic Precompiled Header Processing"
- "Other Ways to Control Precompiled Headers"
- "PCH Performance Issues"

About Precompiled Headers

The precompiled header (PCH) file mechanism is available through the compiler front end: *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*/*fecc* takes a snapshot of the state of the compilation at a particular point and writes 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 the precompiled header processing by using the **–pch** option (**–Wf,––pch** in 32-bit mode) on the command line. With the PCH mechanism enabled, *fec/fecc* searches for a qualifying PCH file to read in and/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 can not be inside a declaration started within a header file, and it can not 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 can not 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__** must not have appeared.
- Use of the `#line` preprocessing directive must not have appeared.
- `#pragma no_pch` must not have appeared.

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/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), *fec/fecc* 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/or 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. For example,

  ```
  #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/or create a PCH file.

PCH Performance Issues

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

In general, writing out a precompiled header file doesn't 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 250K bytes to several megabytes or more), and so you probably don't 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/or 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, such as *cc* and *f77* call subsystems that compile, optimize, assemble, and link your programs. This section describes:

- "Default Behavior for Compiler Drivers"
- "General Options for Compiler Drivers"

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*), then 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.

General Options for Compiler Drivers

The command-line options for MIPSpro compiler drivers are listed and explained in Table 2-5. The table lists only the most frequently used options, not all available options. See the appropriate compiler reference (manual) page for additional details.

In addition to the general options in Table 2-5, each driver has options that you typically won't use. These options primarily aid compiler development work. For information about nonstandard driver options, consult the appropriate driver reference page. Click the word cc to view the *cc*(1) and *CC*(1) reference pages.

You can use the compiler system to generate profiled programs that, when executed, provide operational statistics. To perform this procedure, use the **–p** compiler option (for pc sampling information) and the *prof* command (for profiles of basic block counts). Refer to Chapter 4, "Optimizing Program Performance," for details.

Table 2-5 General Driver Options

Option	Purpose
–n32	Generates an new 32-bit object (see "Using a Defaults Specification File" on page 10).
–o32	Generates an old 32-bit object.
–64	Generates a 64-bit object (see "Using a Defaults Specification File" on page 10).
–ansi	Compiles strict ANSI/ISO C. Preprocessing adds only standard predefined symbols to the name space, and standard include files declare only standard symbols.
–avoid_gp_overflow	Asserts flags that are intended to avoid GOT overflow. See **–multigot** option, below.
–c	Prevents the linker from linking your program after code generation. This option forces the driver to produce a *.o* file after the back-end phase, and prevents the driver from producing an executable file.
–C	Used with the **–P** or **–E** option. Prevents the macro preprocessor from stripping comments. Use this option when you suspect the preprocessor is not producing the intended code and you want to examine the code with its comments. For C and C++ compiles only.

Table 2-5 (continued) General Driver Options

Option	Purpose
–cord	Runs the procedure rearranger, *cord*(1) on the resulting file after linking. Rearranging improves the paging and caching performance of the program's text. The output of *cord* is placed in *a.out*, by default, or a file specified by the **–o** option. If you don't specify **–feedback**, then *outfile.fb* is used as the default.
–cckr	K&R/Version7 C compatibility compilation mode. Preprocessing may add more predefined symbols to the name space than in **–ansi** mode. Compilation adheres to the K&R language semantics.
–D*name*[**=***def*]	Defines a macro *name* as if you had specified a **#define** in your program. If you do not specify a definition with **=***def, name* is set to 1.
–DEBUG:*options*	Debugs run-time code and generates compile, link, and run-time warning messages. See also **–g**.
–E	Runs only the macro preprocessor and sends results to the standard output. To retain comments, use the **–C** option as well. Use **–E** when you suspect the preprocessor is not producing the intended code.
–feedback	Use with the **–cord** option to specify feedback file(s). You can produce this file by using *prof* with its **–feedback** option from an execution of the instrumented program produced by *pixie*(1). Specify multiple feedback files with multiple **–feedback** options.
–fullwarn	Does various extra checks and produces additional warnings that are normally suppressed. This option is recommended for all compiles during software development. You can turn off warnings selectively by using the **–woff** option.
–g[*num*]	Produces debugging information. The default is **–g0**: do not produce debugging information. See also **–DEBUG**.
–G[*num*]	Specifies the maximum size, in bytes, of a data item that is accessed from the global pointer. The default is **–G8**.
–help	Lists the available compiler options (available only with **–n32** and **–64**).

Table 2-5 (continued) General Driver Options

Option	Purpose
–I*dirname*	Adds *dirname* to the list of directories to be searched for specified header files. These directories are always searched before the default directory, */usr/include* and */usr/include/CC* for C++.
–INLINE:*options*	Controls standalone inliner options: control application of intra-file subprogram inlining when inter-procedural analysis is not enabled.
–IPA:*options*	Controls inter-procedural analyzer options control application of inter-procedural analysis and optimization, including inlining, constant propagation, common block array padding, dead function elimination, alias analysis, and others.
–KPIC	Generates position-independent code. This is the default and is required for programs linking with dynamic shared objects. Specify **–non_shared** if you don't want to generate PIC code.
–L *directory*	In XPG4 mode, changes the algorithm of searching for libraries named in **–L** operands to look in the directory named by directory path-name before looking in the default location. Directories names in **–L** options are searched in the specified order. Multiple instances of **–L** options can be specified.
–l *library*	In XPG4 mode, searches the library named *lib.IRlibrary.a*. A library is searched when its name is encountered, so the placement of a **–l** operand is significant.
–LIST:*options*	Controls the information that is written to the *.l* file.
–mips2	Generates code using the MIPS II instruction set (MIPS I + R4000 specific extensions). Note that code compiled with **–mips2** does not run on R2000/R3000-based machines. This option implies **–o32**.
–mips3	Generates code using the full MIPS R4000 instruction set, including 64-bit code.
–mips4	Generates code using the MIPS R5000, R8000, or R10000 instruction set.
–mp	Enables the multiprocessing and DSM directives.

Table 2-5 (continued) General Driver Options

Option	Purpose
–nocpp	Suppresses running of the macro preprocessor of the source files prior to processing.
–non_shared	Turns off the default option, **–KPIC**, to produce non-shared code. This code can be linked to only a few standard libraries (such as *libc.a* and *libm.a*) that are provided in non-shared format in the directory */usr/lib/nonshared*. You should use this option only when building device drivers.
–nostdinc	Suppresses searching of standard include directories for the specified header files.
–o *filename*	Names the result of the compilation *filename*. If an executable is being generated, it is named *filename* rather than the default name, *a.out*.
–O*num*	Specifies optimization options. **–O0** turns off optimizations. **–O1** turns on local optimizations that can be done quickly. **–O2** (**–O**) turns on global optimizations. This is the default. **–O3** turns on all optimizations. For more information, see Chapter 4, "Optimizing Program Performance."
–OPT:*options*	Controls optimization options. For more information, see Chapter 4, "Optimizing Program Performance."
–p	Sets up for profiling by periodically sampling the value of the program counter (only effects the loading).
–P	Runs only the macro preprocessor on the files and puts the result of each file in a *.i* file. Specify both **–P** and **–C** to retain comments.
–pca, **–pfa**	Runs the *pca*/*pfa* preprocessor to automatically discover parallelism in the source code. Also enables the multiprocessing (**–mp**) directive. (This is an optional package.)
–pch	Enables the precompiled header processing; *fec/fecc* searches for a qualifying PCH file to read in and/or creates one for use on a subsequent compilation. (For **–o32**, use **–Wf**, **– –pch**.)

Table 2-5 (continued) General Driver Options

Option	Purpose
–S	Similar to **–c**, except that it produces assembly code in a *.s* file instead of object code in a *.o* file. **–S** provides information about the code and comments about such things as software pipelining, the loops it works on, and the results.
–show	Lists compiler phases as they are executed. Use this option to see the default options for each compiler phase along with the options you've specified.
–TARG:*options*	Controls the target architecture and machine for which code is generated. For more information, see Chapter 4, "Optimizing Program Performance."
–TENV:*options*	Controls the target environment assumed by the compiler. For more information, see Chapter 4, "Optimizing Program Performance."
–U*name*	Overrides a definition of the macro *name* that you specified with the **–D** option, or that is defined automatically by the driver. Note that this option does not override a macro definition in a source file, only on the command line.
–woff *n*	Suppresses ANSI/ISO warning message number *n*. Suppress multiple warning numbers by using a comma-separated list **(–woff n1,n2**...), a range of warning numbers by using a hyphen-separated list (**–woff n1-n5**), or any combination thereof. You can suppress all warning messages via **–woff all.**
–xansi	Compilation follows an extended ANSI/ISO C language semantics, which is more lenient in terms of the forms of expressions it allows. Preprocessing combines predefined macros. This is the default C compilation mode.

Note: To use 4.3 BSD extensions in C, compile using the **–xansi** or the **–D__EXTENSIONS__** option on the command line. For example:

```
cc prog.c -ansi -prototypes -fullwarn -lm -D__EXTENSIONS__
```

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) reference 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 Manually"
- "Linking Assembly Language Programs"
- "Linking Libraries"
- "Linking to Previously Built Dynamic Shared Objects"
- "Linking Multilanguage Programs"

Invoking the Linker Manually

Usually the compiler invokes the linker as the final step in compilation (as explained in "Compiler Drivers"). If object files exist that were produced by previous compilations, and you want to link them, invoke the linker by using a compiler driver instead of calling *ld* directly. Just pass the object filenames to the compiler driver in place of source filenames. If the original source files are in one language, simply invoke the associated driver and specify the list of object files. (For information about linking objects derived from several languages, see "Linking Multilanguage Programs.")

Circumstances may exist when you need to invoke *ld* directly, such as when you're building a shared object or doing special linking not supported by compiler drivers (such as building an embedded system).

Note: To build C++ shared objects, use the CC driver.

Linker Syntax

A summary of *ld* syntax follows.

`ld` *options object1* [*object2*...*objectn*]

options One or more of the options listed in Table 2-6.

object Specifies the name of the object file to be linked.

Table 2-6 contains only a partial list of linker options. Many options that apply only to creating shared objects are discussed in Chapter 3, "Using Dynamic Shared Objects." For complete information on options and libraries that affect linker processing, refer to the ld(1) reference page.

Table 2-6 Linker Options

Option	Purpose
–n32	Links new 32-bit programs and DSOs.
–o32	Links old 32-bit programs and DSOs.
–64	Links 64-bit programs and DSOs.
–elspec *filename*	Specifies an ELF layout specification file (specifies the layout of object files, programs, and shared objects). See elspec(5) for details.
–ivpad	Improves cache behavior by causing the linker to perform intervariable padding of some large variables.
–l*name*	Specifies the name of a library, where *lname* is the library name. The linker searches for *lname.so* (and then *lname.a*) first in any directories specified by **–L** *dirname* options, and then in the standard directories: */usr/lib*, */lib*, and */usr/local/lib*.
–L *dirname*	Adds *dirname* to the list of directories to be searched for (as well as libraries searched for) as specified by subsequent **–l***name* options.
–m	Produces a linker memory map, listing input and output sections of the code, in System V format.
–M	Produces a link map in BSD format, listing the names of files to be loaded.

Table 2-6 (continued) Linker Options

Option	Purpose
–nostdlib	This option must be accompanied by the **–L** *dirname* option. If the linker does not find the library in *dirname* list, then it does not search any of the standard library directories.
–o *filename*	Specifies a name for your executable. If you do not specify *filename*, the linker names the executable *a.out*.
–read	Specifies that the linker uses the *open*(2), *lseek*(2), and *read*(2) as its preferred mode for reading object files. This option often improves performance when many object files are remotely mounted with high network latency.
–s	Strips debugging information from the program object, reducing its size. This option is useful for linking routines that are frequently linked into other program objects, but may hamper debugging.
–v	Produces verbose linker output providing information about various linker passes.
–y*symname*	Reports all references to, and definitions of, the symbol *symname*. Useful for locating references to undefined symbols.

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*.

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

If the linker doesn't find an appropriate library, it looks in */usr/local/lib* for *libcurses.a*. (Note that the linker does not look for DSOs in */usr/local/lib*, so don't put shared objects there.) If found in any of these places, the DSO or library is linked with the objects *foiled.o* and *again.o*; otherwise an error is generated.

Note: If the linker reports GOT overflow, GOT unreachable, or GP-related errors, see the **–multigot** option. Also see the gp_overflow(5) reference page, which describes some causes of and possible solutions for overflowing the GP-relative area in the linker.

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's 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 filename 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's called unresolved.) Once a library has been searched in this way, it is never searched again. Therefore, libraries should come after object files on the command line in order to resolve as many references as possible. Note that 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 pathname or by use of the library search rules. To specify a library or DSO by path, simply include that path on the command line (relative to the current directory, or else absolute):

```
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 **–lname** option:

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

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

The default search order is to use 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 **–lname** option. 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, built using *ar* from *.o* files that were compiled with **–non_shared**. These archives must also be linked **–non_shared**. 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 for details.

If the linker tells you that a reference to a certain function is unresolved, check that function's reference page to find out which library the function is in. If it isn't in one of the standard libraries (which *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."

Note: Simply including the header file associated with a library routine is not enough; you also must specify the library itself when linking (unless it's 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."

To build an executable that uses a DSO, call a compiler driver just as you would for a non-shared library. For instance,

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

This 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. 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 now 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. In this case, you can link multilanguage programs.

Follow the steps below to link multilanguage programs. (Refer to Figure 2-1 for an illustration of the process.)

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 used. For instructions on specifying libraries, see "Linking Libraries."

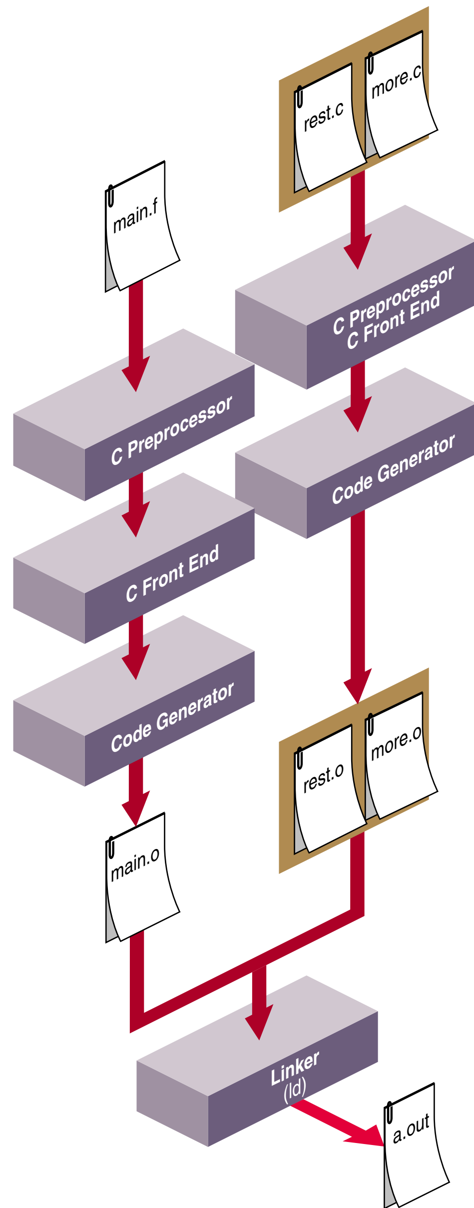

Figure 2-1 Compilation Control Flow for Multilanguage Programs

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

For specific details about compiling multilanguage programs, refer to 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're compiling a program, and *ld* tells you that you're using an unresolved symbol. However, you don't 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*.

Debugging

The compiler system provides a debugging tool, *dbx* (described in detail in the *dbx User's Guide*). In addition, CASEVision/WorkShop™ contains debugging tools. For information about obtaining WorkShop for your computer, contact your dealer or sales representative.

Before using *dbx*, specify the **–g** driver option (see Table 2-7) to produce executables containing information that the debugger can use (see the dbx(1) reference page).

Table 2-7 Driver Options for Debugging

Option	Purpose
–g0	Produces a program object with a minimum of source-level debugging information. This is the default. Reduces the size of the program object but allows optimizations. Use this option with the **–O** option after you finish debugging.
–g, **–g2**	Produces additional debugging information for full symbolic debugging. This option overrides the optimization options (**–O***num*).
–g3	Produces additional debugging information for full symbolic debugging of fully optimized code. This option makes the debugger less accurate. You can use **–g3** with an optimization option (**–O***num*).

Getting Information About Object Files

The following tools provide information on object files:

- *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 (some such sections are named *text*, *data*, and *sbss*).
- *strip* removes symbol table and relocation bits.

Note that you can trace system call and scheduling activity by using the *par* command. For more information, see the par(1) reference 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.

dis Syntax

The syntax for *dis* is:

`dis` *options filename1* `[`*filename2...filenamen*`]`

options One or more of the options listed in Table 2-8.

filename Specifies the name of one or more files to disassemble.

dis Options

Table 2-8 lists *dis* options. For more information, see the dis(1) reference page.

Table 2-8 *dis* Options

Option	Description
–b begin_addr	Starts disassembly at *begin_address*. You can specify the address as decimal, octal (with a leading **0**), or hexadecimal (with a leading **0x**).
–d section	Disassembles the named *section* as data, and prints the offset of the data from the beginning of the section.
–D *section*	Disassembles the named *section* as data, and prints the address of the data.
–e *end_address*	Stops disassembly at *end_address*. You can specify the address as decimal, octal (with a leading **0**), or hexadecimal (with a leading **0x**).
–F *function*	Disassembles the named *function* in each object file you specify on the command line.
–h	Substitutes the hardware register names for the software register names in the output.
–H	Removes the leading source line, and leaves the hex value and the instructions.

Table 2-8 (continued) *dis* Options

Option	Description
–i	Removes the leading source line and hexadecimal value of disassembly, and leaves only the instructions.
–I *directory*	Uses *directory* to help locate source code.
–l *string*	Disassembles the archive file specified by *string*.
–L	Looks up source labels for subsequent printing.
–o	Prints addresses and contents in octal. The default is hexadecimal.
–t *section*	Disassembles the named *section* as text.
–T	Specifies the trace flag for debugging the disassembler.
–V	Prints (on *stderr*) the version number of the disassembler being executed.
–w	Prints source code to the right of assembly code (produces wide output). Use this option with the **–s** option.
–x	Prints offsets in hexadecimal (the default).

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, see files in the */usr/share/src/compiler/dwarf* directory.

dwarfdump Syntax

The syntax for *dwarfdump* is:

`dwarfdump` *options filename*

options One or more of the options listed in Table 2-9.

filename Specifies the name of the object file whose contents are to be dumped.

dwarfdump Options

Table 2-9 lists *dwarfdump* options. For more information, see the dwarfdump(1) reference page.

Table 2-9 *dwarfdump* Options

Option	Dumps
–a	All sections.
–b	The **.debug_abbrev** section.
–c	The .**debug_loc** section.
–d	Uses dense mode. Prints die information of the **.debug_info** section. Does not imply the **–i** option.
–e	Uses ellipsis mode. Uses the short names for DW_TAG_* and DW_ATTR_* in the output for the **.debug_info** section.
–f	The **.debug_frame** section.
–i	The **.debug_info** section.
–l	The **.debug_line** section.
–m	The **.debug_macinfo** section.
–o	The **.reloc.debug_*** sections.
–p	The **.debug_pubnames** section.
–r	The **.debug_aranges** section.
–s	The **.debug_string** section.
–ta	The **.debug_static_funcs** and **.debug_static_vars** sections (same as **–tfv**).
–tf	The **.debug_static_funcs** section.
–tv	The **.debug_static_vars** section.
–u*ofile*	Dumps sections to the named object file, *ofile*.
–v	Prints detailed information (verbose mode).
–w	The **.debug_weaknames** section.
–y	The **.debug_types** section.

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.

elfdump Syntax

The syntax for *elfdump* is:

`elfdump` *options* *filename1* [*filename2...filenamen*]

options One or more of the options listed in Table 2-10.

filename Specifies the name of one or more object files whose contents are to be dumped.

elfump Options

Table 2-10 lists *elfdump* options. For more information, see the elfdump(1) reference page.

Table 2-10 *elfdump* Options

Option	Dumps
–a	Archive header of each member of the archive.
–A	Beginning address of a section.
–c	String table.
–C	Decoded C++ symbol names.
–cmt	The *.comment* sections.
–cnt	The *.content* sections.
–d	Range of sections.
–Dc	Conflict list in Dynamic Shared Objects.
–Dg	Global Offset Table in Dynamic Shared Objects.
–dinfo	The *.MIPS.dclass* section.
–dinst	The *.MIPS.inst* section.

Table 2-10 (continued) *elfdump* Options

Option	Dumps
–Dl	Library list in Dynamic Shared Objects.
–dsym	The *.MIPS.sym* section.
–Dsymlib	Symbol library table (*.MIPS.symlib*).
–Dt	String table entries of the dynamic symbol table in Dynamic Shared Objects.
–e	Events sections.
–f	Each file header.
–F	Literal tables (*.lit4* and *.lit8* sections).
–g	Archive symbol table.
–G	The global pointer table information.
–h	All section headers in the file.
–hash	Hash table entries.
–i	The *.interp* section, which lists the path name of the program interpreter.
–info	Prints whether the object is marked quickstart, or is corded and is marked requickstart.
–l	Suggests using *dwarfdump* to dump debugging line information (objects compiled **–n32** or **–64**).
–L	Dumps *.dynamic* (various flags and values) and *.liblist* (list of named DSOs) sections.
–n	The specified section (such as .MIPS.content, .dynamic, .got, MIPS.sym, .liblist, .conflict, .reginfo, and so forth).
–o	Each program execution header.
–op	Options section.
–p	Suppresses the printing of headings.
–r	Relocation information.

Table 2-10 (continued) *elfdump* Options

Option	Dumps
–R	Register information.
–reg	The *.reginfo* section.
–rpt	The run-time procedure table.
–s	Section contents (see **–d**).
–svr4	Information in SVR4-style format.
–t	Symbol table entries.
–T	Symbol table range.
–V	Version information only.

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, see the file(1) reference page.

file Syntax

The syntax for *file* is:

`file` *filename1* `[`*filename2...filenamen*`]`

Each *filename* is the name of a file to be examined.

file Example

Information given by *file* is self-explanatory for most kinds of files, but using *file* on object files and executables gives somewhat cryptic output.

```
file test.o a.out /lib/libc.so.1
test.o:        ELF 64-bit MSB relocatable MIPS - version 1
a.out:         ELF 64-bit MSB executable MIPS - version 1
/lib/libc.so.1:   ELF 64-bit MSB dynamic lib MIPS - version 1
```

In this example, `MSB` indicates Most Significant Byte, also called Big-Endian; `relocatable` means the object contains relocation information that allows it to be linked with other objects to form an executable; `executable` means an executable file; and `dynamic lib` indicates a DSO.

Listing Symbol Table Information: *nm*

The *nm* tool lists symbol table information for object files and archive files.

nm Syntax

The syntax for *nm* is:

`nm` *options filename1* [*filename2...filenamen*]

options — One or more of the options listed in Table 2-11.

filename — Specifies the object files or archive files from which symbol table information is to be extracted. If you do not specify a filename, *nm* assumes the file is named *a.out*.

To get XPG4 (X/Open Portability Group) format, set the environment variable, _XPG in your environment.

nm Symbol Table Options

Table 2-11 lists *nm* symbol table options. For more information, see the nm(1) reference page.

Table 2-11 Symbol Table *nm* Options

Option	Purpose
–A	Prints the listing in System V Release 4 format (the default for n32-bit and 64-bit objects).
–b	Prints the value field in octal.
–B	Prints the listing in BSD format.
–C	Prints decoded C++ names.
–d	Prints the value field in decimal (the default for System V output).

Table 2-11 (continued) Symbol Table *nm* Options

Option	Purpose
–g	Prints globally visible names.
–h	Suppresses printing of headers.
–n	Sorts external symbols by name for System V format (and for BSD default output).
–o	Prints value field in octal (System V output). When used with **–B** option, prepends the filename to the output line.
–p	Produces easily parsible, terse output similar to the BSD format.
–P	In XPG4 mode, writes information in a portable output format according to the XPG standard.
–r	Prepends the name of the object file or archive to each output line.
–u	Prints only undefined symbols.
–v	Sorts external symbols by value (default for BSD format).
–V	Prints the version number of *nm*.
–x	Prints the value field in hexadecimal.

Table 2-12 defines the one-character codes shown in an *nm* listing. Refer to the example that follows the table for a sample listing.

Table 2-12 Character Code Meanings

Key	Description
a	Local absolute data
A	External absolute data
b	Local zeroed data
B	External zeroed data
C	Common data
d	Local initialized data

Table 2-12 (continued) Character Code Meanings

Key	Description
D	External initialized data
E	Small common data
G	External small initialized data
N	Nil storage class (unused external reference)
r	Local read-only data
R	External read-only data
s	Local small zeroed data
S	External small zeroed data
t	Local text
T	External text
U	External undefined data
V	External small undefined data

nm Example of Obtaining a Symbol Table Listing

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) reference page describes the format of these sections.

size Syntax

The syntax for *size* is:

`size` *options* [*filename1 filename2...filenamen*]

options Specifies the format of the listing (see Table 2-13).

filename Specifies the object or archive files whose properties are to be listed. If you do not specify a filename, the default is *a.out*.

size Options

Table 2-13 lists *size* options. For more information, see the size(1) reference page.

Table 2-13 *size* Options

Option	Action
–A	Prints data section headers in System V format (default).
–B	Prints output in BSD-style format.
–d	Prints sizes in decimal (default).
–f	Prints data on allocatable sections including the size, permission flags, and the total of the loadable sizes.
–F	Prints data on loadable segments including the name and the total of the section sizes.
–n	Prints nonloadable and nonallocatable section sizes.
–o	Prints sizes in octal.
–4	Prints output in SVR4-style format.
–V	Prints the version of *size* that you are using.
–x	Prints sizes in hexadecimal.

size Example

An example of the *size* command and the listings produced follows.

```
size a.out
```

```
 Section         Size           Physical              Virtual
                                 Address              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*.

strip Syntax

The syntax for *strip* is:

`strip` *options filename1* [*filename2...filenamen*]

options One or more of the options listed in Table 2-14.

filename Specifies the name of one or more object files whose contents are to be stripped.

For more information, see the strip(1) reference page.

Table 2-14 *strip* Options

Option	Description
–f	Forces stripping of a DSO; you must use this option to strip a DSO.
–o *filename*	Puts the stripped information in the *filename* that you specify.

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 the code from the 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) reference page for details.

Note: *ar* simply strings together whatever object files you tell it to archive. Therefore you can use *ar* to build either non-shared or PIC libraries, depending on how the included *.o* files were built in the first place. If you do create a non-shared library with *ar*, remember to link it **–non_shared** with your other code. For information about building DSOs and converting libraries to DSOs, see Chapter 3.

ar Syntax

The syntax for *ar* is:

ar *options* [*posObject*] *libName* [*object1...objectn*]

options — Specifies the action that the archiver is to take. Table 2-15 and Table 2-16 list some of the options.

posObject — Specifies the name of an object within an archive library. It specifies the relative placement (either before or after *posObject*) of an object that is to be copied into the library or moved within the library. This parameter is required when the **a, b,** or **i** suboptions are specified with the **m** or **r** option. The last example in "ar Examples," shows the use of a *posObject* parameter.

libName — Specifies the name of the archive library you are creating, updating, or extracting information from.

object — Specifies the name(s) of the object file(s) to manipulate.

ar Options

When running the archiver, specify exactly one of the options **d, m**, **p**, **q**, **r**, **t**, or **x** (listed in Table 2-15). In addition, you can optionally specify any of the modifiers in Table 2-16.

Table 2-15 Archiver Options

Option	Purpose
–c	Suppresses the warning message that the archiver issues when it creates the archive file archive.
–d	Deletes the specified objects from the archive.
–f	Adds padding to the end of each object file archived, using the character **\n**. This enables the loader to have faster access to members in the archive while performing static linking. **Warning**: This option results in a permanent change in the size of object files.
–p	Prints the specified objects in the archive on the standard output device.

Table 2-15 (continued) Archiver Options

Option	Purpose
–q	Appends the specified object files to the end of the archive. This option is similar to the **–r** option (described below), but does not remove any older versions of the object files that may already be in the archive. Use the **–q** option when creating a new library. To avoid quadratic behavior when building an archive one object at a time, use **–qz**.
–r	Replaces or adds specified object files to the end of the archive file. If an object file with the same name already exists in the archive, the new object file overwrites it. Use the **–r** option when updating existing libraries.
–t	Prints a table of contents on the standard output for the specified object or archive file.
–x	Copies the specified objects from the archive and places them in the current directory. Duplicate files are overwritten. The last modified date is the current date (unless you specify the **–o** suboption, in which case the date stamp on the archive file is the last modified date). If no objects are specified, **–x** copies all the library objects into the current directory.

Table 2-16 Archiver Modifiers

Option	Purpose
–l	Puts the archiver's temporary files in the current working directory. Ordinarily, the archiver puts those files in */tmp* (unless the STMDIR environment variable is set, in which case *ar* stores temporary files in the directory indicated by that variable). This option is useful when */tmp* (or STMDIR) is full.
–s	Creates a symbol table in the archive. This modifier is rarely necessary since the archiver updates the symbol table of the archive library automatically. Options **–d**, **–m**, and **–r**, in particular, create a symbol table by default and thus do not require **–s** to be specified.
–v	Lists descriptive information during the process of creating or modifying the archive. When specified with the **–t** option, produces a verbose table of contents.

ar Examples

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

```
ar cr 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.

Chapter 3

Using Dynamic Shared Objects

This chapter explains how to build and use dynamic shared objects.

Chapter 3

Using Dynamic Shared Objects

A dynamic shared object (DSO) is an object file that's meant to be used simultaneously—or *shared*—by multiple applications (*a.out* files) while they're 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" explains the benefits of DSOs.
- "Using DSOs" tells you how to obtain the most benefit from using DSOs when creating your executable.
- "Taking Advantage of QuickStart" discusses an optimization you can use to make sure that the DSOs you build load as quickly as possible.
- "Building DSOs" describes how to build a DSO.
- "Run-Time Linking" discusses the run-time linker, and how it locates DSOs at run time.
- "Dynamic Loading Under Program Control" explains the use of *dlopen() and dlsym()* to control run-time linking.
- "Versioning of DSOs" discusses a versioning mechanism for DSOs that allows binaries linked against different, incompatible versions of the same DSO to run correctly.

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

Benefits of Using DSOs

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

- "DSOs Minimize Overall Memory Use"
- "Executables Linked with DSOs Are Smaller"
- "DSOs Are Easier To Use, Build, and Debug"
- "Executables Using DSOs Don't Have to be Relinked"
- "DSOs and Executables Are Mapped Into Memory"

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's required for A, B, and C, plus some small overhead. If C is an unshared library, the memory used is what's 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 aren't 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 Silicon Graphics 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 Don’t Have to be Relinked

Executables that use a DSO don’t 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 MipsIV and a MipsIII versions of a shared object. You want your program to use the MipsIV 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 as 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 MipsIV 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 MipsIV 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.” The sections after that explain how to build and optimize DSOs and how *rld* works. See the rld(1) reference (man) page for more information. The dso(5) reference page also contains more information about DSOs.

Using DSOs

Using DSOs is easy—the 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," which briefly explains how QuickStart minimizes start-up times for executables.
- "Guidelines for Using Shared Libraries," 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* isn't 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's 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, don't 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* doesn't 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* doesn't 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 can't do a QuickStart, you'll 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,

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

- "Choosing DSO Library Members" explains what routines to include and exclude when you choose library members.
- "Tuning Shared Library Code" 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. Specifically, it explains the following topics:

- Include large, frequently used routines
- Exclude infrequently used routines
- Exclude routines that use much static data
- Make libraries self-contained

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 isn't shared and can't 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's 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 a few things to consider in tuning shared library code:

- Minimize global data
- Organize to Improve locality
- Align for paging

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.

1. Try to use automatic (stack) variables. Don't use permanent storage if automatic variables work. Using automatic variables saves static data space and reduces the number of symbols visible to application processes.
2. Determine whether variables really must be external. Static symbols are not visible outside the library, so they may change addresses between library versions. Only external variables must remain constant.
3. 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. The *cflow*(1) command generates static dependency information. You can combine it with 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 don't 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 5.*x* operating system uses 4Kb 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         trcmp.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 nonoverlapping locations. 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* won't 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. 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* weren't identical, *yourApp* won't be able to use QuickStart.

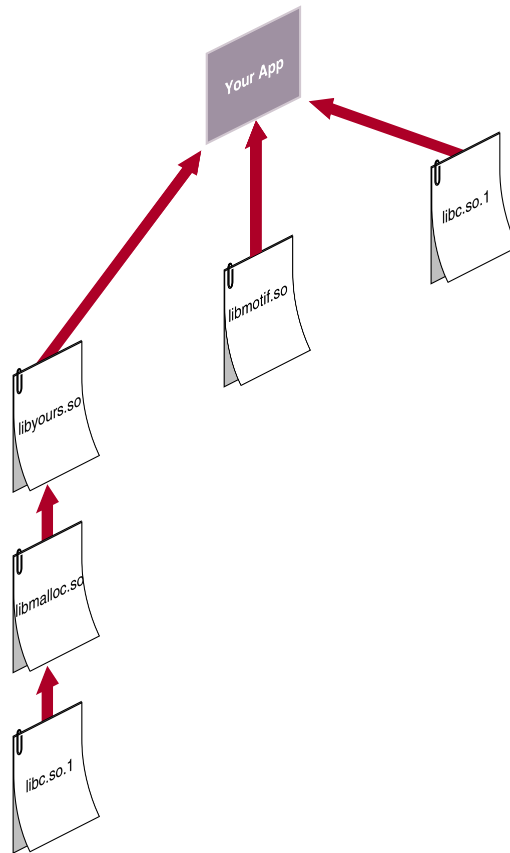

Figure 3-1 An Application Linked with DSOs

- You link with a DSO that can't use QuickStart. This may occur because the DSO wasn't 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 can't 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, you may have written your own functions to allocate memory in *libmalloc.so* for *libyours.so* to use. If you didn't 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" explains how the run-time linker searches for DSOs.

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

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

Since shared objects mentioned on the command line always take precedence over those that are not mentioned, the command above 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'll get that **malloc()**. (In the examples above, you do not need to specify **–lc**; it was added for clarity).

Thus, it's 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*(5) reference 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 won't 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 "Position-Independent Coding in Assembly Language" 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"
- "Controlling Symbols to Be Exported or Loaded"
- "Building DSOs With C++"

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 above 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're 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*.

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 won't be exported. (For more information about exported symbols, see "Controlling Symbols to Be Exported or Loaded.") 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* won't 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

By default, to help avoid conflicts, symbols defined in an archive or a DSO that's used to build another DSO aren't externally visible. You can explicitly export or hide symbols with the **–exported_symbol** and **–hidden_symbol** 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_fil*e or *-hiddens_file* option:

```
-exports_file yourFile
-hiddens_file anotherFile
```

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

Another useful option, **–delay_load**, prevents a library from being loaded until it's 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 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) reference 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 the option to *ld*. See the CC(1) reference 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"

Searching for DSOs at Run Time

When you run a dynamically linked executable, the run-time linker, *rld*, identifies the DSOs required by the executable, loads the required DSOs, and if necessary relocates DSOs within the process's 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.

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

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

1. the path of the DSO in the *liblist* (if an explicit path is given)
2. RPATH environment variable, if defined in the main executable
3. LD_LIBRARY_PATH, if defined
4. */usr/lib:/lib* is used as the default search path directory

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. Thus, if an application using the old 32-bit ABI sets LD_LIBRARY_PATH to:

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

the run-time linker searches the directory */d/src/lib1*, then the directory */d/src/lib2*, and then the current directory.

Note: For security reasons, if an executable has its set-user-ID or set-group-ID bits set, the run-time linker ignores the environment variables LD_LIBRARY_PATH and _RLD_ROOT. However, it still searches the directories in RPATH and the default path.

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

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

1. The path of the DSO in the *liblist* (if an explicit path is given)
2. RPATH environment variable, if defined in the main executable

3. LD_LIBRARYN32_PATH if defined, otherwise LD_LIBRARY_PATH, if defined
4. _RLDN32_ROOT is used for the list of paths
5. */usr/lib32:/lib32* is used as the default search path directory

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

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

1. The path of the DSO in the *liblist* (if an explicit path is given)
2. RPATH environment variable, if defined in the main executable
3. LD_LIBRARY64_PATH if defined, otherwise LD_LIBRARY_PATH, if defined
4. _RLD64_ROOT is used for the list of paths
5. */usr/lib64:/lib64* is used as the default search path directory

Run-Time Symbol Resolution

Dynamically linked executables can contain symbol references that aren't 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 aren't 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're used, so a program can run with unresolved text symbols, as long as the unresolved symbols aren't 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 doesn't 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 doesn't define *foo*, then the link will fail. However, if you convert *mylib.a* to a DSO, then any program that uses the DSO and doesn't 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 don't 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."

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()**. 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) reference page.

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

```
include <dlfcn.h>
void *dlhandle;
   ..
dlhandle = dlopen("/usr/lib/mylib.so", RTLD_LAZY);
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.

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 Silicon Graphics and includes the following topics:

- "The Versioning Mechanism"
- "What Is a Version?"
- "Building a Shared Library Using Versioning"
- "Example of Versioning"

The Versioning Mechanism

In the IRIX 5.0.1 release, a mechanism for the versioning of shared objects was introduced for the Silicon Graphics 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 Silicon Graphics systems are marked 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*(1) 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	is 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	is the literal string *sgi*.
major	is the major version number, which is a string of digits [0-9].
.	is a literal period.
minor	is 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 filename *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 doesn't use that part of the interface that is changed.

Chapter 4

Optimizing Program Performance

This chapter explains how to reduce program execution time by using optimization techniques.

Chapter 4

Optimizing Program Performance

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

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

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

See the *MIPSpro Automatic Parallelizer Programmer's Guide* for information about the optional parallelizers: *pca* and *pfa*. You can find additional information about optimization in *MIPSpro 64-Bit Porting and Transition Guide*, Chapter 6, "Performance Tuning." For information about writing code for 64-bit programs, see Chapter 5, "Coding for 64-Bit Programs." For information about porting code to **–n32** and **–64**, see Chapter 6, "Porting Code to N32 and 64-Bit Silicon Graphics Systems."

Optimization Overview

This section covers optimization benefits and debugging.

Benefits of Optimization

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

Optimization and Debugging

Optimize your programs only when they are 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*.

Using the Optimization Options

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

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

Table 4-1 Optimization Options

Option	Result
–O0	Performs no optimization that may complicate debugging. No attempt is made to minimize memory access by keeping values in registers, and little or no attempt is made to eliminate redundant code. This is the default.
–O1	Performs as many local optimizations as possible without affecting compile-time performance. No attempt is made to minimize memory access by keeping values in registers, but much of the locally redundant code is removed.
–O2, –O	Performs extensive global optimization. The optimizations at this level are generally conservative in the sense that they: (1) provide code improvements commensurate with the compile time spent (2) are almost always beneficial (3) avoid changes that affect such things as floating point results
–O3	Performs aggressive optimization. The additional optimization at this level focuses on maximizing code quality even if that requires extensive compile time or relaxing language rules. **–O3** is more likely to use transformations that are usually beneficial but can hurt performance in isolated cases. This level may cause noticeable changes in floating point results due to relaxing expression evaluation rules (see the discussion of floating point optimization and the **–OPT:roundoff=2** option below).
–Ofast	Uses optimizations selected to maximize performance for the given SGI target platform. The selected optimizations always enable the full instruction set of the target platform (for example, **–mips4** for an R10000). Although the optimizations are typically safe, they may affect floating point accuracy due to rearrangement of computations.

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

Performance Tuning with Interprocedural Analysis

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

Note: If you are using the automatic parallelizer (**–pfa** or **–pca**), 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:

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

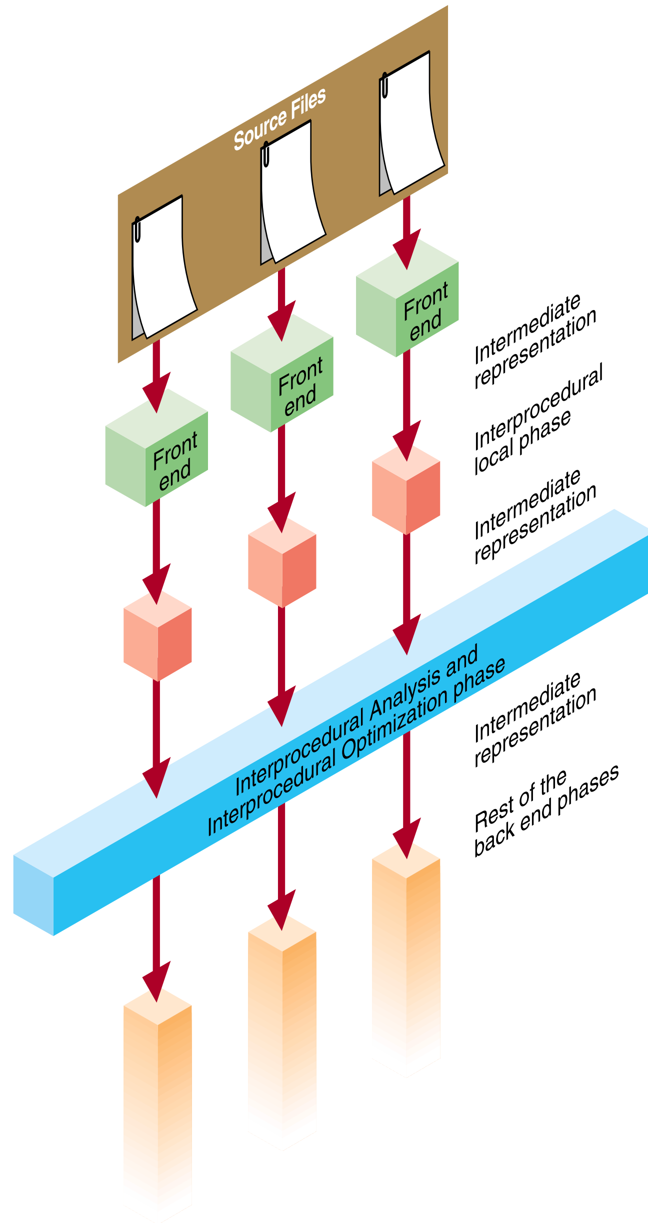

Figure 4-1 Compilation Process Showing Interprocedural Analysis

Typically, you invoke IPA with the **–IPA:** option group to *f77*, *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) reference page.

This section covers some IPA options including:

- “Inlining”
- “Common Block Padding”
- “Alias and Address Taken Analysis”

Inlining

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

- “Benefits of Inlining”
- “Inlining Options for Routines”
- “Options To Control Inlining Heuristics”

Benefits of Inlining

Your code may benefit from inlining for the following reasons:

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

Inlining Options for Routines

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

Table 4-2 Inlining Options for Routines

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

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

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

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

The –INLINE:none and –INLINE:all Options

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
```

Options To Control Inlining Heuristics

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

Table 4-3 Options to Control Inlining Heuristics

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

Common Block Padding

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

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

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

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

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

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

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

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

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

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

Alias and Address Taken Analysis

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

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

- “The –IPA:alias=ON Option”
- “The –IPA:addressing=ON Option”

The –IPA: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

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

This section covers the following topics:

- “Running LNO”
- “LNO Optimizations”
- “Compiler Options for LNO”
- “Pragmas and Directives for LNO”

Running LNO

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

After LNO performs high-level transformations, it may be desirable to view the transformed code in the original source language. Two translators that are integrated into the back end translate the compiler internal code representation back into the original source language after the LNO transformation (and IPA inlining). You can invoke either one of these translators by using the 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 usually compilable Silicon Graphics Fortran representation of the original program after the LNO phase (see Figure 4-2). LNO

is not a preprocessor, which means that recompiling the *.w2f.f* file directly may result in an executable that is different from the original compilation of the *.f* file.

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.

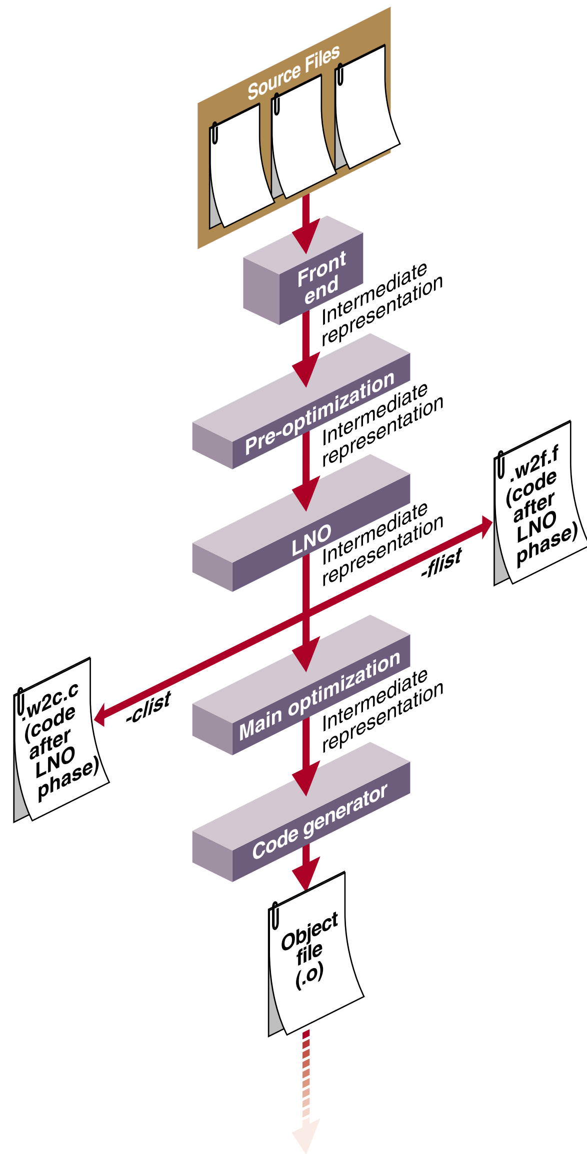

Figure 4-2 Compilation Process Showing LNO Transformations

LNO Optimizations

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

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

Loop Interchange

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

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

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

Interchanging the loops improves performance.

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

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

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

Blocking and Outer Loop Unrolling

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

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

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

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

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

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

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

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

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

Loop Fusion

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

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

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

In each loop, you need to do one store and one load in every iteration (the remaining loads are eliminated by the software pipeliner). If *n* is 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 below) to create larger perfectly nested loops. In other cases, LNO decides whether or not to fuse two loops by using a heuristic based on loop sizes and the number of variables common to both loops.

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

Prefetching

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

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

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

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

Prefetching can improve performance in compute-intensive operations where data is too large to fit in the cache. Conversely, prefetching won't 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. IF conversion does not

work well when the ‘if’ is frequently not taken. An alternative is to gather-scatter the loop, so the loop is divided as follows:

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

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

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

Compiler Options for LNO

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

- “Controlling LNO Optimization Levels”
- “Controlling Fission and Fusion”
- “Controlling Gather-Scatter”
- “Controlling Cache Parameters”
- “Controlling 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.

Controlling LNO Optimization Levels

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

Table 4-4 LNO Options to Control Optimization Levels

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

Controlling Fission and Fusion

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

Table 4-5 LNO Options to Control Fission and Fusion

Option	Description
–LNO:fission={0,1,2}	**0** Performs no fission. **1** Uses normal heuristics when deciding on loop fission (the default). **2** Tries fission before fusion when trying to create perfect nests, and fissions inner loops as much as possible.
–LNO:fusion={0,1,2}	**0** Performs no fusion. **1** Uses normal heuristics when deciding on loop fusion (the default). **2** Aggressively fuses outer loops even if fusing destroys perfect nests; tries fusion before fission when trying to create perfect nests.
–LNO:fusion_peeling_limit=n	Sets the limit (n>=0) for number of iterations allowed to be peeled in fusion. The default is 5.
–LNO:fission_inner_register_limit=n	Sets the limit (n>=0) for estimated register usage of loop bodies after inner loop fission. The default is processor specific.

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

Controlling Gather-Scatter

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

Table 4-6 LNO Option to Control Gather-Scatter

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

Controlling Cache Parameters

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

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

For example, on a machine with 128Mb of main memory, you can block for it by using the parameters below, for example, **–LNO:cs3=128M:ls3=...**. In this case, **assoc3** is ignored and doesn't have to be specified. Instead, you must specify **is_mem3..**, since virtual memory is fully associative.

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

Table 4-7 LNO Options to Control Cache Parameters

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

Controlling Permutation Transformations and Cache Optimization

Table 4-8 lists options that control transformations.

Table 4-8 LNO Options to Control Transformations

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

Table 4-9 lists the LNO option that indicates how hard to try to optimize for the cache.

Table 4-9 LNO Option to Control Cache Optimization

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

Table 4-10 lists the LNO option to control illegal transformation.

Table 4-10 LNO Option to Control Illegal Transformation

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

Controlling Prefetch

Table 4-11 lists LNO options that control prefetch operations.

Table 4-11 LNO Options to Control Prefetch

Option	Description
–LNO:prefetch=[0,1,2]	Enables or disables prefetching. **0** Disables prefetch. **1** Enables prefetch but is conservative. **2** Enables prefetch and is aggressive. The default is enabled and conservative for the R5000/R10000, and disabled for all previous processors.
–LNO:prefetch_ahead=[*n*]	Prefetches the specified number of cache lines ahead of the reference. The default is **2**.
–LNO:prefetch_leveln=[on,off] –LNO:pfn=[on,off]	Selectively enables/disables prefetching for cache level *n* where *n* ranges from [1..4].
–LNO:prefetch_manual=[on,off]	Ignores or respects manual prefetches (through pragmas). **on** Respects manual prefetches (the default for R10000). **off** Ignores manual prefetches (the default for all processors except R10000).

Dependence Analysis

Table 4-12 lists options that control dependence analysis.

Table 4-12 Options to Control Dependence Analysis

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

Pragmas and Directives for LNO

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

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

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

This section covers:

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

Fission/Fusion

The following pragmas/directives control fission and fusion.

C*$* AGGRESSIVE INNER LOOP FISSION

#pragma aggressive inner loop fission

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

C*$* FISSION [(*n***)]**

#pragma fission [(*n***)]**

C*$* FISSIONABLE

#pragma fissionable

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

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

#pragma fuse [(*n,level***)]**

C*$* FUSABLE

#pragma fusable

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

C*$* NO FISSION

#pragma no fission

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

C*$* NO FUSION

#pragma no fusion

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

Blocking and Permutation Transformations

The following pragmas/directives control blocking and permutation transformations.

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

#pragma interchange (i,j,k)

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

C*$* NO INTERCHANGE

#pragma no interchange

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

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

#pragma blocking size (n1,n2)

The loop specified, if it is involved in a blocking for the primary (secondary) cache, will have a blocksize of n1 {n2}. The compiler tries to include this loop within such a block. If a 0 blocking size is 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 didn't, add the following pragma just before the `k` loop.

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

This pragma suggests that the compiler generates a nest like:

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

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

C*$* NO BLOCKING

#pragma no blocking

Prevent the compiler from involving this loop in cache blocking.

C*$* UNROLL (*n*)

#pragma unroll (*n*)

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

For example, the following code:

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

becomes:

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

and not:

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

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

```
C*$* INTERCHANGE i,j
   DO j = 1, 10
C*$* UNROLL 2
```

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

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

#pragma blockable (i,j,k)

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

Prefetch

The following pragmas/directives control prefetch operations.

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

#pragma prefetch [(*n1*,*n2*)]

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

C*$* PREFETCH_MANUAL[(*n*)]

#pragma prefetch_manual[(*n*)]

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

C*$* PREFETCH_REF=[*obj*], stride=[*str*][,*str*], level=[*lev*][,*lev*], kind=[*rd/wr*], size=[*sz*]

#pragma prefetch_ref =[*obj*], stride=[*str*][,*str*], level=[*lev*][,*lev*], kind=[*rd/wr*], size=[*sz*]

obj specifies the reference itself, for example, an array element `A(i, j)`. *obj* can also be a scalar, for example, `x`, an integer. Must be specified.

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

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

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

size specifies the size (in Kbytes) of this 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=A, size=[*num*]

#pragma prefetch_ref_disable=A,size=[*num*]

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

Scope: entire function containing the pragma.

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 (s, *L1cacheline***)**
#pragma fill_symbol (s, *L1cacheline***)**

C*$*   FILL_SYMBOL (s, *L2cacheline***)**
#pragma fill_symbol (s, *L2cacheline***)**

C*$*   FILL_SYMBOL (s, *page***)**
#pragma fill_symbol (s, *page***)**

C*$*   FILL_SYMBOL (s, *user-specified-power-of-two***)**
#pragma fill_symbol (s, *user-specified-power-of-two***)**

C*$*   ALIGN_SYMBOL (s, *L1cacheline***)**
#pragma align_symbol (s, *L1cacheline***)**

C*$*   ALIGN_SYMBOL (s, *L2cacheline***)**
#pragma align_symbol (s, *L2cacheline***)**

C*$*   ALIGN_SYMBOL (s, *page***)**
#pragma align_symbol (s, *page***)**

C*$*   ALIGN_SYMBOL (s, *user-specified-power-of-two***)**

#pragma align_symbol (s, *user-specified-power-of-two***)**

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's breaking an obvious dependence.

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

ivdep does break the dependence.

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

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

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

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

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

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

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

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

–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 (described below) are the preferred controls:

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

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

–OPT:roundoff=n

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

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

roundoff=0 Do no transformations that could affect floating point results. This is the default for optimization levels **–O0** to **–O2**.

roundoff=1 Allow transformations with limited effects on floating point results. For roundoff, limited means that only the last bit or two of the mantissa is affected. For overflow (underflow), it means that intermediate results of the transformed calculation may overflow within a factor of two of where the original expression may have overflowed (underflowed). Note that effects may be less limited when compounded by multiple transformations.

roundoff=2 Allow transformations with more extensive effects on floating point results. Allow associative rearrangement, even across loop iterations, and distribution of multiplication over addition/subtraction. Disallow only transformations known to cause cumulative roundoff errors or overflow/underflow for operands in a large range of valid floating point values.

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

roundoff=3 Allow any mathematically valid transformation of floating point expressions. This allows floating point induction variables in loops, even when they are known to cause cumulative roundoff errors, and fast algorithms for complex absolute value and divide, which overflow (underflow) for operands beyond the square root of the representable extremes.

–OPT:IEEE_arithmetic=n

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

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

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

–OPT:IEEE_arithmetic=2
: Minor degradation: allow transformations with limited effects on floating point results, as long as exact results remain exact. This option allows use of the mips4 **recip** and **rsqrt** operations.

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

As an example, consider optimizing the Fortran code fragment:

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

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

Using **IEEE_arithmetic=3**, the divide can be treated like `a(i)*(1.0/divisor)`. 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 0.1is not exact. The complex absolute value is mathematically equal to `SQRT(r*r + r*r)`. However, calculating it this way causes an overflow if `2*r*r` is greater than the maximum REAL value, even though a representable result can be calculated for a much wider range of values of *r* (at greater cost). Both of these transformations are forbidden for **roundoff=2**, but enabled for **roundoff=3**.

Other Options to Control Floating Point Behavior

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

–OPT:div_split[=(**ON | OFF**)]

Enable/disable the calculation of `x/y as x*(1.0/y)`, normally enabled by **IEEE_arithmetic=3**. 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[=(**ON | OFF**)]

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

–OPT:fast_exp[=(**ON** | **OFF**)]

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

–OPT:fast_io[=(**ON** | **OFF**)]

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

–OPT:fast_sqrt[=(**ON** | **OFF**)]

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

–OPT:fold_reassociate[=(**ON** | **OFF**)]

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[=ON]

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

–OPT:recip[=(**ON | OFF**)]
Allow use of the mips4 reciprocal instruction for `1.0/y`, normally enabled by **–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[=(**ON | OFF**)]
Allow use of the mips4 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[=(**ON | OFF**)]
The MIPS IV architecture supports fused multiply-add instructions, which add the product of two operands to a third, with a single roundoff step at the end. Because the product is not separately rounded, this can produce slightly different (but more accurate) results than a separate multiply and add pair of instructions. This is normally enabled for **–mips4**.

Debugging Floating-Point Problems

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

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

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

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

Controlling Miscellaneous Optimizations With the –OPT Option

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

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

Using the –OPT:Olimit=n Option

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

Using the –OPT:alias Option

–OPT:alias=*name*
The compilers must 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: 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 code generator and covers the following topics:

- "Overview of the Code Generator"
- "Code Generator and Optimization Levels –O2 and –O3"
- "Modifying Code Generator Defaults"
- "Miscellaneous Code Generator Performance Topics"

Overview of the Code Generator

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

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

This section covers the following topics:

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

Code Generator and Optimization Levels

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

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

An Example of Local Optimization for Fortran

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

The *.s* file for **–O0**:

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

The *.s* file for **–O1**:

```
lw $1,0($6)                    #  load i
lw $4,12($sp)                  #  load base address for b
addiu $3,$1,-1                 #  i - 1
sll $3,$3,3                    #  8 * (i-1)
lw $2,4($sp)                   #  load base address for a
addu $3,$3,$4                  #  address for b(i)
addiu $1,$1,-1                 #  i - 1
ldc1 $f0,0($3)                 #  load b
sll $1,$1,3                    #  8 * (i-1)
addu $1,$1,$2                  #  address for a(i)
sdc1 $f0,0($1)                 #  store a
```

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

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

Code Generator and Optimization Levels –O2 and –O3

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

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

If Conversion

If conversion is a transformation that converts control-flow into conditional assignments. For example, consider the following code before *if* conversion. Note that `expr1` and

`expr2` are arbitrary expressions without calls or possible side effects. For example, if `expr1` is `i++,` the following example would be wrong since '`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;
```

Benefits of *if* Conversion

Performing *if* conversion:

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

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

In the code above that was *if conv*erted, the expressions, `expr1` and `expr2`, are UNCONDITIONALLY evaluated. This can conceivably result in generation of exceptions that do not occur without *if conv*ersion. 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 conv*ert. For information about the **–TENV** option, see “Controlling the Target Environment” and the appropriate compiler reference page.

For example, suppose `expr1 = x + y;` is a floating point add, and `X=1`. Speculating 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, define `p_safe = (cond) ? p : known_safe_address;` and then `tmp1 = *p_safe;` cannot generate a spurious memory exception.

Notice that with **–TENV:X=2**, it is legal to speculate 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, CG initializes ***known_safe_address** to 1.0.

Cross-Iteration Optimizations

Four main types of cross-iteration optimizations include:

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

Read-Read Elimination

Consider the example below:

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

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

Read-Write Elimination

Sometimes a 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 example below:

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

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

Common Sub-expression Elimination

Consider the example below:

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

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

Loop Unrolling

In this example, unrolling 4 times converts this code:

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

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.
- 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 examples below), 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 (SWP) schedules innermost loops to keep the hardware pipeline full. For information about software pipelining, see *MIPSpro 64-Bit Porting and Transition Guide*, Chapter 6, "Performance Tuning." Also, for 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 (GCM) phase performs various code motion transformations in order to reduce the overall execution time of a program. The GCM phase is useful because it does the following:

- **Moves instructions in nonloop code.** In the code example below, 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 GCM, 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.

- **Moves instructions in loops with control-flow.** In the code example below, 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 GCM, 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.

- **Moves instructions across procedure calls.** In the code example below, 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 GCM, the code looks like this:

```
...
expr1;
foo();
...
```

Benefits of GCM

The benefits of GCM include:

- **Exposes more instruction-level parallelism.** GCM identifies regions and/or blocks that have excessive and/or insufficient parallelism than that provided by the target architecture. GCM 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.
- **Provides branch delay slot filling.** GCM fills branch delay slots and converts most frequently executed branches to branch-likely form (for example, *beql*, *rs*, *rt*, *L1*).
- **Enables other compiler optimizations.** As a result of performing GCM, 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. Non-loop *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-BB loop into a single BB containing OPs with "guards." 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 (SWP).
8. If not software pipelined, reverse *if* convert.
9. Reorder BBs to minimize (dynamically) the number of taken branches. Also eliminate branches to branches when possible, and remove unreachable BBs. This step also happens at **–O1**.
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.
- CG 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

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

You can control the code generator by:

- Increasing or decreasing the unroll amount.

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

  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 can't be broken (since unrolling can't 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 CG is concerned, **–O3 –OPT:swp=off** is the same as **–O2**. Since LNO does not run at **–O2**, however, the input to CG 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.

Miscellaneous Code Generator Performance Topics

This section explains a few miscellaneous topics including:

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

Prefetch and Load Latency

At **–O3**, with **–r10000**, LNO generates prefetches for memory references that are likely to miss either the L1 or the L2 cache. CG 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.

Typically several replications of a software pipelined loop that differ only in the registers used for corresponding values. (This is necessary because values may have to survive in registers across multiple iterations of the loop.)

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

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

Frequency and Feedback

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

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.

Controlling the Target Architecture

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

–o32 –n32 or **–64** This option determines the base ABI to be assumed, either the old 32-bit ABI for mips1/2 targets, or the new 32-bit (n32) and 64-bit ABI for mips3/4 targets.

–mips[**1**|**2**|**3**|**4**]
: This option identifies the instruction set architecture (ISA) to be used for generated code. It defaults to **mips2** for **–**o32-bit compilations, **mips3** for **–**n32 compilations, and **mips4** for **–**64-bit compilations.

–TARG:madd[=(**ON**|**OFF**)]
: This option enables/disables use of the multiply/add instructions for mips4 targets. These instructions multiply two floating point operands and then add (or subtract) a third with a single roundoff of the final result. They are therefore slightly more accurate than the usual discrete operations, and may cause results not to match baselines from other targets. This option may be used to determine whether observed differences are due to **madd**s. This option is ON by default for mips4 targets; otherwise, it is ignored.

–r[4000|**5000**|**8000**|**10000]**
: This option identifies the probable execution target of the code to be generated; it will be scheduled for optimal execution on that target. This option does not affect the ABI and/or ISA selected by the options described above (that is, it has no effect on which target processors can correctly execute the program, but just on how well they do so).

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

```
cc -64 -mips3 -r8000 ...
```

Controlling the Target Environment

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

Executing instructions speculatively can make a significant difference in the quality of generated code. What instructions can be executed speculatively depends on the trap enable state at run time.

–TENV:X=*n* Specify the level, from 0 to 5, of enabled traps that are assumed (and enforced) for purposes of performing speculative code motion. At level 0, no speculation is done. At level 1, only safe speculative motion may be done, assuming that the IEEE 754 underflow and inexact traps are disabled. At level 2, all IEEE 754 floating point traps are disabled except divide by zero. At level 3, divide by zero traps are disabled. At level 4, memory traps may be disabled or dismissed by the operating system. At level 5, traps may be disabled or ignored. The default is 1 at **–O0** through **–O2** and 2 at **–O3**.

Use nondefault levels with great care. Disabling traps eliminates a useful debugging tool, since the problems that cause them are detected later (often much later) in the execution of the program. In addition, many memory traps can't be avoided outright, but must be dismissed by the operating system after they occur. As a result, level 4 or 5 speculation can actually slow down a program significantly if it causes frequent traps.

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

–TENV:align_aggregates=*n*

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

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"

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.

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

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

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

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

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

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

Pointers in particular tend to cause aliasing problems, since it is often impossible for the compiler to identify their target objects. Therefore, you can help the compiler avoid possible aliases by introducing local variables to store the values obtained from dereferenced pointers. Indirect operations and calls affect dereferenced values, but do not affect local variables. Therefore, local variables can be kept in registers. The following example shows how the proper placement of pointers and the elimination of aliasing produces better code.

In the example below, 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 glocal 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).

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

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

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**s in 64-bit code if they may appear in mixed type expressions with **long int**s (or with **long long int**s in either 32-bit or 64-bit code). Since the hardware

automatically sign-extends the results of most 32-bit operations, this may avoid explicit zero-extension code. For example:

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

- 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.
- If your code uses **const ref**, use **–LANG:alias_const** when compiling (see “const reference Parameter Optimization With –Lang:alias_const” for more information).
- 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.

const reference Parameter Optimization With –Lang:alias_const

Consider the following example:

```
extern void pass_by_const_ref(const int& i);

int test(){
   //This requires -LANG:alias_const for performance enhancements
   int i = 10
   int j = 15
   pass_by_const_ref(i);
   pass_by_const_ref(j);
   return i + j;
}
```

In the example above, the compiler determined that the function `pass_by_const_ref` does not modify its formal parameter `i`. That is, the parameter `i` passed by const reference does not get modified in the function. Consequently the compiler can forward propagate the values of `i` and `j` to the return statement, whereas it would otherwise have to reload the values of `i` and `j` after the two calls to `pass_by_const_ref`.

Note: For this optimization to work correctly, both the caller and the callee have to be compiled with **–LANG:alias_const**.).

You can legally cast away and modify the const parameter, in the callee which is why the above option is not on by default. With this option, the compiler makes an effort to flag warnings about such cases where the callee casts away the const and modifies the parameter. For example:

```
void f(const int &x) {int *y = (int *) &x; *y = 99;}

int main() {
  int z;
  f(z); // call to f does modify z; Hence z needs to be reloaded after
the call
  return z;
}
```

With the above example, and **–LANG:alias_const**, the compiler gives a warning:

```
Compiling f__GRCi
"ex9.C", line 2 (col. 28): warning(3334): cast to type "int *" may not
be safe in presence of -LANG:alias_const. Make sure you are not
casting away const to MODIFY the parameter
```

If you specify the **mutable** keyword, then this const optimization is disabled. For example:

```
class C {
public:
  mutable int p;
  void f() const { p = 99;} //mutable data member can be modified
                            // by a const function
  int getf() const { return p;}
};

int main() {
  C c;
  c.f();        // even with -LANG:alias_const, f() can modify c.p
  return c.getf();
};
```

Using SpeedShop

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

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

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

Chapter 5

Coding for 64-Bit Programs

This chapter describes how to write/edit code for 64-bit programs.

Chapter 5

Coding for 64-Bit Programs

This chapter provides information about ways to write/update your code so that you can take advantage of the Silicon Graphics implementation of the IRIX 64-bit operating system. Specifically, this chapter describes:

- "Coding Assumptions to Avoid"
- "Guidelines for Writing Code for 64-Bit Silicon Graphics Platforms"

Also, refer to Chapter 6, "Porting Code to N32 and 64-Bit Silicon Graphics Systems," for information about compatibility, porting guidelines, and details on data types, predefined types, typedefs, memory allocation, and so forth.

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:

- "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**s to do arithmetic, from unions that implicitly identify **int**s and **pointer**s, or from passing **pointer**s as actual arguments to functions where the corresponding formal arguments are declared as **int**s. 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 **int**s and **pointer**s.

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**s are used to map fields in data structures defined externally to be 32 bits, or where **union**s attempt to identify a **long** with four **char**s.

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 signedness).

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 Silicon Graphics 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 above in "Coding Assumptions to Avoid." 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").

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 namespace 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 Silicon Graphics 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 **typedef**ed differently among the targets you must support, you may need to define constant format strings for each of the types defined in step 1. For example:

```
#define _fmt32 "%d"
#define _fmt32u "%u"
#define _fmt64 "%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);
}
```

Chapter 6

Porting Code to N32 and 64-Bit Silicon Graphics Systems

This chapter provides guidelines for porting code.

Chapter 6

Porting Code to N32 and 64-Bit Silicon Graphics Systems

This section 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.

Specifically, this chapter discusses the following topics:

- "Compatibility," which describes compatibility between 32, n32, and 64-bit programs.
- "N32 Porting Guidelines," which explains guidelines for porting high-level languages.
- "Porting Code to 64-Bit Silicon Graphics Systems," which describes data types, typedefs, maximum memory allocation, and use of large files on XFS.

This chapter uses the following terminology:

o32
: The current 32-bit ABI generated by the ucode compiler, that is, 32-bit compilers prior to IRIX 6.1 operating system.

n32
: The new 32-bit ABI generated by the MIPSpro 64-bit compiler (for a list of n32 features, see Chapter 1).For information about the n32 ABI, see *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 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.

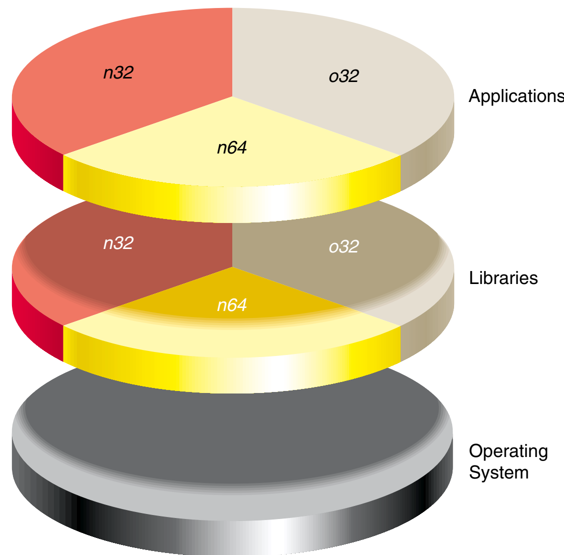

Figure 6-1 Application Support Under Different ABIs

Figure 6-2 illustrates the library locations for different ABIs.

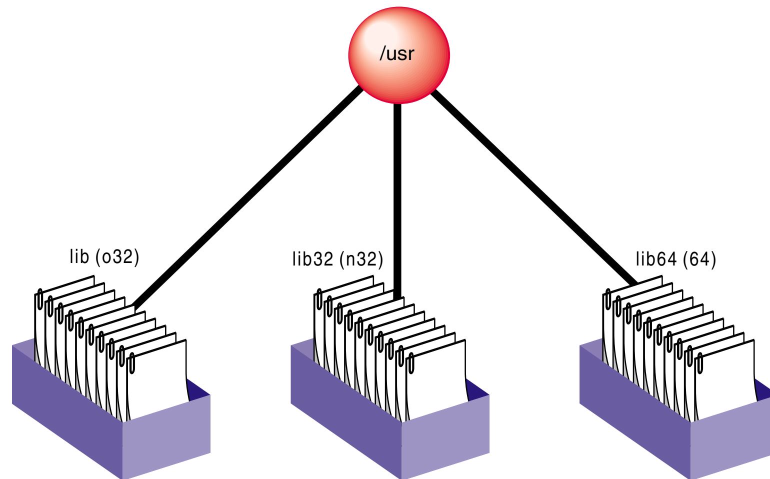

Figure 6-2 Library Locations for Different ABIs

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”)
- Identifying and making the necessary source code changes (see “Source Code Changes”)
- Rebuilding the application for the target machine (see “Build Procedure”)
- Analyzing and debugging runtime issues (see “Runtime Issues”)

Each of these tasks is described below. 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 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).

Runtime 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, R1000 (madd instructions round slightly differently than a multiply instruction followed by an add instruction).

For more information refer to Chapter 5 of the *MIPSpro 64-bit Porting and Transition Guide*.

Porting Code to 64-Bit Silicon Graphics Systems

This section covers porting code to 64-bit Silicon Graphics 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 Silicon Graphics systems in the *MIPSpro Application 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
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
–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**s and **long**s and 32-bit **int**s. 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 **typedef**s 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 **typedef**s are enabled by compiler-predefined macros (listed in Table 6-2), 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. **Typedef**s 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 **typedef**s 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;
```

intmax_t and **uintmax_t** 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 Gb, or 2,048 Mb).

Previous implementations of Fortran, C, and C++ limited the total program size, as well as the size of any single array, to 2 GB. 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. (Note: 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.

```
$cat a2.c

#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 vmemoryuse variables set to values large enough to support the sizes of the arrays (see sh(1) reference page).

The following example compiles and runs the above code after setting the stacksize 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 XFS. If a program is doing sequential I/O and does not maintain internal byte counters, files greater than 2 Gb won't encounter problems.

However, if an application uses internal byte counters, then modifications are required.Table 6-3 lists potential problems and modifications required to enable files greater than 2 Gb 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 doesn't read the raw disk)

For more information about XFS, see *Getting Started With XFS Filesystems*.

Index

Numbers

32-bit mode, 10, 170
  Also see n32
  compatibility, 164
  data types, 168
  definition, 163
  libraries, 164
  n32 definition, 163
  overflow, 157
  porting to n32, 165
–*32* option, 12, 25
4.3 BSD extensions, 29
64-bit mode, 10, 155-159
  data types, 168
  libraries, 164
  unsigned integers, 155
–*64* option, 12, 25

A

ABI
  options, 143
abi options, 10
ABI specification, 10
address aliases, 95-96
addresses, optimization, 147
address space, 79
alias analysis, 95-96
aliasing
  and pointer placement, 146
  memory, 127
  optimization, 146
align/fill pragmas, 117-119
analysis, dependence, 120-121
analyzer, parallel, 4
–*ansi* option, 25
a.out files, 30
architecture
  instruction set, 143
  optimizing programs, 143
  options, 143
archive libraries, 59
archiver. *See ar* command
ar command, 52-55
  command syntax, 53
  options, 53
argument registers, 170
arguments
  store, 170
arrays
  2 gigabyte, 171
as assembler, 33
assembly language programs
  porting to n32, 166
assembly language programs, linking, 33

B

back substitution, 137
bit masks, 159
BLOCK DATA, 75
blocking and permutation transformations, 113-116
  controlling, 109
block padding, 93-95
  restrictions, 93
branch elimination, 133
BSD 4.3 extensions, 29
–Bsymbolic, compiling, 74
build procedure
  n32, 166
byte counters
  and file size, 173

C

C++
  language definitions, 17
  precompiled headers, 17
C++ programs
  optimization, 145-149
cache
  conflicts and padding, 93
  misses, 104
cache optimization
  –LNO option, 109
cache parameters
  controlling with LNO, 107
CC compiler. *See* drivers
–cckr option, 26
char, 168
C language
  floating point, 121
  precompiled headers, 17
–clist option, 96
code
  arithmetic, 157
  assumptions, 155
  conversion, 132
  executed at startup, 129
  hints, 155
  overflow 32 bits, 157
  portable, 158
  porting to 64-bit system, 167
  porting to n32-bit systems, 163
  rarely executed, 129
  shifts, 157
  signed ints, 148
  sizeof(int)==sizeof(long), 156
  sizeof(int)==sizeof(void*), 156
  sizeof(long)==4, 156
  sizeof(void*)==4, 156
  transformation, 132
  typedefs, 170
  view transformations, 96
  writing for 64-bit applications, 155-159
  zero-extension, 148
code generator, 130-142
  Also see optimizing programs
  and optimization levels, 131, 132, 140
  back substitution, 137
  branch elimination, 133
  cross-iteration optimization, 134-135
    read-read elimination, 134
    read-write elimination, 134
    sub-expression elimination, 135
    write-write elimination, 135
  feedback, 142
  frequency of execution, 142
  if conversion, 132
  if conversion and floating points, 133
  instruction-level parallelism, 133
  latency, 142
  loop unrolling, 135, 141
  memory exceptions, 133

modify default, 141
–*O0* option, 131
–*O1* option, 131
–*O2* option, 132-140
–*O3* option, 132-140
prefetch, 142
R10000 optimization, 133
recurrence breaking, 136
software pipelining, 137, 141
steps at –*O2* and –*O3*, 139

COFF, 13

common block padding, 93-95
restrictions, 93

Common Object File Format, 13

COMMON symbols, 75

COMPILER_DEFAULTS_PATH environment variable, 10

compiler back end, 4

compiler.defaults file, 10

compiler drivers. *See* drivers, 4

compiler front end, 4

compiler options. *See* drivers

compiler system
32-bit mode, 12
64-bit mode, 12
components, 4
macros, 169
n32-bit mode, 12
overview, 4
predefined types, 169

compiling with –*Bsymbolic*, 74

constant format strings, 159

constants, 157
negative values, 159

conventions, syntax, xviii

conversion of code, 132

–*C* option, 25

–*c* option, 5, 25

copt optimizer, 4

–*cord* option, 26

counters, internal byte, 173

cpp preprocessor, 4

C programs
optimization, 145-149

cross-file inlining, 92

cross-iteration optimization, 134-135
read-read elimination, 134
read-write elimination, 134
sub-expression elimination, 135
write-write elimination, 135

D

–D__EXTENSIONS__ option, 29

–D_MIPS_FPSET, 169

–D_MIPS_ISA, 169

–D_MIPS_SIM, 169

–D_MIPS_SZINT, 169

–D_MIPS_SZLONG, 169

–D_MIPS_SZPTR, 169

data
prefetching, 104

data alignment
optimizing, 144

data type
signed, 148

data types
sizes, 168

debugging
driver options, 38
floating points, 126

defaults
compilation modes, 10
specification file, 10

dependence analysis, 111, 120-121

directives
  DSM, 27
  LNO, 111
  multiprocessing, 27
disabling traps, 144
disassemble object file, 39
dis command, 39, 40
  command syntax, 40
  options, 40
dlclose(), 79
dlerror(), 79
dlopen(), 77
dlsym(), 78
–Dname option, 26
double, 168
drivers
  as assembler, 33
  bypassing, 4
  CC compiler, 4
  cc compiler, 4
  –c option, 5
  defaults, 10, 24
  f77/90 compiler, 4
  fec preprocessor, 4
  file name suffixes, 15
  input file suffixes, 15
  –KPIC, 14
  linking, 5
  –non_shared, 14
  omit linking, 5
  optimizing programs, 121
  options, 25, 38
  –show option, 5
  stages of compilation, 5
DSOs, 3, 13, 14, 59-82
  building new DSOs, 69
  converting libraries, 76
  creating DSOs, 69
  dlclose(), 79
  dlerror(), 79
  dlopen(), 77
  dlsym(), 78
  dynamic loading diagnostics, 79
  exporting symbols, 71
  guidelines, 63
  hiding symbols, 71
  libraries, shared, 63
  linking, 35
  loading dynamically, 77
  mmap() system call, 79
  munmap() system call, 79
  naming conventions, 69
  QuickStart, 66-69
  search path, 72
  sgidladd(), 77
  shared libraries, 63
  starting quickly, 66
  unloading dynamically, 79
  versioning, 79
dump command. *See elfdump*
dwarfdump command, 39, 41
  options, 42
DWARF symbolic information, 41
dynamic linking, 3, 13, 77
Dynamic Shared Objects. *See* DSOs

E

elfdump command, 39, 43
  command syntax, 43
  options, 43
Elf object file, 41
ELF. *See* executable and linking format
elimination
  branches, 133
  read-read, 134
  read-write, 134
  sub-expression, 135
  write-write, 135

–elspec option, 31
environment
  optimizing programs, 143
environment variable
  COMPILER_DEFAULTS_PATH, 10
environment variables
  32-bit compilation, 12
  64-bit compilation, 12
  n32-bit compilation, 12
–E option, 26
executable and linking format, 3, 13
executable files, 13
execution
  controlling, 129
exporting symbols, 71
expressions
  optimizing, 124
extension
  sign, 157
  zero, 157

F

f77/90 compiler, 4
fecc preprocessor, 4
fec preprocessor, 4
  bypassing, 4
feedback
  and code generator, 142
–feedback option, 26
fef77/90p analyzer, 4
fef77/90 preprocessor, 4
file command, 39, 45
  command syntax, 45
  example, 45
  options, 45
file inlining, 88-96
files
  2 gigabyte size, 173
  compilation specification, 10
  executable, 13
  header, 16
  include, 16
  internal byte counters, 173
  listing properties, 39
  naming conventions, 15
  precompiled header, 17
  relocatable, 13
  size, 173
file type, determining, 45
fill/align pragmas, 117-119
fission
  controlling, 106
  LNO, 112
  loops, 102
–flist option, 96
float, 168
float.h include file, 168
floating points
  debugging, 126
  if conversion, 133
  optimization, 121-126
  optimizing, 125
  reassociation, 125
Force, 125
format
  object file, 3, 13
Fortran
  floating point, 121
  padding global arrays, 93
  program optimization, 131
Fortran programs
  optimization, 145-149
–fullwarn option, 26
functions
  implicitly declared, 157

fusion
  controlling, 106
  LNO, 112
  loop, 101

G

gather-scatter, 104
  controlling, 107
global arrays
  padding, 93
global offset table, 14
global optimizer, 145-148
–*G* option, 26
–*g* option, 26, 38
GOT, 14
guidelines
  porting, 165

H

header files, 16-24
  multiple languages, 17
  portable, 158
  precompiled, 17
  specification, 16
–*help* option, 26
high-order bit, 157

I

–*Idirname* option, 27
IEEE
  floating points, 123
  optimization, 123
if conversion, 132
if-then-else statements
  optimization, 147
implicitly declared function, 157
include files, 16
  float.h, 168
  inttypes.h, 170
  limits.h, 168
  multiple languages, 17
  n32, 166
indirect
  calls, using, 145
–*INLINE*, 91-93
  all option, 91
  file option, 92
  must option, 92
  never option, 92
  none option, 91
inliner
  standalone, 92
inlining, 88-96
  benefits, 90
input file names, 15
instruction
  mips4 rsqrt, 126
  prefetching, 104
instruction-level parallelism, 133
int, 156, 168, 170
integer
  overflow, 129
  scaling, 158
integers
  64-bit registers, 155
interleaving
  reduction, 136
internal byte counters
  and file size, 173
inttypes.h include file, 170

–IPA, 93-96
  addressing=ON option, 96
  alias=ON option, 95
  forcedepth option, 93
  maxdepth option, 93
  Olimit option, 93
  plimit option, 93
  space option, 93
ISA
  options, 143
ISA specification, 10

K

–KPIC option, 14, 27

L

latency
  and code generator, 142
  network, improving performance
    latency problems, 32
ld
  and assembly language programs, 33
  command syntax, 31
  dynamic linking, 3, 13
  example, 32
  libraries, default search path, 34
  libraries, specifying, 33
  link editor, 4
  multilanguage programs, 35
  options, 31
  –shared option, 69
LD_BIND_NOW, 74
–ld option
  –read option, 32
libdl, 77
libraries
  archive, 59
  global data, 65
  header files, 16
  libdl, 77
  locality, 65
  non-shared, converting to DSOs, 76
  paging, 65
  path, 11
  routines to exclude, 63
  routines to include, 63
  self-contained, 63
  shared, 3, 13
  shared, static, 14, 59
  specifying, 33
  static data, 63
  tuning, 65
lib.so functions
  optimization, 148
limits.h include file, 168
linking
  dynamic. *See ld*
  omit, 5
linking. *See ld*
LNO. *See* optimizing programs, *–LNO* option
loader
  runtime. *See rld*
loading
  symbols, 71
local variables
  optimization, 145
long, 168, 170
long double, 168
long long, 168
loop interchange, 99-100
loop-nest optimization. *See* optimizing programs,
    –LNO option

loops
  fission, 102
  fusion, 101
  interchanging, 99
  optimizing, 106
  parallel, 104
loop unrolling
  code generator, 135

M

machine instructions, 39
macro preprocessors, 4
macros
  NARGSAVE, 170
  predefined, 169
  typedefs, 170
makefiles, 166
maximum integer type, 171
memory
  2 gigabyte arrays, 171
  referencing, 127, 144
memory allocation
  arrays, 171
memory exceptions
  if conversion, 133
–mips2 option, 27
–mips3 option, 27
–mips4 option, 27
mips4 rsqrt instruction, 126
MIPS Instruction Set Architecture, 169
mmap() system call, 79
mode
  32-bit, 10
  64-bit, 10
  n32-bit, 10
-mp option, 27
multilanguage programs
  and *ld*, 35
  header files, 17
multiprocessing
  enabling, 27
munmap() system call, 79

N

n32, 166
  assembly language programs, 166
  build procedure, 166
  include files, 166
  libraries, 164, 166
  porting environment, 166
  porting guidelines, 165
  runtime issues, 167
  source code changes, 166
n32-bit mode, 10
–n32 option, 12, 25
naming source files, 15
NARGSAVE macro, 170
negative values
  problems, 159
nm command, 39, 46-49
  character codes, 47
  command syntax, 46
  example, 48
  example of undefined symbol, 38
  options, 46
  undefined symbol, 38
–nocpp option, 28
–non_shared option, 28
–nostdinc option, 28

O

object file information
- disassemble, 39
- format, 3, 13
- listing file properties, 39
- listing section sizes, 39, 49
- symbol table information, 39, 46
- tools, 39
- using, 39
- using *dwarfdump*, 39
- using *elfdump*, 39, 43

–*o filename* option, 28

–*Onum* option, 28

operating system
- 64 bit, 155-159

operations
- relational, 129
- unsigned relational, 129

optimization, 85-??
- addresses, 147
- *Also see* optimizing programs
- C++ programs, 145-149
- C programs, 145-149
- Fortran, 145-149
- function return values, 145
- global, 145-148
- if-then-else statements, 147
- *libc.so* functions, 148
- –*O0* compiler option, 87
- –*O1* compiler option, 87
- –*O2* compiler option, 87
- –*O3* compiler option, 87
- options, 87
- pointer placement, 146
- signed data types, 148
- STDARG, 147
- *stdarg.h*, 147
- switch statements, 147
- tables, 147
- tips for improving, 145
- unions, 145
- value parameters, 145
- VARARG, 147
- *varargs.h*, 147
- variables, global vs. local, 145

optimizer, 4
- *copt* optimizer, 4

optimizing programs
- –*32* option, 143
- –*64* option, 143
- alias analysis, 95-96
- –*align* option, 144
- *Also see* code generator
- benefits, 86
- cache, 104
- code generator, 130-142
  - overview, 130
- common block padding, 93-95
  - restrictions, 93
- data alignment, 144
- debugging, 86
- dependence analysis, 120-121
- execution frequency, 129
- fill/align pragmas, 117-119
- floating points, 121-126
- Fortran optimization, 131
- IEEE floating points, 123
- ignoring pragmas, 106
- –*INLINE* option, 91-93
- inlining benefits, 90
- interprocedural analysis, 88-96
- –*IPA* option, 93, 95
- –*LNO* option, 96-121
  - blocking, 100-101
  - blocking and permutation transformations, ??-110
  - cache optimization, 109
  - code transformation, 96
  - controlling cache parameters, 107
  - controlling dependence analysis, 111

controlling fission and fusion, 106
controlling gather-scatter, 107
controlling illegal transformations, 110
controlling prefetch, 110
controlling transformations, 109
directives, 111-121
fission, 112
fusion, 112
gather-scatter, 104-105
loop fission, 102-103
loop fusion, 101-102
loop interchange, 99-100
optimization levels, 106
outter loop unrolling, 100-101
pragmas, 111-121
prefetching, 104
running LNO, 96
–*mips* option, 143
–*n32* option, 143
–*OPT* option, 122-129
alias=any option, 127
alias=disjoint option, 128
alias=name option, 127
alias=restrict option, 128
alias=typed option, 128
alias=unnamed option, 128
div_split option, 124
fast_complex option, 124
fast_exp option, 125
fast_io option, 125
fast_sqrt option, 125
fold_reassociate option, 125
fold_unsafe_relops, 129
fold_unsigned_relops, 129
IEEE_arithmetic option, 123
IEEE option, 121
recip option, 126
roundoff option, 121, 122
rsqrt option, 126
pragmas, ignore, 106
pragmas, mips_frequency_hint, 129
prefetch pragmas, 116-117
shared code, 144
target architecture, 143
target architecture options, 143
target environment, 143
–*TARG* option
isa=mips option, 143
madd option, 126, 143
–*TENV* option, 143-144
align_aggregates option, 144
X option, 144
transformation pragmas, 113-116
transformations, 122
–*OPT* option, 28
div_split option, 124
fold_reassociate option, 125
fold_unsafe_relops, 129
fold_unsigned_relops option, 129
overflow
integer, 129
integers, 129
overflow of code, 157

P

padding, blocks, 93-95
restrictions, 93
page size, 65
paging
alignment, 65
parallel analyzer, 4
parallel loops, 104
parameters
optimization, 145
pca analyzer, 4
pc compiler. *See* drivers
–*pch* option, 28
PIC. *See* position-independent code

pixie
  and SpeedShop, 151
pointer, 156, 168, 170
pointer placement
  and aliasing, 146
  example, 146
pointers
  referencing memory, 127
–*P* option, 28
–*p* option, 28
porting code, 167
porting guidelines, 165
position-independent code, 3, 13, 14, 69
pragmas
  ignore, 106
  LNO, 111
  mips_frequency_hint, 129
precompiled header files, 17-23
  automatic, 18
  controlling, 22
  deletion, 22
  performance, 23
  requirements, 19
  reuse, 20
prefetch
  and code generator, 142
  controlling, 110
prefetching instructions, 104
prefetch pragmas, 116-117
preprocessing, 4
preprocessors
  macro, 4
printf command, 159
problems, 156
  constants, 157
  floating points, 126
  implicitly declared functions, 157
  negative values, 159
  porting code, 155
  printf, 159
  scanf, 159
  sizeof(int)==sizeof(long), 156
  sizeof(int)==sizeof(void*), 156
  sizeof(long)==4, 156
  solving, 158
  types, 155
processor specification, 10
proc options, 11
prof
  and SpeedShop, 151

Q

QuickStart DSOs. *See* DSOs, QuickStart

R

–*r10000* option, 143
–*r5000* option, 143
–*r8000* option, 143
–*read* option, 32
read-read elimination, 134
read-write elimination, 134
recurrence breaking
  back substitution, 137
  code generator, 136
  reduction interleaving, 136
reduction interleaving, 136
registers
  64-bit, 155
  argument, 170
  blocking, 100
  temp, 170
relational operations
  unsigned, 129

relational operators
  integer overflow, 129
relocatable files, 13
relocation bits, removing, 39
remove
  relocation bits, 39
  symbol table, 39
resolve text symbols, 74
return values, optimization, 145
rld, 61
  dynamic linking, 77
  libdl, 77
  search path, 72
roundoff
  floating points, 122
  optimization, 122
rsqrt instruction, 126
RTLD_GLOBAL, 78
RTLD_LAZY, 78
RTLD_NOW, 78
runtime issues
  n32, 167
runtime linker. *See rld*

S

scalar optimizer, *copt*, 4
scalar variables
  word size, 148
scanf function, 159
search path
  rld, 72
selecting
  compilation mode, 10
  instruction set, 10
  ISA, 10
  processor, 10
sgidladd(), 77
shared code
  optimizing, 144
shared libraries, static, 59
shared library, 3, 13
shared objects, dynamic, 59
short, 168
–show_defaults option, 11
–show option, 5, 29
sign bit set, 157
signed data type
  optimization, 148
signed ints
  64-bit code, 148
  64-bit registers, 155
sign extension, 156, 157
size command, 39, 49, 49-51
  command syntax, 49
  example, 51
sizeof(int)==sizeof(long), 156
sizeof(int)==sizeof(void*), 156
sizeof(long)==4, 156
sizeof(void*)==4, 156
size of object file, 39
software pipelining
  and code generator, 137
–S option, 29
source code
  n32, 166
source file names, 15
specifying compilation mode, 10
SpeedShop, 151
  pixie command, 151
  prof command, 151
  ssrun command, 151
standalone inliner, 92
stdarg.h, 147

STDARG. *See* optimization
stdio.h header file, 16
storing arguments, 170
strings
  printf, 159
  scanf, 159
strip command, 39, 51
  command options, 51
  command syntax, 51
sub-expression elimination, 135
suffixes
  input files, 15
switch statements
  optimization, 147
symbol resolution, 74
symbols
  exporting, 71
  fill, align, 117
  loading, 71
symbol table
  data, 39
  dumping data, 46
  get listing, 48
  removing, 39
syntax, conventions, xviii

T

–TARG option, 29
temp registers, 170
–TENV option, 29
transformation
  of code, 132
transformation pragmas, 113-116
transformations
  controlling illegal, 110
  controlling with LNO, 109
  view code, 96
traps
  disable, 144
troubleshooting
  constants, 157
  implicitly declared functions, 157
  negative values, 159
  printf, 159
  scanf, 159
  sizeof(int)==sizeof(long), 156
  sizeof(int)==sizeof(void*), 156
  sizeof(long)==4, 156
  sizeof(void*)==4, 156
  solving problems, 158
truncation of code, 157
type, determining for files, 45
typedefs, 159, 170
types
  assumptions, 155
  change in size, 157
  char, 168
  constants, 157
  double, 168
  float, 168
  int, 156, 168, 170
  largest integer type, 171
  long, 168, 170
  long double, 168
  long long, 168
  pointer, 156, 168, 170
  problems, 155
  scaling integer, 158
  short, 168
  sizes, 168
typographical conventions, xviii

U

–*Uname* option, 29
unions
  optimization, 145
unrolling loops, 100, 115, 135, 141
unsigned ints
  32-bit, 155
unsigned relational operations, 129

V

VARARG. *See* optimization
varargs.h, 147
variables
  scalar, 148
virtual address space, 79

W

–*woff* option, 29
word-size scalar variables, 148
write-write elimination, 135

X

–*xansi* option, 29
XFS
  file size, 173

Z

zero extension, 157
zero-extension code, 148

Tell Us About This Manual

As a user of Silicon Graphics products, you can help us to better understand your needs and to improve the quality of our documentation.

Any information that you provide will be useful. Here is a list of suggested topics:

- General impression of the document
- Omission of material that you expected to find
- Technical errors
- Relevance of the material to the job you had to do
- Quality of the printing and binding

Please send the title and part number of the document with your comments. The part number for this document is 007-2360-007.

Thank you!

Three Ways to Reach Us

- To send your comments by **electronic mail**, use either of these addresses:
  - On the Internet: techpubs@sgi.com
  - For UUCP mail (through any backbone site): *[your_site]*!sgi!techpubs
- To **fax** your comments (or annotated copies of manual pages), use this fax number: 650-932-0801
- To send your comments by **traditional mail**, use this address:

  Technical Publications
  Silicon Graphics, Inc.
  2011 North Shoreline Boulevard, M/S 535
  Mountain View, California 94043-1389