

MIPS Compiling and Performance Tuning Guide

Document Number 007-2479-001

CONTRIBUTORS

Written by Arthur Evans, Wendy Ferguson, Jed Hartman, and Jackie Neider

Edited by Christina Cary

Production by Laura Cooper and Lorrie Williams

Engineering contributions by Dave Anderson, Dave Babcock, Jack Carter, Wei-Chau Chang, Julia Chow, Jay Gischer, W. Wilson Ho, Bill Mannell, Bron Nelson, Andy Palay, John Wilkinson

Cover design and illustration by Rob Aguilar, Rikk Carey, Dean Hodgkinson, Erik Lindholm, and Kay Maitz

© Copyright 1994, 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 Unpublished rights reserved under the Copyright Laws of the United States.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 94043-1389.

Silicon Graphics and IRIS are registered trademarks and IRIX, CASEVision, IRIS IM, IRIS Showcase, Impresario, Indigo Magic, Inventor, IRIS-4D, POWER Series, RealityEngine, CHALLENGE, Onyx, and WorkShop are trademarks of Silicon Graphics, Inc. UNIX is a registered trademark of UNIX System Laboratories. OSF/Motif is a trademark of Open Software Foundation, Inc. The X Window System is a trademark of the Massachusetts Institute of Technology. Ada is a registered trademark of Ada Joint Program Office, U.S. Government. Post-It is a registered trademark of Minnesota Mining and Manufacturing. PostScript is a registered trademark and Display PostScript is a trademark of Adobe Systems, Inc. NFS is a trademark of Sun Microsystems, Inc. Speedo is a trademark of Bitstream, Inc.

MIPS Compiling and Performance Tuning Guide
Document Number 007-2479-001

Contents

List of Figures ix

List of Tables xi

About This Guide xiii

What This Guide Contains xiii

What You Should Know Before Reading This Guide xiv

Suggestions for Further Reading xiv

Conventions Used in This Guide xv

1. About the Compiler System 1

2. Using the Compiler System 9

Object File Format and Dynamic Linking 10

 Executable and Linking Format 10

 Dynamic Shared Objects 11

 Position-Independent Code 11

Source File Considerations 12

 Source File Naming Conventions 12

 Header Files 13

 Specifying a Header File 14

 Creating a Header File for Multiple Languages 14

Compiler Drivers 15

 Default Behavior for Compiler Drivers 16

 General Options for Compiler Drivers 16

- Linking 19
 - Invoking the Linker Manually 20
 - Linker Syntax 21
 - Linker Example 22
 - Linking Assembly Language Programs 22
 - Linking Libraries 23
 - Specifying Libraries and DSOs 23
 - Examples of Linking DSOs 25
 - Linking to Dynamic Shared Objects 25
 - Linking Multilanguage Programs 26
 - Finding an Unresolved Symbol With *ld* 27
- Debugging 28

Getting Information About Object Files	28
Disassembling Object Files with <i>dis</i>	29
<i>dis</i> Syntax	29
<i>dis</i> Options	30
Listing Selected Parts of Object Files and Libraries With <i>elfdump</i>	31
<i>elfdump</i> Syntax	31
<i>elfump</i> Options	31
Determining File Type With <i>file</i>	33
<i>file</i> Syntax	33
<i>file</i> Example	33
Listing Symbol Table Information: <i>nm</i>	33
<i>nm</i> Syntax	34
<i>nm</i> Options	34
<i>nm</i> Example	36
Finding a Symbol in an Unknown Library	37
Listing Selected Parts of COFF Files With <i>odump</i>	38
<i>odump</i> Syntax	38
Determining Section Sizes With <i>size</i>	40
<i>size</i> Syntax	40
<i>size</i> options	40
<i>size</i> Example	41
Removing Symbol Table and Relocation Bits with <i>strip</i>	41
<i>strip</i> Syntax	41
<i>strip</i> Options	41
Using the Archiver to Create Libraries	42
<i>ar</i> Syntax	43
Archiver Options	43
<i>ar</i> Examples	46
3. Dynamic Shared Objects	49
Benefits of Using DSOs	50

- Using DSOs 51
 - DSOs vs. Archive Libraries 51
 - Using QuickStart 52
 - Guidelines for Using Shared Libraries 53
 - Choosing Library Members 53
 - Tuning Shared Library Code 54
- Taking Advantage of QuickStart 56
- Building DSOs 58
 - Creating DSOs 59
 - Making DSOs Self-Contained 59
 - Controlling Symbols to be Exported or Loaded 60
 - Using DSOs With C++ 61
 - Using Registry Files 61
 - Registry File Format 62
 - Directive Lines 62
 - Shared Object Specification Lines 63
- Runtime Linking 64
 - Searching for DSOs at Runtime 64
 - Runtime Symbol Resolution 65
 - Compiling with `-Bsymbolic` 66
 - Converting Libraries to DSOs 67
- Dynamic Loading Under Program Control 69
- Versioning of DSOs 71
 - The Versioning Mechanism of Silicon Graphics 71
 - What Is a Version? 71
- 4. Using the Performance Tools 77**
 - Overview of Profiling 78
 - Profiling With *prof* 78
 - Running the Profiler 79
 - prof* Syntax 79
 - prof* Defaults 79
 - prof* Options 80

pc Sampling	81
Obtaining pc Sampling	82
Creating Multiple Profile Data Files	83
pc Sampling Frequency	84
Examples Using prof to Obtain pc Sampling	85
Example Using prof -pcsample	85
Example Using prof -pixie -dis	86
Basic Block Counting	88
Using pixie	89
<i>pixie</i> Syntax	89
<i>pixie</i> Options	89
Obtaining Basic Block Counts	90
Examples of Basic Block Counting	93
Example Using prof -pixie -invocations	94
Example Using prof -pixie -heavy	95
Example Using prof -pixie -lines	96
Example Using prof -pixie -quit	97
Example Using prof -pixie -procedures	98
Example Using prof -pixie -procedures -clock	99
Summing Basic Block Count Results	100
Using <i>pixstats</i>	100
<i>pixstats</i> Syntax	101
<i>pixstats</i> Options	101
Examples Using <i>pixstats</i>	103
Profiling Multiprocessed Executables	106
Rearranging Procedures With <i>cord</i>	107
<i>cord</i> Syntax	107
<i>cord</i> Options	108
Example Using <i>cord</i>	108

- 5. **Optimizing Program Performance** 113
 - Optimization Overview 113
 - Global Optimizer 114
 - Benefits of Optimization 114
 - Optimization and Debugging 114
 - Using the Optimization Options 115
 - Compiler Optimization Options 115
 - Examples of Full Optimization 117
 - Loop Optimization 119
 - Unoptimized Code 120
 - Optimized Code 121
 - Loop Unrolling 121
 - Optimizing Separate Compilation Units 122
 - Optimizing Frequently Used Modules 122
 - Ucode Object Libraries 125
 - Building Ucode Object Libraries 125
 - Using Ucode Object Libraries 126
 - Improving Global Optimization 126
 - Optimizing C and Fortran Programs 127
 - C Programs Only 127
 - Example of Pointer Placement and Aliasing 128
 - Ada® Programs 130
 - Improving Other Optimization 130
 - C and Fortran Programs 130
 - C Programs Only 131
 - Register Allocation 132
- A. **Position-Independent Coding in Assembly Language** 135
 - Examples 137
- Index** 141

List of Figures

Figure 1-1	Compiler System Flowchart	5
Figure 2-1	Compilation Control Flow for Multilanguage Programs	27
Figure 3-1	An Application Linked with DSOs	57
Figure 4-1	How pc Sampling Works	83
Figure 4-2	How Basic Block Counting Works	93
Figure 4-3	How <i>cord</i> Works	109
Figure 5-1	Optimization Phases of the Compiler	117
Figure 5-2	Compiling with the <code>-j</code> Option	118
Figure 5-3	Executing Full Optimization	119
Figure 5-4	Optimization Process	124

List of Tables

Table In-1	Suggestions for Further Reading	xiv
Table 1-1	Compiler System Functional Components	1
Table 1-2	Compilers and Default Libraries	4
Table 2-1	Driver Input File Suffixes	12
Table 2-2	General Driver Options	17
Table 2-3	Linker Options	21
Table 2-4	Driver Options for Debugging	28
Table 2-5	<i>dis</i> Options	30
Table 2-6	<i>elfdump</i> Options	31
Table 2-7	Symbol Table Dump Options	34
Table 2-8	Character Code Meanings	35
Table 2-9	<i>odump</i> Options	38
Table 2-10	<i>size</i> Options	40
Table 2-11	<i>strip</i> Options	41
Table 2-12	Archiver Options	43
Table 2-13	Archiver Modifiers	44
Table 2-14	Archiver Suboptions	45
Table 3-1	<i>libdl</i> functions	69
Table 4-1	Options for <i>prof</i>	80
Table 4-2	Setting a PROFDIR Environment Variable	84
Table 4-3	Options for <i>pixie</i>	89
Table 4-4	Options for <i>pixstats</i>	101
Table 4-5	Options for <i>cord</i>	108
Table 5-1	Optimization Options	115

About This Guide

This guide discusses a variety of issues and tools involved in programming under the IRIX™ operating system. It describes the components of the compiler system, other programming tools, and dynamic shared objects. It also explains ways to improve program performance.

What This Guide Contains

This guide consists of the following chapters:

- Chapter 1, “About the Compiler System,” provides a brief overview of the compiler system.
- Chapter 2, “Using the Compiler System,” describes the components and related tools of the compiler system and explains how to use them.
- Chapter 3, “Dynamic Shared Objects,” explains how to build and use dynamic shared objects, which replace the static shared libraries used by previous versions of IRIX.
- Chapter 4, “Using the Performance Tools,” describes how to use the *prof*, *pixie*, and *cord* commands.
- Chapter 5, “Optimizing Program Performance,” covers how to reduce program execution time by using optimization options and techniques.
- Appendix A, “Position-Independent Coding in Assembly Language,” describes assembly language coding techniques required by this version of IRIX.

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 under the IRIX operating system. We assume you are familiar with the IRIX (or UNIX®) operating system and a programming language such as C. This guide does not explain how to write or compile programs.

Suggestions for Further Reading

In addition to this guide, which describes general compilation issues for MIPS compilers, refer to Table In-1 for a list of other Silicon Graphics manuals you can consult for information about IRIX programming and languages.

Table In-1 Suggestions for Further Reading

Topic	Document
IRIX programming	Programming on Silicon Graphics Systems: An Overview Topics in IRIX Programming
Assembly language	MIPSpro Assembly Language Programmer's Guide
C language	C Language Reference Manual
C++ language	C++ Programming Guide
Fortran77 language	Fortran77 Programmer's Guide
Pascal language	Pascal Programming Guide
Real-time programming	REACT/Pro Release Notes

You can order a printed manual from Silicon Graphics by calling SGI Direct at 1-800-800-SGI1 (800-7441). Outside the U.S. and Canada, contact your local sales office or distributor.

Silicon Graphics also provides manuals online. 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 may also want to find out more about standard UNIX topics. For UNIX information, consult a computer bookstore or one of the following:

- AT&T. *UNIX System V Release 4 Programmer's Guide*. 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

IRIX executes all binaries that are compliant with the SVR4 ABI, as specified in the *System V Applications Binary Interface—Revised Edition* and the *System V ABI MIPS Processor Supplement*. Consult these manuals for details.

Conventions Used in This Guide

This guide uses these conventions and symbols:

<code>Courier</code>	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.
<i>italics</i>	Words in italics represent characters or numerical values that you define. Replace the abbreviation with the defined value. Also, italics are used for manual page names and commands. The section number, in parentheses, follows the name.
[]	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 the Fortran keyword DIMENSION must be typed as shown, that the user-defined entity *a*(*d*) is required, and that one or more of *a*(*d*) can be specified. The parentheses () enclosing *d* are required.

```
{STATIC | AUTOMATIC} v [, v] ...
```

indicates that either the STATIC or AUTOMATIC keyword must be typed as shown, that the user-defined entity *v* is required, and that one or more *v* items can be specified.

Chapter 1

About the Compiler System

This chapter provides a brief description of the compiler system and its components.

About the Compiler System

The IRIS-4D compiler system consists of a set of components that enable you to create executable modules from programs written in languages such as C, C++, Fortran 77, and Pascal.

The 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 linking 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, “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 IRIS-4D compiler system components and the task each performs.

Table 1-1 Compiler System Functional Components

Tool	Task	Examples
Text editor	Write and edit programs	<i>vi, jot, emacs</i>
Compiler driver	Compile and link programs	<i>cc, f77, pc, as</i>
Object file analyzer	Analyze object files	<i>elfdump, file, nm, odump, size, strip</i>
Profiler	Analyze program performance	<i>prof, pixie</i>

Table 1-1 (continued) Compiler System Functional Components

Tool	Task	Examples
Procedure rearranger	Minimize paging/maximize instruction cache hit rate	<i>cord</i>
Optimizer	Improve program performance	<i>uopt</i>
Archiver	Produce object-file libraries	<i>ar</i>
Run-time linker	Link Dynamic Shared Objects at run time	<i>rld</i>
Debugger	Debug programs	<i>dbx</i>

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

- Macro preprocessor (*cfe*, *cpp*, *acpp*)
- Parallel analyzer (*pca*, *pfa*)
- Scalar optimizer (*copt*)
- Compiler front end (*cfe*, *fcom*, *upas*, *accom_mp*, *ccom_mp*)
- Ucode tools (*ujoin*, *uld*, *umerge*)
- Optimizer (*uopt*)
- Code generator (*ugen*)
- Assembler (*as*)
- Linker (*ld*)

Note: C++ has a specialized driver, *CC*, with slightly different options from *cc*, *f77*, and *pc*. Refer to the C++ *Programming Guide* and C++ reference page for details.

You can invoke a compiler driver with various options (described later in this chapter) and with one or more source files as arguments. All specified source files are automatically sent to the macro preprocessor.

Note: Preprocessing is done by *cfe*. The old preprocessors (*cpp* for “traditional” Kernighan & Ritchie C, or *acpp* for ANSI C) are still available for non-compilation preprocessing and preprocessing for *copt*, *ccom_mp*, and *acom_mp*, in case you want to use them.

Although the macro preprocessor was originally designed for C programs, it is now run by default as part of most compilations. To prevent the preprocessor from being run, specify the **-nocpp** option on the driver command line.

If available, the parallel analyzers *pca* and *pfa* produce parallelized source code from standard source code. The result takes advantage of multiple CPUs (when present) to achieve higher computation rates. *pca* and *pfa* are part of the Power C and Power Fortran packages; for information about these packages and how to obtain them, contact your dealer or sales representative.

The compilers proper, often called “front ends,” translate source code into intermediate code. The available compiler front ends are *cfe* (C), *ccom_mp* and *acom_mp* (parallel C), *fcom* (Fortran 77), and *upas* (Pascal). *ujoin*, *uld*, *umerge*, and *uopt* comprise the optimization subsystem of the compiler system. (For more information about profiling, see Chapter 4, “Using the Performance Tools.” For information about optimization, see Chapter 5, “Optimizing Program Performance.”) *ugen* and *asl* make up the code-generation subsystem of the compiler system.

The linker *ld* combines several object files into one, performs relocation, resolves external symbols, and merges symbol table information for symbolic debugging. The driver automatically runs *ld* unless you specify the **-c** option to skip the linking step.

To see the various utilities a program passes through during compilation, invoke the appropriate driver with the **-v** option (or **+v** for the C++ driver *CC*).

When you compile or link programs, by default, the compiler searches */usr/lib*, */lib*, and */usr/local/lib*. Certain default libraries are automatically linked. Drivers and their respective libraries are listed in Table 1-2.

Table 1-2 Compilers and Default Libraries

Compiler	Default Libraries
cc	<i>libc.so</i>
CC	<i>libC.so, libc.so</i>
f77	<i>libftn.so, libc.so, libm.so</i>

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

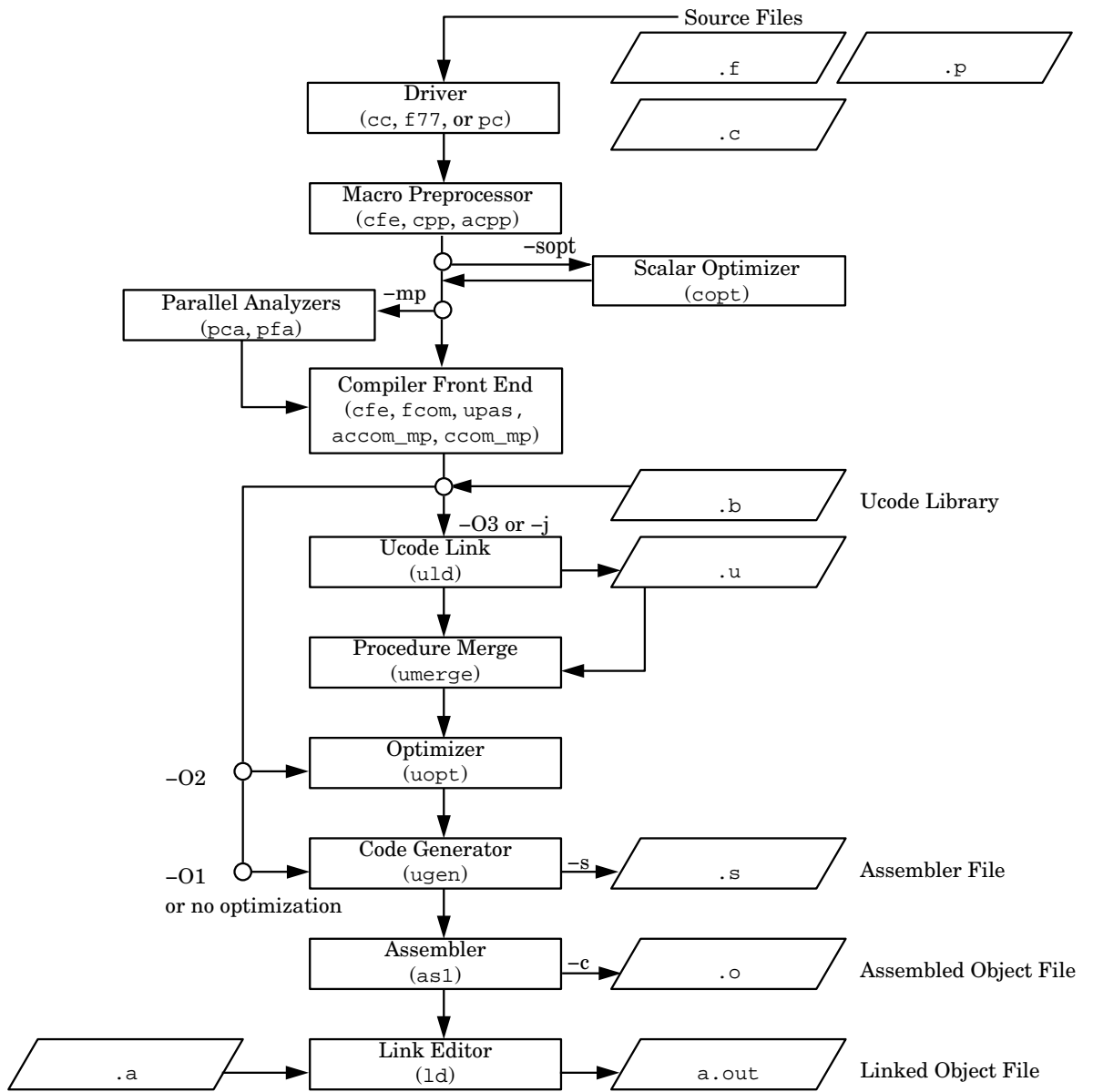


Figure 1-1 Compiler System Flowchart

Chapter 2

Using the Compiler System

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

Using the Compiler System

This chapter provides information about the compiler system and includes information about topics such as object file format and compiler options. Specifically, this chapter contains these sections:

- “Object File Format and Dynamic Linking” discusses the major differences between the latest version of IRIX and previous versions.
- “Source File Considerations” explains source file naming conventions and the procedure for including header files.
- “Compiler Drivers” lists and explains the general compiler-driver options.
- “Linking” explains how to manually link programs (using **ld** or a compiler driver) and how to compile multilanguage programs. It also describes Dynamic Shared Objects and how to link them into your programs.
- “Debugging” explains the compiler-driver options for debugging.
- “Getting Information About Object Files” explains 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 tools such as *dis* and *size*, see Chapter 4, “Using the Performance Tools.” For information about optimizing your program, see Chapter 5, “Optimizing Program Performance.”

Object File Format and Dynamic Linking

A new object file format was adopted in IRIX version 5.0. The major differences between the current compiler system and pre-5.0 compiler systems are summarized below:

- The compiler system uses “Executable and Linking Format” (ELF) for object files.
- The compiler system uses shared libraries, called “Dynamic Shared Objects” (DSOs).
- The compiler system creates “Position-Independent Code,” (PIC) by default, to support dynamic linking.

Executable and Linking Format

Previous versions of IRIX used an extended version of the Common Object File Format (COFF) for object files. The current compiler system produces ELF object files instead. ELF is the format specified by the System V Release 4 Applications Binary Interface (the SVR4 ABI). In addition, ELF provides support for Dynamic Shared Objects, described below. Types of ELF object files include:

- 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.

COFF executables continue to run on new releases of IRIX, but the current compiler system has no facility for creating or linking COFF executables. COFF and ELF object files may not be linked together. To take advantage of new IRIX features, you must recompile your code.

IRIX executes all binaries that are compliant with the SVR4 ABI, as specified in the *System V Applications Binary Interface—Revised Edition* and the *System V ABI MIPS Processor Supplement*. However, binaries compiled under this

version of the compiler system are not guaranteed to comply with the SVR4 ABI. The MIPS-specific version of the SVR4 ABI is referred to as the MIPS ABI. Programs that comply with the MIPS ABI can be run on any machine that supports the MIPS ABI.

Dynamic Shared Objects

IRIX 5.0 introduced a new kind of shared object called a *Dynamic Shared Object*, or *DSO*. 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.

Static shared libraries are only supported under this release for the purposes of running old (COFF) binaries. The current compiler system has no facilities for generating static shared libraries.

You can find additional information about DSOs in Chapter 3, "Dynamic Shared Objects."

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 Appendix A, "Position-Independent Coding in Assembly Language," 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.

The compiler system now produces PIC by default when compiling higher-level language files. All of the standard libraries are now 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 driver 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). These libraries can be linked **-non_shared** with other non-PIC files.

When running position-independent code, the global pointer is used to point to the global offset table, so you can no longer use the **-G** option to store data in the global pointer region (that is, **-KPIC**, the default, implies **-G 0**). The compiler ignores any user-specified **-G** number other than zero. For more information about this option, see the *ld(1)* reference page.

You can find additional information about PIC in Appendix A, "Position-Independent Coding in Assembly Language."

Source File Considerations

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

- "Source File Naming Conventions"
- "Header Files"

Source File Naming Conventions

Each compiler driver recognizes the type of an input file by the suffix assigned to the file name. Table 2-1 describes the possible file name suffixes.

Table 2-1 Driver Input File Suffixes

Suffix	Description
.s	Assembly source
.i	Preprocessed source code in the language of the processing driver
.c	C source

Table 2-1 (continued) Driver Input File Suffixes

Suffix	Description
.C, .cxx, .cc, .c++	C++ source
.f, .F, .for, .FOR	Fortran 77 source
.p	Pascal source
.u	Ucode object file
.b	Ucode object library
.o	Object file
.a	Object library
.so	Dynamic shared object library

The following example compiles preprocessed source code:

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

The Fortran 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 they're associated with. They define such things as data structures, symbolic constants, and prototypes and parameters for the functions in the library.

For example, the *stdio.h* header file describes, among other things, the data types of the parameters required by **printf()**. To use those definitions without having to type them into each of your source files, you can use the *#include* command to tell the macro preprocessor to include the complete text of the given header file in the current source file. Including header files in your source files allows you to specify such definitions conveniently and consistently in each source file that uses any of the library routines.

By convention, header file names have a *.h* suffix. Each programming language handles these files the same way, via the macro preprocessor.

Note: Do not put any code other than definitions in an include file, particularly if you intend to debug your program using *dbx*. The debugger recognizes an include file as only one line of source code, so source lines in an include file do not appear during debugging sessions.

Specifying a Header File

The *#include* command 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 file name 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*).

Another specification format exists, in which the file name is given between double quotation marks:

```
#include "filename"
```

In this case, the macro preprocessor searches for the specified header file in the current directory first, then (if it doesn't find the requested file) goes on and searches in the other directories as in the angle-bracket specification.

Note: When you specify header files in your source files, the *#include* keyword should always start in column 1 (that is, the left-most column) to be recognized by the preprocessor.

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

and so on for other language definitions
```

Note: To indicate C++ definitions you must use `_LANGUAGE_C_PLUS_PLUS`, not `_LANGUAGE_C++`.

You can specify the various language definitions in any order, but you must specify `_LANGUAGE_` before the language name.

Compiler Drivers

The driver commands, such as *cc*, *f77*, and *pc*, 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, optimization, and profiling facilities. You can also specify the names assigned to output files. However, some options have default values that apply if you do not specify the option.

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*. So 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 those together 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 IRIS-4D compiler drivers are listed and explained in Table 2-2. The table lists only the most frequently used options; for a list of all available options, refer to the appropriate compiler reference page. Note that not all of the options work with every driver.

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 *pixie* program (for profiles of basic block counts). Refer to Chapter 4, “Using the Performance Tools,” for details on *prof* and *pixie*.

In addition to the general options in Table 2-2, 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.

Table 2-2 General Driver Options

Option	Purpose
<code>-ansi</code>	Strict ANSI/ISO C compilation mode. Preprocessing adds only standard predefined symbols to the name space, and standard include files declare only standard symbols.
<code>-c</code>	Prevents the linker from linking your program after assembly code generation. This option forces the driver to produce a <i>.o</i> file after the assembler phase, and prevents the driver from producing an executable file.
<code>-C</code>	(C driver only) Used with the <code>-P</code> or <code>-E</code> 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. Note that <code>-C</code> is an option to <i>cfe</i> ; this option is passed along to <i>cfe</i> if you specify it with <i>cc</i> .
<code>-C</code>	(Pascal and Fortran drivers only) Generates code that invokes range checking for subscripts during program execution.
<code>-cord</code>	Runs the procedure rearranger, <i>cord</i> (1) on the resulting file after linking. Rearranging improves the paging and caching performance of the program's text. The output of <i>cord</i> is placed in <i>a.out</i> , by default, or a file specified by the <code>-o</code> option. If you don't specify <code>-feedback</code> , then <i>outfile.fb</i> is used as the default.
<code>-cckr</code>	K&R/Version7 C compatibility compilation mode. Preprocessing may add more predefined symbols to the name space than in <code>-ansi</code> mode. Compilation adheres to the K&R language semantics.
<code>-Dname[=def]</code>	Defines a macro <i>name</i> as if you had specified a <code>#define</code> in your program. If you do not specify a definition with <code>=def</code> , <i>name</i> is set to 1.
<code>-E</code>	(C driver only) Runs only the macro preprocessor and sends results to the standard output. To retain comments, use the <code>-C</code> option as well. Use <code>-E</code> when you suspect the preprocessor is not producing the intended code.

Table 2-2 (continued) General Driver Options

Option	Purpose
-feedback	Use with the -cord option to specify feedback file(s). You can produce this file by using <i>prof</i> with its <i>-feedback</i> option from an execution of the instrumented program produced by <i>pixie(1)</i> . Specify multiple feedback files with multiple -feedback options.
-g[num]	Produces debugging information. The default is -g0 : do not produce debugging information.
-Idirname	Adds <i>dirname</i> to the list of directories to be searched for specified header files. These directories are always searched before the default directory, <i>/usr/include</i> .
-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.
-mips1	Generates code using the instruction set of the MIPS R2000/R3000 RISC architecture. This is the default.
-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.
-nocpp	Suppresses running of the macro preprocessor on 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 <i>libc.a</i> and <i>libm.a</i>) that are provided in non-shared format, in the directory <i>/usr/lib/nonshared</i> . You should use this option only when building device drivers.
-nostdinc	Suppresses searching of <i>/usr/include</i> for the specified header files.
-o filename	Names the result of the compilation <i>filename</i> . If an executable is being generated, it is named <i>filename</i> rather than the default name, <i>a.out</i> . If a single source file is compiled with -c , the object is named <i>filename</i> (not, it should be noted, <i>filename.o</i> ; if you want the object file name to end with <i>.o</i> , you should specify that in the argument to -o). Otherwise, this option is ignored.

Table 2-2 (continued) General Driver Options

Option	Purpose
<code>-P</code>	Runs only the macro preprocessor on the files and puts the result of each file in a <i>.i</i> file. Specify both <code>-P</code> and <code>-C</code> to retain comments.
<code>-S</code>	Similar to <code>-c</code> , except that it produces assembly code in a <i>.s</i> file instead of object code in a <i>.o</i> file.
<code>-Uname</code>	Overrides a definition of the macro <i>name</i> that you specified with the <code>-D</code> 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.
<code>-v</code>	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.
<code>-w</code>	Suppresses warning messages.
<code>-xansi</code>	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 `-xansi` or by using 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*.

This section summarizes the functions of the linker. Also described here are how to link a program manually (without using a compiler driver) and how to compile multilanguage programs. Refer to the *ld(1)* reference page for complete information on the linker.

Specifically, this section explains:

- “Invoking the Linker Manually”
- “Linking Assembly Language Programs”
- “Linking Libraries”
- “Linking to Dynamic Shared Objects”
- “Linking Multilanguage Programs”
- “Finding an Unresolved Symbol With ld”

Invoking the Linker Manually

Usually the linker is invoked by the compiler driver as the final step in compilation (as explained in “Compiler Drivers”). If you have object files produced by previous compilations that you want to link together, you can invoke the linker using a compiler driver instead of calling *ld* directly; just pass the object-file names to the compiler driver in place of source-file names. If the original source files were in a single language, simply invoke the associated driver and specify the list of object files. (For information about linking together objects derived from several languages, see “Linking Multilanguage Programs.”)

A few command-line options to *ld*, such as `-p`, have different meanings when used as command-line options to *cc*. To pass such options to *ld* through an invocation of a compiler driver, use the `-WI` option to the driver (see the reference page for details).

Typically, the compiler driver invokes *ld* as necessary. Circumstances exist under which you may 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). 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 2-3.

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

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

Table 2-3 Linker Options

Option	Purpose
-klibname	Similar to -libname , but the library is a ucode library named <i>liblibname.b</i> .
-libname	Specifies the name of a library, where <i>libname</i> is the library name. The linker searches for a <i>liblibname.so</i> (and then <i>liblibname.a</i>) first in any directories specified by -L dirname options, and then in the standard directories: <i>/lib</i> , <i>/usr/lib</i> , and <i>/usr/local/lib</i> .
-L dirname	Adds <i>dirname</i> to the list of directories to be searched for along with libraries specified by subsequent -libname 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.
-nostdlib	This option must be accompanied by the -L dirname option. If the linker does not find the library in <i>dirname</i> , then it does not search any of the standard library directories.
-o filename	Specifies a name for your executable. If you do not specify <i>filename</i> , the linker names the executable <i>a.out</i> .
-s	Strips symbol table information from the program object, reducing its size. This option is useful for linking routines that are frequently linked into other program objects.

Table 2-3 (continued) Linker Options

Option	Purpose
<code>-v</code>	Prints the name of each file as it is processed by the linker.
<code>-Xsortbss</code>	Sorts bss symbols (this is the default in C but not in Fortran).
<code>-Xnobsschange</code>	Overrides defaults, eliminating all global bss reordering.
<code>-ysymname</code>	Reports all references to, and definitions of, the symbol <i>symname</i> . 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 */lib*. If it does not find that DSO, the linker then looks for *libcurses.a* in */lib*; then for *libcurses.so* in */usr/lib*, then in the same directory for *libcurses.a*. If it hasn't found an appropriate library by then, 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 those places, the DSO or library is linked with the objects *foiled.o* and *again.o*:

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

Note: The `-G` option, which formerly allowed you to specify which data items should be stored in the global pointer region, is no longer useful. `-KPIC`, the default, implies `-G 0`, and the compiler ignores any user attempts to specify otherwise. Compiling `-non_shared` (to avoid `-KPIC`) is primarily useful only for creating device drivers, in which case there is no direct linking step in which to specify a `-G` number. For more information, see the *cc* and *ld* reference pages.

Linking Assembly Language Programs

The assembler driver *as1* 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 *ld* processes its arguments from left to right as they appear on the command line. Arguments to *ld* can be DSOs, object files, or libraries.

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 file name ending in *.a*, it uses only the object files from that archive that can resolve currently unresolved symbol references. (When a symbol is referred to but not defined in any of the object files that have been loaded so far, it'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 library 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 *-llibname* 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 look first in */lib*, then in */usr/lib*. After looking in both of those directories, *ld* looks in */usr/local/lib* for archives only (DSOs should not be installed in */usr/local/lib*). You can modify these defaults by specifying the **-L dir** and/or **-nostdlib** options. Directories specified by **-L dir** before the **-llibname** argument are searched in the order they appear on the command line, before the default directories are searched. If **-nostdlib** is specified, then **-L dir** must also be specified because the default directories aren't searched at all.

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 **-llibname** option. For example:

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

This command invokes, at the linking stage of compilation:

```
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-fashioned kind of 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 in IRIX 5.0, built using *ar* from *.o* files compiled with **-KPIC** (a 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.

When compiling multilanguage programs, be sure to specify any required run-time libraries using the **-llibname** option. For a list of the libraries that a language uses, see the corresponding compiler driver reference page.

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 a Symbol in an Unknown Library."

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). There is no magical connection 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 -lg1 -lX11
```

Linking to 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, "Dynamic Shared Objects."

Note: DSOs replace the older static shared libraries, which were named with the extension *_s.a*. The *_s.a* libraries are no longer shipped with IRIX; however, the run-time versions of those libraries, named with *_s* at the end (and no *.a*), are still present under IRIX 5.0 for backward compatibility with older executables that used static shared libraries.

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
```

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*, and 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

When the source language of the main program differs from that of a subprogram, use the following steps to link (refer to Figure 2-1):

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 driver associated with the language of the main program to link the objects together:

```
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 explicitly specify the default libraries for one or more of the languages used. For instructions on specifying libraries, see “Linking Libraries.”

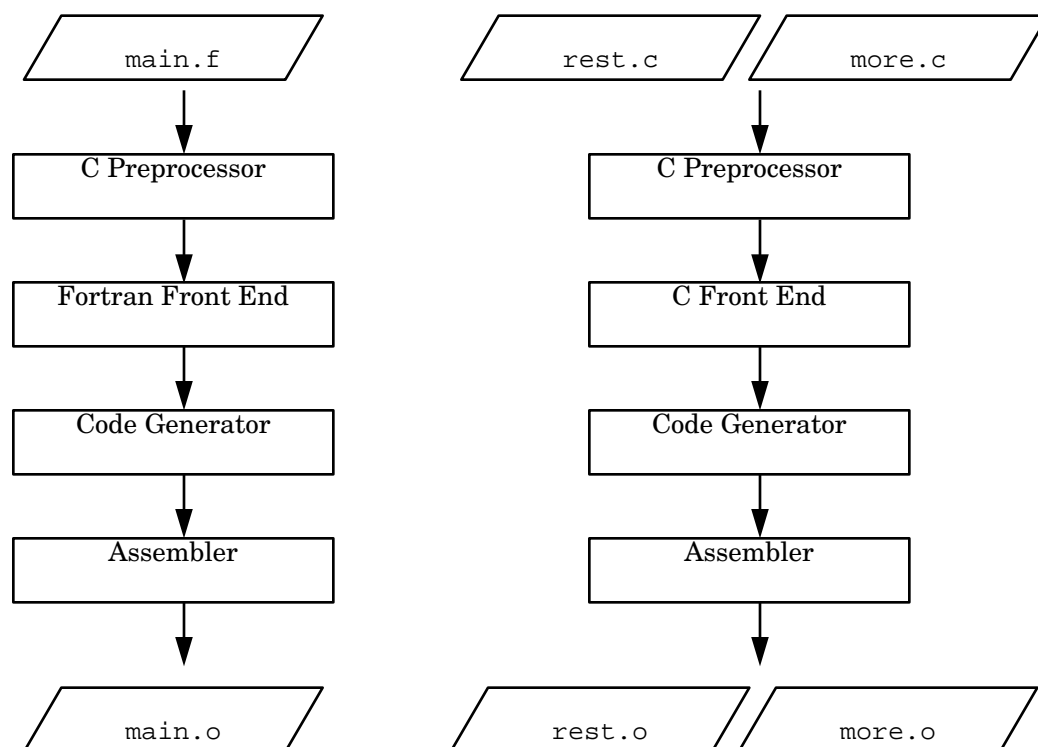


Figure 2-1 Compilation Control Flow for Multilanguage Programs

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 -ysymbol file1... filen
```

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

Debugging

The compiler system provides a debugging tool, **dbx**, which is explained 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 a debugging tool, you must use one of the standard driver options, listed in Table 2-4, to produce executables containing information that the debugger can use.

Table 2-4 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 or -g2	Produces additional debugging information for full symbolic debugging. This option overrides the optimization options (-O <i>num</i>).
-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 <i>num</i>).

Getting Information About Object Files

The following tools provide information on object files:

- *dis* disassembles an object file into machine instructions.
- *elfdump* lists the contents (including the symbol table and header information) of an ELF-format object file.

- *file* provides descriptive information on the general properties of the specified file.
- *nm* lists symbol table information.
- *odump* lists the contents of a COFF-format object file.
- *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 from an object file.

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 2-5.

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

dis Options

Table 2-5 lists *dis* options. For more information, see the *dis(1)* reference page.

Table 2-5 *dis* Options

Option	Description
-b <i>begin_address</i>	Starts disassembly at <i>begin_address</i> . You can specify the address as decimal, octal (with a leading 0), or hexadecimal (with a leading 0x).
-d <i>section</i>	Disassembles the named <i>section</i> as data, and prints the offset of the data from the beginning of the section.
-D <i>section</i>	Disassembles the named <i>section</i> as data, and prints the address of the data.
-e <i>end_address</i>	Stops disassembly at <i>end_address</i> . You can specify the address as decimal, octal (with a leading 0), or hexadecimal (with a leading 0x).
-F <i>function</i>	Disassembles the named <i>function</i> 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.
-i	Removes the leading source line and hexadecimal value of disassembly, and leaves only the instructions.
-I <i>directory</i>	Uses <i>directory</i> to help locate source code.
-l <i>string</i>	Disassembles the archive file specified by <i>string</i> .
-L	Looks up source labels for subsequent printing.
-o	Prints numbers in octal. The default is hexadecimal.
-s	Performs symbolic disassembly where possible. Prints (using C syntax) symbol names on the line following the instruction. Displays source code mixed with assembly code
-t <i>section</i>	Disassembles the named <i>section</i> as text.

Table 2-5 (continued) *dis* Options

Option	Description
-T	Specifies the trace flag for debugging the disassembler.
-V	Prints (on <i>stderr</i>) 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 <i>-s</i> option.
-x	Prints offsets in hexadecimal (the default).

Listing Selected Parts of 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 2-6.

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

elfump Options

Table 2-6 lists *elfdump* options. For more information, see the *elfdump(1)* reference page.

Table 2-6 *elfdump* Options

Option	Dumps
-c	String table information
-cr	Compact relocation information.
-Dc	Conflict list (.conflict) in Dynamic Shared Objects.

Table 2-6 (continued) *elfdump* Options

Option	Dumps
-Dg	Global Offset Table (.got) in Dynamic Shared Objects.
-Dinfo	The .MIPS.dclass section.
-Dinst	The .MIPS.dinst section.
-DI	Library list (.liblist) in Dynamic Shared Objects.
-Dmsym	The Msym table.
-dsym	The .MIPS.dsym section.
-Dsymlib	The library that resolves the symbols in the dysym section.
-Dt	String table entries (.dynsym) of the dynamic symbol table in Dynamic Shared Objects.
-f	The ELF file header.
-h	All section headers in the file.
-hash	Hash table (.hash) entries.
-info	Library information (for example, QuickStart enabled).
-L	Dynamic section (.dynamic) in Dynamic Shared Objects.
-o	Optional program header.
-r	Relocation information.
-reg	Register information (.reginfo) section.
-rpt	Runtime procedure table.
-t	Symbol table (.symtab) entries.

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 is occasionally fooled. For example, it 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. However, using *file* on object files and executables gives somewhat cryptic output.

```
file test.o a.out /lib/libc.so.1
test.o:      ELF 32-bit MSB relocatable MIPS - version 1
a.out:      ELF 32-bit MSB dynamic executable (not stripped) MIPS - version 1
/lib/libc.so.1: ELF 32-bit MSB dynamic lib MIPS - version 1
```

In this example, MSB indicates Most Significant Byte, also called Big-Endian; `dynamic executable` indicates the executable was linked with DSO libraries; and `(not stripped)` indicates the executable contains at least some symbol table information. `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... filename_n]

options One or more of the options listed in Table 2-7.

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

nm Options

Table 2-7 lists symbol table dump options. For more information, see the *nm(1)* reference page.

Table 2-7 Symbol Table Dump Options

Option	Purpose
-a	Prints debugging information. If used with -B, uses BSD ordering with System V formatting.
-A	Prints the listing in System V format (default).
-b	Prints the value field in octal.
-B	Prints the listing in BSD format.
-d	Prints the value field in decimal (the default for System V output).
-e	Prints only external and static variables.
-h	Suppresses printing of headers.
-n	Sorts external symbols by name for System V format. Sorts all symbols by value for Berkeley format (by name is the BSD default output).
-o	Prints value field in octal (System V output). Prints the file name immediately before each symbol name (BSD output).
-p	Lists symbols in the order they appear in the symbol table.
-r	Reverses the sort that you specified for external symbols with the -n and -v options.

Table 2-7 (continued) Symbol Table Dump Options

Option	Purpose
-T	Truncates characters in exceedingly long symbol names; inserts an asterisk as the last character of the truncated name. This option may make the listing easier to read.
-u	Prints only undefined symbols.
-v	Sorts external symbols by value (default for Berkeley format).
-V	Prints the version number of <i>nm</i> .
-x	Prints the value field in hexadecimal.

Table 2-8 defines the one-character codes shown in an *nm* listing. Refer to the example that follows the table for a sample listing.

Table 2-8 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
D	External initialized data
E	Small common data
G	External small initialized data
N	Nil storage class (avoids loading of unused external references)
r	Local read-only data
R	External read-only data
s	Local small zeroed data

Table 2-8 (continued) Character Code Meanings

Key	Description
S	External small zeroed data
t	Local text
T	External text
U	External undefined data
V	External small undefined data

***nm* Example**

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:

```
nm -B tnm.o
0000000000 T main
0000000000 B mydata
0000000000 U sqrt
0000000000 D unused_item
00000000 N _bufendtab
```

To obtain symbol table information for the object file *trm.o* in System V format use the *nm* command without any options:

```
nm trm.o
Symbols from trm.o:
```

[Index]	Value	Size	Class	Type	Section	Name
[0]		0	File	ref=4	Text	trm.c
[1]		0	Proc	end=3 int	Text	main
[2]		116	End	ref=1	Text	main
[3]		0	End	ref=0	Text	trm.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

Finding a Symbol in an Unknown Library

When *ld* indicates that a symbol is undefined, you can use *nm* to figure out which DSO or library needs to be linked in by piping *nm*'s output through appropriate *greps*.

For example, you're compiling a program, and *ld* tells you that you're using an undefined symbol:

```
cc prog.c -lg1
ld:
Unresolved:
XGetPixel
```

However, you don't know where *XGetPixel* is defined. Use *nm* to list the symbol tables for all of the available DSOs, and filter that output to find only the places where *XGetPixel* is mentioned. Then filter the result to find the places where *XGetPixel* is defined, as indicated by the *T* character code.

```
nm -Bo /usr/lib/lib*.so* | grep XGetPixel | grep T
/usr/lib/libX11.so.1: 0f790ff8 T XGetPixel
```

Some DSOs end in *.so*, while others end in *.so.1*, so we need to use multiple wildcards to get all of them. Also, this command line has to be modified to

look in PIC archives or non-shared libraries; as written it only looks in DSOs. Now that *XGetPixel* is defined in */usr/lib/libX11.so.1*, the X11 DSO; use the *-l* option to tell *cc* to link in that library, and *ld* won't complain again.

```
cc prog.c -lg1 -lX11
```

Listing Selected Parts of COFF Files With *odump*

The *odump* tool lists headers, tables, and other selected parts of a COFF-format object or archive file. It is provided with this release of IRIX for compatibility; use *elfdump* for ELF-format files.

odump Syntax

The syntax for *odump* is:

```
odump options filename1 [filename2... filenamen]
```

- options* One or more of the options listed in Table 2-9.
- filename* Specifies the name of one or more object files whose contents are to be dumped.

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

Table 2-9 *odump* Options

Option	Dumps
<i>-a</i>	Archive header of each object file in the specified archive library file.
<i>-c</i>	String table.
<i>-d number</i>	The section numbered <i>number</i> , or a range of sections starting with <i>number</i> and ending with the last section number available (or the number you specify with the <i>+d</i> auxiliary option).
<i>+d number</i>	All sections starting with the first section (or with the section specified with the <i>-d</i> option) and ending with the section numbered <i>number</i> .
<i>-f</i>	File header for each object file in the specified file.

Table 2-9 (continued) *odump* Options

Option	Dumps
-F	File descriptor table for each object file in the specified file.
-g	Global symbols in the symbol table of an archive library file.
-h	Section headers.
-i	Symbolic information header.
-l	Line number information.
-n <i>name</i>	Information for section named <i>name</i> only. Use this option with the -h, -s, -r, -l, or -t option.
-o	Optional header for each object file.
-p	Suppresses the printing of headers.
-P	Procedure descriptor table.
-r	Relocation information.
-R	Relative file index table.
-s	Section contents.
-t	Symbol table entries.
-t <i>index</i>	Only the indexed symbol table entry. Use the +t option with the -t option to specify a range of table entries.
+t <i>index</i>	Symbol table entries in a range that ends with the indexed entry. The range begins with the first symbol table entry or with the section that you specify with the -t option.
-v	Information in symbolic rather than numeric representation. This option may be used with any <i>odump</i> option except -s.
-z <i>name, number</i>	Line number entry (or a range of entries starting at the specified number) for the named function.
+z <i>number</i>	Line number entries starting with the function name or line number specified by the -z option and ending with <i>number</i> .

Determining Section Sizes With *size*

The *size* tool prints information about the sections (such as *text*, *rdata*, and *sbs*) of the specified object or archive files. The *a.out(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 2-10).

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

size options

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

Table 2-10 *size* Options

Option	Action
-A	Prints data section headers in System V format.
-B	Prints data section headers in Berkeley format.
-d	Prints sizes in decimal (default).
-F	Prints data on loadable segments.
-n	Prints symbol table, global pointer, and more.
-o	Prints sizes in octal.
-s	Follows shared libraries, adding them as they're encountered to the list of files to be <i>sized</i> .
-V	Prints the version of <i>size</i> that you are using.
-x	Prints sizes in hexadecimal.

size Example

Below are examples of the *size* command and the listings they produce:

```
size -B -o test.o
      text data bss  rdata sdata sbss decimal hex
test.o 31250 2010 40470 550   210   50   31232   7a00

size -B -d test.o
      text data bss  rdata sdata sbss decimal hex
test.o 12968 1032 16696 360   136   40   31232   7a00
```

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 2-11.

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

***strip* Options**

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

Table 2-11 *strip* Options

Option	Description
<i>-l</i>	Strips line number information, and keeps the symbol table and debugging information.
<i>-o filename</i>	Puts the stripped information in the <i>filename</i> that you specify.

Table 2-11 *strip* Options

Option	Description
-V	Prints the version number of <i>strip</i> .
-x	Keeps symbol table information, but may strip debugging and line number information.

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 the following 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

The following section explains the syntax of the *ar* command and lists some 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; thus, it can be used 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]
```

<i>options</i>	Specifies the action that the archiver is to take. Table 2-12, Table 2-13, and Table 2-14 list the available options. To specify more than one option, don't use a dash or put spaces between the options. For example, use <i>ar ts</i> , not <i>ar -t -s</i> .
<i>posObject</i>	Specifies the name of an object within an archive library. It specifies the relative placement (either before or after <i>posObject</i>) 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 <i>posObject</i> parameter.
<i>libName</i>	Specifies the name of the archive library you are creating, updating, or extracting information from.
<i>object</i>	Specifies the name(s) of the object file(s) to manipulate.

Archiver Options

When running the archiver, specify exactly one of the options **d**, **m**, **p**, **q**, **r**, **t**, or **x** (listed in Table 2-12). In addition, you can optionally specify any of the modifiers in Table 2-13, as well as any of the archiver suboptions listed in Table 2-14.

Table 2-12 Archiver Options

Option	Purpose
d	Deletes the specified objects from the archive.
m	Moves the specified files to the end of the archive. If you want to move the object to a specific position in the archive library, specify an a , b , or i suboption together with a <i>posObject</i> parameter.
p	Prints the specified objects in the archive on the standard output device (usually the terminal screen).

Table 2-12 (continued) Archiver Options

Option	Purpose
q	Adds the specified object files to the end of the archive. This option is similar to the r option (described below), but is faster and 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.
r	Adds the 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. If you want to add an object at a specific position in the archive library, specify an a , b , or i suboption together with a <i>posObject</i> parameter. Use the r option when updating existing libraries.
t	Prints a table of contents on the standard output (usually the screen) 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 2-13 Archiver Modifiers

Option	Purpose
c	Suppresses the warning message that the archiver issues when it discovers that the archive you specified does not already exist.
C	Makes an archive compatible with pre-SVR4 IRIX.
E	The default; creates an archive matching the specifications given by the SVR4 ABI.
l	Puts the archiver's temporary files in the current working directory. Ordinarily, the archiver puts those files in <i>/tmp</i> (unless the <i>STMDIR</i> environment variable is set, in which case <i>ar</i> stores temporary files in the directory indicated by that variable). This option is useful when <i>/tmp</i> (or <i>STMDIR</i>) is full.

Table 2-13 (continued) Archiver Modifiers

Option	Purpose
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 m , p , q , 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.

Table 2-14 Archiver Suboptions

Suboption	Use with Option	Purpose
a	m or r	Specifies that the object file being added should follow the already-archived object file specified by the <i>posObject</i> parameter on the command line.
b	m or r	Specifies that the object file precede the object file specified by the <i>posObject</i> parameter.
i	m or r	Same as b .
o	x	Forces the last modified date of the extracted object file to match that of the archive file.
u	r	Tells the archiver not to replace the existing object file in the archive if the last modified date indicates that the object file already in the archive is newer (more recently modified) than the one you're adding.

Note: The **a** and **b** suboptions are only useful if the same symbol is defined in two or more of the object files in the archive (in which case, the symbol table shows the first definition listed in the archive). Under other circumstances, order of object files in an archive is irrelevant (and the **a** and **b** suboptions are useless), since *ld* uses the archive symbol table rather than searching linearly through the file.

ar Examples

Create a new library, *libtest.a*, and add object files to it by entering:

```
ar cq libtest.a mcount.o mon1.o string.o
```

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

An example of replacing an object file in an existing library:

```
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.

To add a new file immediately before *mcount.o* in this library, enter:

```
ar rb mcount.o libtest.a new.o
```

The **r** option adds *new.o* to the library *libtest.a*. The **b** option followed by *mcount.o* as the *posObject* causes the archiver to place *new.o* immediately before *mcount.o* in the archive.

Chapter 3

Dynamic Shared Objects

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

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. DSOs can be used in place of archive libraries, and they replace static shared libraries provided with earlier releases of the IRIX operating system.

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

- “Benefits of Using DSOs” describes some benefits (such as minimizing memory usage) of using DSOs.
- “Using DSOs” tells you how to obtain the most benefit from using DSOs when creating your executable and covers a few guidelines for using shared libraries.
- “Taking Advantage of QuickStart” explains how you can make sure that DSOs load as quickly as possible.
- “Building DSOs” describes how to build a DSO.
- “Runtime Linking” discusses the run-time linker, and how it locates DSOs at run time.
- “Dynamic Loading Under Program Control” explains the use of the *libdl* library 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.

Benefits of Using DSOs

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

- 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 shared components 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.

- A related benefit is that 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 much easier to use, build, and debug than static shared libraries. Most of the libraries supplied by Silicon Graphics are available as DSOs. (In the past, only a few static shared libraries were available; most libraries were unshared.)
- 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.

You can design the hardware-dependent routines required by your application so that they have the same interface across all platforms. Then, you create different DSOs for each of the platforms, each DSO containing the implementation of those routines for that particular platform. The shrink-wrapped software package can then contain all the DSOs and is able to run on all the platforms.

- DSOs and the executables that use them are mapped into memory by a run-time loader, *rld*. It 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 can 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 can 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.

Naturally, some costs are involved with using DSOs, and these are explained in the next section, “Using DSOs.” The sections after that explain how to build and optimize DSOs and how *rld* works. 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*.

You should note that a significant difference exists between DSOs and archive libraries in terms of what gets loaded into memory 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) gets loaded, not the entire library. In contrast, the entire DSO that’s

linked gets loaded. Thus, to conserve memory, don't link with DSOs unless your application actually needs them.

Also, you should avoid listing any archive libraries on the compile line after you list shared libraries; instead, list the archive libraries first and then list 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 quickly (or QuickStarts). When an application QuickStarts, memory use is smaller 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 how 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 unshared 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 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 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. It probably should not be included 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 -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 currently 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. (See "Using Registry Files" for more information about how to use the registry when you're building a DSO.) 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 it fails. QuickStart 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* aren't identical, *yourApp* won't be able to QuickStart.

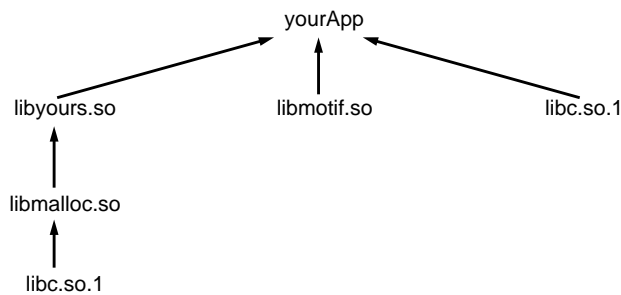


Figure 3-1 An Application Linked with DSOs

- You link with a DSO that can't QuickStart. This may occur because the DSO wasn't registered and therefore was assigned a location that overlaps with the location assigned 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 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*. (Conflicts are resolved by proceeding through the hierarchy from left to right and then moving to the next level. See "Searching for DSOs at Runtime" for more information about how the run-time linker searches for DSOs.)

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 override function definitions intentionally, but you should be sure you have control over what is overriding what over time.

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 in assembly language, you must modify it to produce PIC, as described in Appendix A, "Position-Independent Coding in Assembly Language."

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

- "Creating DSOs"
- "Making DSOs Self-Contained"
- "Controlling Symbols to be Exported or Loaded"
- "Using DSOs With C++"
- "Using Registry Files"

Creating DSOs

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

```
ld -shared stuff.o nonsense.o -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 **-llib** 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 -o libdada.so
```

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

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. To make the archive *libmine.a* into a DSO, for example, and *libmine.a* depends on routines in another archive, *libutil.a*, include *libutil.a* on the link line:

```
ld -shared -all -no_unresolved libmine.a -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. See “Controlling Symbols to be Exported or Loaded” for more information about exported symbols. The **-no_unresolved** option causes a list of unresolved symbols to be created; generally, this list should be empty to enable QuickStarting.

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 -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_file* 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.

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

Using DSOs With C++

To make a DSO, build the C++ objects as you would normally:

```
CC -c
```

Then type:

```
CC -shared -o libmylib.so <list your objects here>
```

For example:

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

In this instance, the `-I` and `-L` options to *ld* will work. However, most *ld* options won't work. If you want to specify other options, first determine the options that you must pass to *ld*. These options include:

```
-init _main  
-fini _fini  
-hidden_symbol _main  
-hidden_symbol _fini  
-hidden_symbol __head  
-hidden_symbol __endlink
```

Finally, link in `/usr/lib/c++init.o`.

Using Registry Files

You can make sure that your DSOs don't conflict with each other by using a QuickStart registry file. The registry files contain location information for shared objects. When creating a shared object, you can specify a registry file to *ld*, and *ld* ensures that your shared object doesn't conflict with any of the shared objects listed in the registry. A registry file containing the locations of all the shared objects provided with the system is supplied in `/usr/lib/so_locations`.

You can use two options to *ld* to specify a registry file: `-check_registry` and `-update_registry`. When you invoke *ld* to build a shared object, with the

argument `-check_registry file`, `ld` makes sure that the new shared object doesn't conflict with any of the shared objects listed in `file`. When invoked with `-update_registry file`, `ld` checks the registry in the same way, but when it's done, it writes an entry in `file` for the DSO being built. If `file` isn't writable, `-update_registry` acts like `-check_registry`. If `file` isn't readable, both `-update_registry` and `-check_registry` are ignored.

By exchanging registry files, providers of DSOs can avoid collisions between their shared objects. You should probably start out with a copy of `/usr/lib/so_locations`, so that your shared objects won't conflict with any of the standard DSOs. However, you should remember that when collisions occur between shared objects, the only effect is slowing program startup.

Registry File Format

Three types of lines in the registry file include:

- comment lines, which begin with a pound sign (#)
- directive lines, which begin with a dollar sign (\$)
- shared object specification lines, which begin with the name of a shared object

Comment lines are ignored by `ld`. Directive lines and shared object specification lines are described below.

Directive Lines

Directive lines specify global parameters that apply to all the DSOs listed in the registry.

```
$text_align_size=align padding=pad-size  
$data_align_size=align padding=pad-size
```

These two directives specify the alignment and padding requirements for text and data segments, respectively. The current default segment alignment is 64K, which is the minimum permissible. The size value of a segment of a DSO appearing in the registry file is calculated based on the actual section size plus padding, and is aligned to the section align size (either the default or the one specified by the above directive). The align values for text and data as well as the padding values must be aligned to the minimum

alignment size (64K). If not, *ld* generates a warning message and aligns these values to the minimum alignment.

```
$start_address=addr
```

This directive specifies where to start looking for addresses to put shared objects. The default *start_address* is 0x6000000.

```
$data_after_text={ 1 | 0 }
```

In this directive, a value of one instructs the linker to place data immediately after the text at specified text and data alignment requirements. A value of zero (the default) allows the linker to place these segments in different portions of the address space.

Shared Object Specification Lines

Shared object specification lines have the format:

```
so_name [ :st = { .text | .data | $range } base_addr, padded_size : ] *
```

where:

<i>so_name</i>	full path name (or trailing component) of a shared object
:st =	literal string indicating the beginning of the segment description
.text, .data	segment types: text or data
\$range	range of addresses that can be used
<i>base_addr</i>	address where the segment starts
<i>padded_size</i>	padded size of the segment
:	literal string indicating the end of the segment description

A shared object specification can span several lines by “escaping” the newline character (using “\” as the last character on the line that is being continued). The following is an example of a shared object specification line:

```
libc.so.1 \  
    :st = $range 0x5fc00000, 0x00400000:\  
    :st = .text 0x5fe40000, 0x000a0000:\  
    :st = .data 0x5fee0000, 0x00030000:
```

This specification instructs *ld* to relocate all segments of *libc.so.1* in the range 0x5fc00000 to 0x5fc00000+0x0040000, and, if possible, to place the text segment at 0x5fe40000 and the data segment at 0x5fee0000. The text segment should be padded to 0xa0000 bytes and the data segment to 0x3000 bytes. See */usr/lib/so_locations* for examples of shared object specifications.

When building a DSO with the **-check_registry** or **-update_registry** flag, if an entry corresponding to this DSO exists in the registry file, the linker tries to assign the indicated addresses for text and data. However, if the size of the DSO changes and no longer fits in the specified location, the linker searches for another location that fits. If the *\$range* option is specified, the linker places the DSO only in the specified range of addresses. If there isn't enough room, an error is returned.

Runtime 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 Runtime”
- “Runtime Symbol Resolution”

Searching for DSOs at Runtime

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 the following sequence:

1. the path of the DSO in the *liblist* (if an explicit path is given)
2. RPATH if it's defined in the main executable

3. LD_LIBRARY_PATH if defined
4. the default path (*/usr/lib/lib*)

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

```
ld -o myprog myprog.c -rpath /d/src/mylib 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 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 you set 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.

Runtime 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, it will cause the executable to

exit 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 `LD_BIND_NOW` is set, the executable will exit with an error, just as if there were unresolvable data symbols.

Compiling with `-Bsymbolic`

When you compile 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 compile *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*:

```
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**, this program prints:

```
1
```

Converting 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, you should exercise 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 dynamically loaded 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” below.

Dynamic Loading Under Program Control

IRIX provides a library interface to the run-time linker that allows programs to dynamically load and unload DSOs. This interface is called *libdl*, and it consists of four functions listed in Table 3-1.

Table 3-1 *libdl* functions

<code>dlopen()</code>	load a DSO
<code>dlsym()</code>	find a symbol in a loaded DSO
<code>dlclose()</code>	unload a DSO
<code>dlerror()</code>	report errors

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 should 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.

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 */
}
i = (*funcptr)(j);
```

In this example, we look up the address of a function called **factorial()** and assign 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 only call **dLError()** after an error has occurred; at other times, its return value is undefined.

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 **mmap**ed by the **dlopen** call of that file (similar to a **munmap** call). The difference, however, is that **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 version mechanism:

- “The Versioning Mechanism of Silicon Graphics”
- “What Is a Version?”

The Versioning Mechanism of Silicon Graphics

In the IRIX 5.0.1 release, a mechanism for the versioning of shared objects was introduced for SGI-specific shared objects and executables. Note that this mechanism is outside the scope of the ABI, and, thus, must not be relied on for code that must be ABI-compliant and run on non-SGI platforms. Currently, all executables produced on SGI systems are marked `SGI_ONLY` to allow use of the versioning mechanism.

Versioning is mainly of interest 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. This is accomplished 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#]sgimajor.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].

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 file name of the old shared object by adding a dot followed by the major number of one of the versions to the file name 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 -DI*.
- When you run an executable, *rld* looks for the proper file name 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 file name 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 file name. The new file is searched for using *rld*'s usual search procedure.

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 file name *foo.so.10* and create a new *foo.so* with the *version_string*:

```
efficient_interfaces#sgi11.0
```

New executables linked with *foo.so* use it directly. Older executables, like *useoldfoo*, attempt to use *foo.so*, but find that its version (*sgi11.0*) is not the

version they need (*sgi10.2*). They then attempt to find a *foo.so* in the file name *foo.so.10* with version *sgi10.2*.

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

Chapter 4

Using the Performance Tools

This chapter describes how to use performance tools such as *prof*, *pixie*, and *cord*.

Using the Performance Tools

This chapter explains how to use performance tools to reduce the execution time of your programs. This chapter describes *prof*, *pixie*, and *cord*. For information about the compiler optimization options, see Chapter 5, “Optimizing Program Performance.”

This chapter covers the following topics:

- “Overview of Profiling” explains how profiling can help you to analyze your data.
- “Profiling With *prof*” describes how to run the profiler, *prof* and lists its options.
- “pc Sampling” explains how to use *prof* to obtain program counter (pc) sampling.
- “Basic Block Counting” covers how to use *prof* and *pixie* perform basic block counting.
- “Profiling Multiprocessed Executables” describes how to profile executables that use *sproc* and *sprobsp* system calls.
- “Rearranging Procedures With *cord*” explains how to rearrange procedures to reduce paging and achieve better instruction cache mapping.

Although it may be possible to obtain short-term speed increases by relying on unsupported or undocumented quirks of the compiler system, it’s a bad idea to do so. Any such “features” may break in future releases of the compiler system.

The best way to produce efficient code that can be trusted to remain efficient is to follow good programming practices; in particular, choose good algorithms and leave the details to the compiler.

The techniques described in this manual comprise only a part of performance tuning. Other areas that you can tune, but are outside the scope of this document, include graphics, I/O, the kernel, system parameters, memory, and real-time system calls.

Overview of Profiling

Profiling is a three-step process that consists of compiling the source program, executing the program, and then running the profiler, *prof*, to analyze the data.

The compiler system provides two types of profiling:

- *Program counter (pc) sampling*, which measures the amount of execution time spent in various parts of the program. This statistical data is obtained by periodically sampling the program counter. For example, *cc -p* interrupts the program ever 10 milliseconds, and records the value of the program counter. By default, *prof* generates pc sampling data.
- *Basic block counting*, which counts the execution of basic blocks (a basic block is a sequence of instructions that is entered only at the beginning and exits only at the end). It produces an exact count of the number of times each basic block is executed, thereby providing more detailed information than pc sampling.

Profiling With *prof*

Profiling produces detailed information about program execution. You can use profiling tools to find the areas of code where most of the execution time is spent. In a typical program, a large part of the execution time is spent in relatively few sections of code. It is a good idea to concentrate on improving code efficiency in those sections first.

The topics covered below include:

- “Running the Profiler”
- “prof Options”

Running the Profiler

The profiler program, *prof(1)*, analyzes raw profiling information and produces a printed report. The program analyzes either pc sampling or basic block counting data.

prof Syntax

The syntax for *prof* is:

```
prof [options] [prog_name] [profile_filename ...]
```

<i>options</i>	One of the keywords or keyword abbreviations shown in Table 4-1. (Specify either the entire name or the initial character of the option, as indicated in the table.)
<i>prog_name</i>	Specifies the name of the program whose profile data is to be profiled.
<i>profile_filename</i>	Specifies one or more files containing the profile data gathered when the profiled program executed (defaults are explained below). If you specify more than one file, <i>prof</i> sums the statistics in the resulting profile listings.

prof Defaults

The *prof* program has these defaults:

- If you do not specify *-pixie*, *prof* assumes pc-sampling data is being analyzed. If you do not specify *profile_filename*, the profiler looks for a *mon.out* file. If this file does not exist in the current directory, *prof* looks for profile input data files in the directory specified by the PROFDIR environment variable (see “Creating Multiple Profile Data Files” for information on PROFDIR).

You may want to use the *-merge* option when you have more than one profile data file. This option merges the data from several profile files into one file.

- If you specify *-pixie* and do not specify *profile_filename*, then *prof* looks for *prog_name.Counts* and provides basic block count information if this file is present.

- If you specify *profile_filename(s)*, *prof* determines the file type based on its content: a *prof*- or *pixie*-mode file.

***prof* Options**

Table 4-1 lists *prof* options. Options that apply to basic block counting are indicated as such.

Table 4-1 Options for *prof*

Name	Result
<code>-c[lock] n</code>	A basic-block-counting option. This option lists the number of seconds spent in each routine, based on the CPU clock frequency <i>n</i> , expressed in megahertz.
<code>-d[is]</code>	Disassembles and annotates the analyzed object code with cycle times or number of pc samples.
<code>-dso [dso_name]</code>	Applies <i>prof</i> analysis to only the named DSO. If you don't specify <i>dso_name</i> , <i>prof</i> prints a list of applicable DSO names.
<code>-e[exclude] procedure_name</code>	Excludes information on the procedures specified by <i>procedure_name</i> . If you specify uppercase <code>-E</code> , <i>prof</i> also omits that procedure from the base upon which it calculates percentages.
<code>-h[eavy]</code>	A basic-block-counting option. Same as the <code>-lines</code> option, but sorts the lines by their frequency of use.
<code>-i[nvocations]</code>	A basic-block-counting option. Lists the number of times each procedure is invoked. The <code>-exclude</code> and <code>-only</code> options described below apply to callees, but not to callers.
<code>-l[ines]</code>	Lists statistics for each line of source code.
<code>-m[erge] filename</code>	Merges the input files into <i>filename</i> (the default is <i>mon.out</i>), allowing you to specify the name of the merged file (instead of several filenames) on subsequent profiler runs. This option is useful when using multiple input files of profile data.
<code>-o[nly] procedure_name</code>	Reports information on only the procedure specified by <i>procedure_name</i> rather than the entire program. You can specify more than one <code>-o</code> option. If you specify uppercase <code>-O</code> , <i>prof</i> uses only the named procedures, rather than the entire program, as the base upon which it calculates percentages.

Table 4-1 (continued) Options for *prof*

Name	Result
<i>-pcsample</i>	Tells <i>prof</i> that the data to be analyzed is from pc sampling. This is the default. This option and <i>-pixie</i> are mutually exclusive.
<i>-pixie</i>	A basic-block-counting option. Indicates that information is to be generated on basic block counting, and that the <i>prog_name.Counts</i> files produced by <i>pixie</i> are to be used by default. This option and <i>-pcsample</i> are mutually exclusive.
<i>-p[rocedures]</i>	Lists the time spent in each procedure.
<i>-q[uit] n</i>	Condenses output listings by truncating unwanted lines. You can specify <i>n</i> in three ways: <i>n</i> , an integer, truncates everything after <i>n</i> lines; <i>n%</i> , an integer followed by a percent sign, truncates everything after the line containing <i>n%</i> calls in the <i>%calls</i> column; <i>ncum%</i> , an integer, followed by <i>cum%</i> , truncates everything after the line containing <i>ncum%</i> calls in the <i>cum%</i> column.
<i>-t[estcoverage]</i>	A basic-block-counting option. Lists line numbers that contain code that is never executed.
<i>-z[ero]</i>	A basic-block-counting option. Lists the procedures that are never invoked.

pc Sampling

Program counter (pc) sampling reveals the amount of execution time spent in various parts of a program. The count includes:

- CPU time and memory access time
- Time spent in user routines

The pc sampling does not count time spent swapping or time spent accessing external resources.

This section explains how to obtain pc sampling and provides examples showing the use of various *prof* options. Specifically, this section covers:

- “Obtaining pc Sampling”
- “Creating Multiple Profile Data Files”

- “pc Sampling Frequency”
- “Examples Using prof to Obtain pc Sampling”

Obtaining pc Sampling

Obtain pc sampling information by linking the desired source modules using the `-p` option and then executing the resulting program object, which generates raw profile data.

Use the procedure below to obtain pc sampling information. Also refer to Figure 4-1, which illustrates how pc sampling works.

1. Compile the program using the appropriate compiler. For example, to compile a C program *myprog.c*:

```
% cc -c myprog.c
```

2. Link the object file created in Step 1.

```
% cc -p -o myprog myprog.o
```

Note: You must specify the `-p` profiling option during this step to obtain pc sampling information.

3. Execute the profiled program (just as you would execute an unprofiled program).

```
% myprog
```

During execution, profiling data is saved in the file *mon.out*. You can run the program several times, altering the input data, to create multiple profile data files. You can also use the environment variable `PROFDIR` as explained in “Creating Multiple Profile Data Files.”

4. Run the profile formatting program *prof*.

```
% prof -pcsample myprog mon.out
```

prof extracts information from *mon.out* and prints it in an easily readable format. If *mon.out* exists, it is overwritten. Therefore, rename each *mon.out* to save its output. For more information, see the *prof(1)* reference page.

Include or exclude information on specific procedures within your program by using the `-only` or `-exclude` profiler options (refer to Table 4-1).

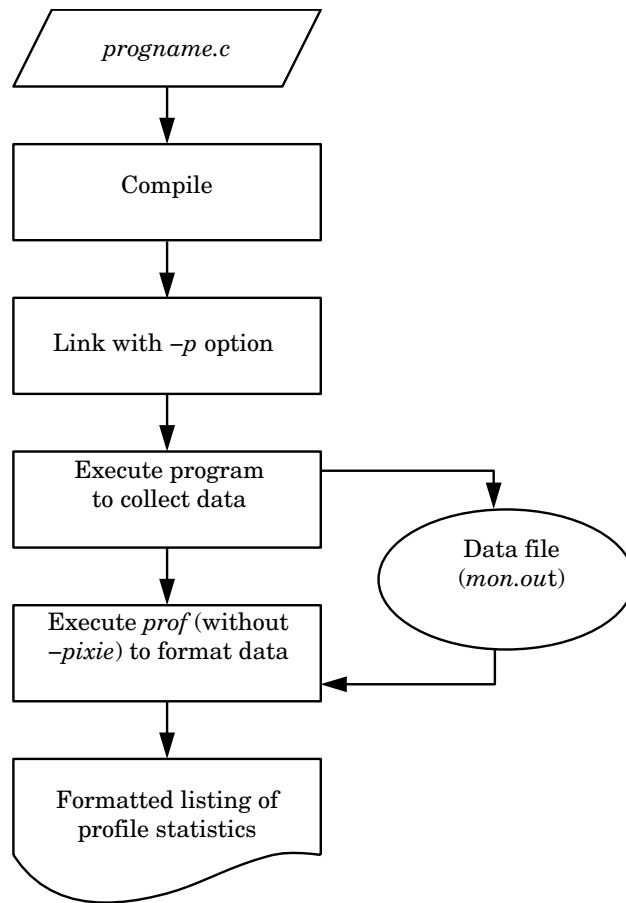


Figure 4-1 How pc Sampling Works

Creating Multiple Profile Data Files

When you run a program using pc sampling, raw data is collected and saved in the profile data file *mon.out*. To collect profile data in several files, or to

specify a different name for the profile data file, set the environment variable `PROFDIR`, using the appropriate method from Table 4-2.

Table 4-2 Setting a `PROFDIR` Environment Variable

C shell and tcsh	Bourne shell and Korn shell
<code>setenv PROFDIR <i>dirname</i></code>	<code>PROFDIR=<i>dirname</i>; export PROFDIR</code>

Setting the environment variable puts the raw profile data of each invocation of *progrname* in files named *dirname/progrname.mon.pid*. (You must create a directory named *dirname* before you run the program.) *pid* is the process ID of the executing program; *progrname* is the name of the program when invoked.

pc Sampling Frequency

The default frequency of pc sampling is 10 milliseconds. You can change the pc sampling to 1 millisecond by setting the environment variable:

```
PROF_SAMPLING=1
```

Since pc sampling is statistical, this provides more accurate profiling data. However, be aware that considerable kernel overhead is incurred for every process executing on the system while the profiled program is running.

Examples Using prof to Obtain pc Sampling

The examples in this section illustrate how to use *prof* and its options to obtain pc sampling data.

Example Using prof -pcsample

The following partial listing is an example of pc sampling output from a profiled version of the program *test*.

```
-----
Profile listing generated Tue Oct  4 14:16:30 1994
with:      prof -pcsample test test.mon.207
-----
samples  time   CPU   FPU  Clock  N-cpu  S-interval  Countsize
  7473   75s  R4000 R4010 100.0MHz  0     10.0ms     0(bytes)
Each sample covers 4 bytes for every 10.0ms ( 0.01% of 74.7300sec)
-----
-p[rocedures] using pc-sampling.
Sorted in descending order by the number of samples in each procedure.
Unexecuted procedures are excluded.
-----
samples  time(%)   cum time(%)   procedure (file)
  3176   32s( 42.5)  32s( 42.5)    _cerror (/usr/lib/libc.so.1:cerror.s)
  2564   26s( 34.3)  57s( 76.8)    _doprint (/usr/lib/libc.so.1:doprint.c)
   578   5.8s(  7.7)  63s( 84.5)    _isatty (/usr/lib/libc.so.1:isatty.c)
   441   4.4s(  5.9)  68s( 90.4)    offtime (/usr/lib/libc.so.1:time_comm.c)
   217   2.2s(  2.9)  70s( 93.3)    atoi (/usr/lib/libc.so.1:atoi.c)
...

```

In the above listing:

- The `samples` column reports the number of samples in each procedure, sorted in descending order. For example, there were 3176 samples for the procedure `_cerror`.
- The `time(%)` column lists the number of seconds and percentage of execution time spent in each procedure. For example, there were 32 seconds (42.5% of execution time) spent in `_cerror`.
- The `cum time(%)` column lists the percentage of the total execution time spent in each procedure. For example, there were 63 seconds (84.5% of total execution time) were spent cumulatively in the `_cerror`,

`_doprnt`, and `_isatty` procedures. Note that this does not imply that these routines called each other; they may have executed sequentially.

- The `procedure (file)` column prints the procedure name and its source file. For example, the source file containing the `_cerror` procedure is `/usr/lib/libc.so.1:cerror.s`.

Example Using `prof -pixie -dis`

You can use the `-dis` option to `prof` to disassemble the analyzed object code and see the number of cycles it takes to execute an instruction. Check the disassembled code for stalls (wasted cycles) and the number of instructions per cycle. For example, partial output looks like this:

```
Profile listing generated Tue Oct  4 13:43:41 1994
with:      prof -pixie -dis hello
-----
```

Total cycles	Total Time	Instructions	Cycles/inst	Clock	Target
5005674	0.05006s	3003856	1.666	100.0MHz	R4000

28 cycles due to code that could not be assigned to any source procedure.

1000547: Total number of Load Instructions executed.
4001532: Total number of bytes loaded by the program.
200261: Total number of Store Instructions executed.
800774: Total number of bytes stored by the program.

100223: Total number nops executed in branch delay slot.
200558: Total number conditional branches executed.
200418: Total number conditional branches actually taken.
0: Total number conditional branch likely executed.
0: Total number conditional branch likely actually taken.

601965: Total cycles waiting for current instr to finish.
4202824: Total cycles lost to satisfy scheduling constraints.
3001119: Total cycles lost waiting for operands be available.

```
-----
* -p[rocedures] using basic-block counts. *
* Sorted in descending order by the number of cycles executed in each *
* procedure. Unexecuted procedures are not listed. *
-----
```



```

cycles(%) cum %   secs   instrns   calls procedure(file)
5000044(99.89) 99.89   0.05  3000032    1 main(hello:hello.c)
1566( 0.03) 99.92   0.00    985 3 fflush(/usr/lib/libc.so.1:flush.c)
1332( 0.03) 99.95   0.00    861 1 _doprint(/usr/lib/libc.so.1:doprint.c)
1130( 0.02) 99.97   0.00   1004 2 _dtoa(/usr/lib/libc.so.1:dtoa.s)
320( 0.01) 99.97   0.00    120 4 _dwmultu(/usr/lib/libc.so.1:tenscale.s)
302( 0.01) 99.98   0.00    196 4 memcpy(/usr/lib/libc.so.1:bcopy.s)
...
-----
* -dis[assemble] listing annotated with cycle counts. *
*   Unexecuted procedures are excluded. *
-----
ctrltext.s
__start: <0x400900-0x400a08>
  77 total cycles(0.00%) invoked 1 times, average 77 cycles/invoke
[91] 0x00400900 0x03e04025 or t0,ra,0 # 1
[91] 0x00400904 0x04110001 bgezal zero,0x40090c # 2
[91] 0x00400908 0000000000 nop # 3
      <2 cycle stall for following instruction>
^--- 5 total cycles(0.00%) executed 1 times, average 5 cycles.---^
[91] 0x0040090c 0x3c1c0fc0 lui gp,0xfc0 # 6
...

```

The previous listing shows statistics about the file `hello`. The statistics detail procedures using basic-block counts and disassembled code. Information at the top of the listing is self-explanatory. Of interest are cycles waiting and cycles lost.

The `-p[rocedures]` information uses basic-block counts to sort in descending order the number of cycles executed in each procedure.

- The `cycles(%)` column lists the number of cycles (and percentage of total cycles) per procedure. For example, there were 5000044 cycles (or 99.89%) for the procedure `main`.
- The `cum%` column shows the cumulative percentage of cycles. For example, `main` used 99.89% of all cycles.
- The `secs` column reports the number of seconds spent in the procedure. For example, 0.05 seconds were spent in `main`.
- The `instrns` column lists the number of instructions executed in the procedure. For example, 3000032 instructions were executed in `main`.

- The `calls procedure(file)` column shows the number of calls in the procedure. For example, there was 1 call in `main`.

The `-dis[assemble]` information provides a listing containing cycle counts. It lists the beginning and ending addresses of `crt1text.s __start: <0x400900-0x400a08>`. It also reports the total cycles for a procedure, number of times invoked, and average number of cycles per invocation:

77 total cycles(0.00%) invoked 1 times, average 77 cycles/invocation

- The first column lists the line number of the instruction: [91]
- The second column lists the beginning address of the instruction:
0x00400900
- The third column shows the instruction in hexadecimal: 0x03e04025.
- The fourth column reports the assembler form (mnemonic) of the instruction: `or t0,ra,0`
- The last column reports the cycle in which the instruction executed: # 1

Other information includes:

- The total number of cycles in a basic block and the percentage of the total cycles for that basic block, the number of times the branch terminating that basic block was executed, and the number of cycles for one execution of that basic block:
5 total cycles(0.00%) executed 1 times, average 5 cycles
- Any cycle stalls (cycles that were wasted):
<2 cycle stall for following instruction>

For information on cycle stalls and what causes them, see the *MIPS Microprocessor Chip Set User's Guide* for your architecture.

Basic Block Counting

Basic block counting, obtained using the program *pixie*, measures the execution of basic blocks. A basic block is a sequence of instructions that is entered only at the beginning and exits only at the end. This section covers:

- "Using *pixie*"
- "Obtaining Basic Block Counts"

- “Summing Basic Block Count Results”
- “Profiling Multiprocessed Executables”

Using *pixie*

Use *pixie*(1) to measure the frequency of code execution. *pixie* reads an executable program, partitions it into basic blocks, and writes (instruments) an equivalent program containing additional code that counts the execution of each basic block.

Note that the execution time of an instrumented program is two-to-five times longer than an uninstrumented one. This timing change may alter the behavior of a program that deals with a graphical user interface (GUI), or depends on events such as SIGALARM that are based on an external clock.

pixie Syntax

The syntax for *pixie* is:

```
pixie prog_name [options]
```

prog_name Name of the input program.

options One of the keywords listed in Table 4-3.

pixie Options

Table 4-3 lists *pixie* options. For a complete list of options refer to the *pixie*(1) reference page.

Table 4-3 Options for *pixie*

Name	Result
<i>-pixie_file out_file</i>	Specifies a name for the instrumented output file. The default is to remove any leading directory names from the input filename and append <i>.pixie</i> .
<i>-counts_file file</i>	Specifies a name for the counts file that is generated while running the instrumented program. The default is to remove any leading directory names from the input filename and append <i>.Counts</i> .

Table 4-3 (continued) Options for *pixie*

Name	Result
- <i>[no]autopixie</i>	Permits (or prevents) a recursive instrumenting all dynamic shared libraries used by the input file during run time. <i>pixie</i> keeps the timestamp and checksum from the original executable. Thus, before instrumenting a shared library, <i>pixie</i> checks any <i>lib.pixie</i> files that it finds matching the <i>lib</i> it is to instrument. If the fields match, they are not instrumented. <i>pixie</i> cannot detect shared libraries opened with <i>dlopen</i> (and hence does not instrument them). All used DSOs need to be instrumented for the <i>a.out</i> to work. The default behavior for shared libraries is <i>-noautopixie</i> , and can be overridden with <i>-autopixie</i> , which is the default in all other cases.
- <i>[no]liblist</i>	Prevents (or permits) printing the names and paths of dynamic shared libraries used by the input program during run time. This uses the same default search path as <i>rld</i> and <i>prof</i> . This list is useful to build a dependency list for <i>makefiles</i> and shell scripts. <i>pixie</i> removes any leading directory names from the input filename and appends <i>.liblist</i> . <i>pixie</i> cannot detect libraries opened with <i>dlopen</i> . The default is <i>-noliblist</i> .
- <i>[no]pids</i>	Appends the process ID number on the end of the <i>.Counts</i> file. This is useful if you want to run the program instrumented with <i>pixie</i> through a variety of tests. This option is only needed for the <i>main</i> program. It will be transferred automatically to the instrumented DSOs during run time. The default is <i>-nopids</i> .
- <i>[no]verbose</i>	Suppresses (or prints) messages summarizing the binary-to-binary translation process. The default is <i>-noverbose</i> .

Obtaining Basic Block Counts

Use this procedure to obtain basic block counts. Also refer to Figure 4-2, which illustrates how basic block counting works.

1. Compile and link your program. Do not use the *-p* option. The following example uses the input file *myprog.c*.

```
% cc -o myprog myprog.c
```

The *cc* compiler compiles *myprog.c* into an executable called *myprog*.

2. Run *pixie* to generate the equivalent program containing basic-block-counting code.

```
% pixie myprog
```

pixie takes *myprog* and writes an equivalent program, *myprog.pixie*, containing additional code that counts the execution of each basic block. *pixie* also writes an equivalent program for each shared object used by the program (in the form: *libname.so.pixie*), containing additional code that counts the execution of each basic block. For example, if *myprog* uses *libc.so.1*, *pixie* generates *libc.so.1.pixie*.

3. Set the path for your *.pixie* files. *pixie* uses the *rld* search path for libraries (see *rld(1)* for the default paths). If the *.pixie* files are in your local directory, set the path as:

```
% setenv LD_LIBRARY_PATH .
```

4. Execute the file(s) generated by *pixie* (*myprog.pixie*) in the same way you executed the original program.

```
% myprog.pixie
```

This program generates a list of basic block counts in files named *myprog.Counts*. If the program executes a *fork/sproc*, a process ID is appended to the end of the filename (for example, *myprog.Counts.345*) for each process.

5. Run the profile formatting program *prof* specifying the *-pixie* option and the name of the original program.

```
% prof -pixie myprog myprog.Counts
```

prof extracts information from *myprog.Counts* and prints it in an easily readable format. If multiple *.Counts* files exist, you can use the wildcard character (*) to specify all of the files.

```
% prof -pixie myprog myprog.Counts*
```

Note: Specifying *myprog.Counts* is optional; *prof* searches by default for names having the form *prog_name.Counts*.

You can run the program several times, altering the input data, to create multiple profile data files. See “Example Using *prof -pixie -procedures -clock*” later in this section for an example.

The time computation assumes a “best case” execution; actual execution may take longer. This is because the time includes predicted stalls within a basic block, but not actual stalls that may occur entering a basic block. Also it assumes that all instructions and data are in cache (for example, it excludes the delays due to cache misses and memory fetches and stores).

The complete output of the *-pixie* option is often extremely large. Use the *-quit* option with *prof* to restrict the size of the output. Refer to “Running the Profiler” for details about *prof* options.

Include or exclude information on specific procedures in your program by using the *prof* options *-only* or *-exclude* (explained in Table 4-1). *prof* timings reflect only time spent in a specific procedure, not time spent including procedures called by that procedure. The *CASEVision/WorkShop* toolset, an optional software product, can show an estimate of inclusive times.

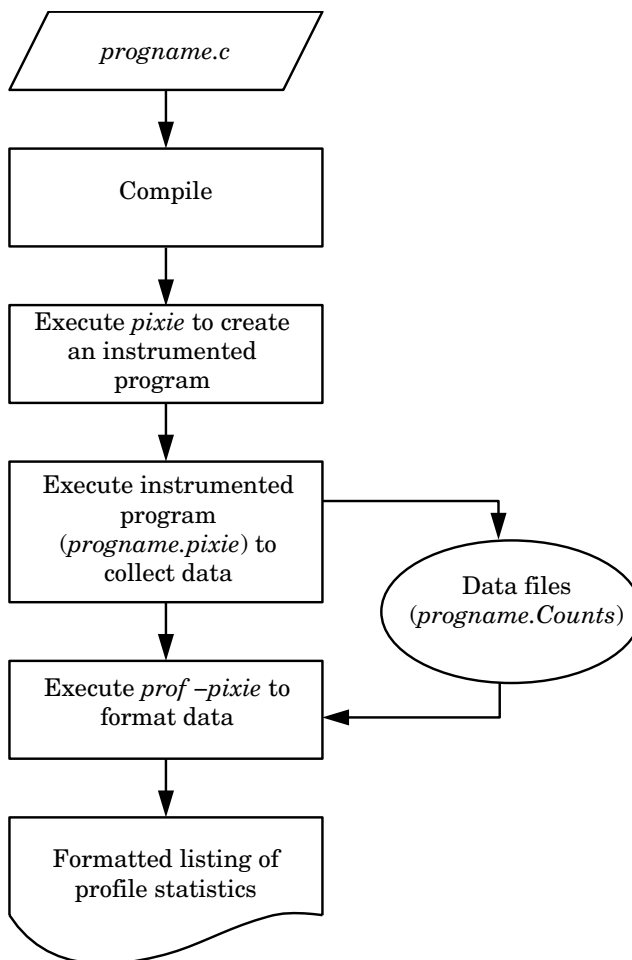


Figure 4-2 How Basic Block Counting Works

Examples of Basic Block Counting

The examples in this section illustrate how to use *prof -pixie* to obtain basic block counting information from a profiled version of a C file, *espresso*.

Example Using `prof -pixie -i` invocations

The partial listing below illustrates the use of the `-i[invocations]` option. For each procedure, `prof` reports the number of times it was invoked from each of its possible callers and lists the procedure(s) that called it.

```
% prof -pixie -i espresso
-----
Profile listing generated Fri May 13 14:25:19 1994
with:      prof -pixie -i espresso
...
*   Sorted in descending order by number of calls per procedure.
*   Unexecuted procedures are excluded.
*   The inst column is a static number of instructions.
*   %coverage column contains the percent inst executed.
-----
Total procedure invocations: 12113082

      calls(%)   cum%      inst %coverage procedure (file)
-----
    3055229(25.22)  25.22      26   25.00 full_row
(espresso:/usr/people/guest/enjoy/008.espresso/setc.c)
     966541( 7.98)  33.20      26   25.00 set_or
(espresso:/usr/people/guest/enjoy/008.espresso/set.c)
     772942( 6.38)  39.58      26   25.00 cleanfree
(espresso:/work/irix/lib/libc/gen/malloc.c)
     611793( 5.05)  44.63      26   25.00 setp_implies
...

```

The above listing shows the total procedure invocations (calls) during the run: 12113082.

- The `calls(%)` column reports the number of calls (and the percentage of total calls) per procedure. For example, there were 3055229 calls (or 25.22% of the total) spent in the procedure `full_row`.
- The `cum%` column shows the cumulative percentage of calls. For example, 25.22% of all calls were spent in `full_row`.
- The `inst` column shows the number of instructions executed for a procedure. For example, there were 26 instructions in the procedure `full_row`.
- The `%coverage` column reports the percentage of instructions executed. For example, 25.00% of instructions were executed in `full_row`.

- The `procedure (file)` column lists the procedure and its file. For example, the first line reports statistics for the procedure `full_row` in the file `setc.c`.

Example Using `prof -pixie -heavy`

The following partial listing shows the source code lines responsible for the largest portion of execution time produced with the `-heavy` option.

```
% prof -pixie -heavy espresso
```

```
-----
Profile listing generated Fri May 13 14:28:56 1994
with:      prof -pixie -heavy espresso
-----
```

```
...
* -h[eavy] using basic block counts.
*   Sorted in descending order by number of cycles per line.
*   Unexecuted lines are excluded.
*   The insts column contains distinct executed instructions for this line.
-----
```

```

          cycles(%)  cum %  line insts procedure (file)
77948528( 4.95%)   4.95%   57   40 cofactor
(espresso:/usr/people/guest/enjoy/008.espresso/cofactor.c)
73800963( 4.69%)   9.65%   213  67 essen_parts
(espresso:/usr/people/guest/enjoy/008.espresso/expand.c)
53399667( 3.39%)  13.04%   48   29 full_row
(espresso:/usr/people/guest/enjoy/008.espresso/setc.c)
44723520( 2.84%)  15.88%   137  22 massive_count
(espresso:/usr/people/guest/enjoy/008.espresso/cofactor.c)
38032848( 2.42%)  18.30%   257  39 taut_special_cases
(espresso:/usr/people/guest/enjoy/008.espresso/irred.c)
...

```

The previous partial listing shows basic block counts sorted in descending order. The most heavily used line (57) was in procedure `cofactor`.

- The `cycles(%)` column shows the total number of program cycles (and percentage of the total number). For example, there were 77948528 cycles (4.95% of the total number) for the procedure `cofactor`.
- The `cum%` column shows the cumulative percentage of the total program cycles. For example, 4.95% of all program cycles were spent in `cofactor`. The first three procedures used 13.04% of the total cycles.

- The `line` column lists the line number of the procedure: 57.
- The `insts` column reports the number of distinct instructions that were executed at least once. For example, line 57 had 40 instructions.

Example Using `prof -pixie -lines`

The following partial listing, produced using the `-lines` option, shows the execution time spent on each line of code, grouped by procedure.

```
% prof -pixie -lines espresso
-----
Profile listing generated Fri May 13 14:30:49 1994
with:      prof -pixie -lines espresso
-----
...
* -l[ines] using basic-block counts.
*   Grouped by procedure.
*   Major sort on cycles executed per procedure.
*   Minor sort on line numbers within procedure.
*   Unexecuted procedures are excluded.
-----
      cycles(%)  cum %  line insts procedure (file)
      856768( 0.05%)  0.05%  121  12 massive_count
(espresso:/usr/people/guest/enjoy/008.espresso/cofactor.c)
      25235712( 1.60%)  1.66%  128  12 massive_count
(espresso:/usr/people/guest/enjoy/008.espresso/cofactor.c)
      934656( 0.06%)  1.72%  134  16 massive_count
(espresso:/usr/people/guest/enjoy/008.espresso/cofactor.c)
      5963136( 0.38%)  2.10%  135   7 massive_count
(espresso:/usr/people/guest/enjoy/008.espresso/cofactor.c)
      20870976( 1.33%)  3.42%  136  13 massive_count
(espresso:/usr/people/guest/enjoy/008.espresso/cofactor.c)
...

```

In the above listing:

- The `cycles(%)` column lists the number of program cycles (and the percentage of the total cycles) for each procedure. For example, there were 856768 program cycles (0.05% of the total) for `massive_count`.
- The `cum%` column shows the cumulative percentage of the total program cycles. For example, 0.05% of all program cycles were spent in `massive_count`.

- The `line` and `insts` columns report the procedure's line number, and number of distinct instructions. For example, for the procedure `massive_count`, lines 121 and 128 each generated 12 instructions that were executed at least once, and line 134 generated 16 instructions that were executed at least once.

Example Using `prof -pixie -quit`

You can limit the output of `prof` to information on only the most time-consuming parts of the program by specifying the `-quit` option. You can instruct `prof` to quit after a particular number of lines of output, after listing the elements consuming more than a certain percentage of the total, or after the portion of each listing whose cumulative use is a certain amount.

Consider the following sample listing:

```

cycles(%)  cum %  seconds  cycles bytes procedure(file)
           /call /line

360331656(22.90)  22.90    4.80    4626   93
massive_count(espreso:/usr/people/guest/enjoy/008.espresso/cofactor.c)
174376925(11.08)  33.99    2.33   15479  108
cofactor(espreso:/usr/people/guest/enjoy/008.espresso/cofactor.c)
157700711(10.02)  44.01    2.10   43817  123
elim_lowering(espreso:/usr/people/guest/enjoy/008.espresso/expand.c)
155670642( 9.89)  53.91    2.08   49216  156
essen_parts(espreso:/usr/people/guest/enjoy/008.espresso/expand.c)
66835754( 4.25)  58.15    0.89    691   76
scofactor(espreso:/usr/people/guest/enjoy/008.espresso/cofactor.c)
66537017( 4.23)  62.38    0.89     21  156
full_row(espreso:/usr/people/guest/enjoy/008.espresso/setc.c)
57747597( 3.67)  66.05    0.77   1670   87
taut_special_cases(espreso:/usr/people/guest/enjoy/008.espresso/irred.c)

```

Any one of the following commands eliminates everything from the line starting with 66835754 to the end of the listing:

```

prof -quit 4
prof -quit 5%
prof -quit 53cum%

```

Example Using prof -pixie -procedures

The following partial listing, produced with the `-procedures` option, shows the percentage of execution time spent in each procedure.

```
% prof -pixie -procedures espresso
```

```
-----
Profile listing generated Fri May 13 14:33:00 1994
with:      prof -pixie -procedures espresso
-----
...
* -p[rocedures] using basic-block counts.
*   Sorted in descending order by the number of cycles executed in each
*   procedure. Unexecuted procedures are not listed.
-----

      cycles(%)  cum %  seconds  cycles bytes procedure(file)
                    /call /line

360331656(22.90)  22.90    4.80    4626   93
massive_count(espresso:/usr/people/guest/enjoy/008.espresso/cofactor.c)
174376925(11.08)  33.99    2.33   15479  108
cofactor(espresso:/usr/people/guest/enjoy/008.espresso/cofactor.c)
157700711(10.02)  44.01    2.10   43817  123
elim_lowering(espresso:/usr/people/guest/enjoy/008.espresso/expand.c)
155670642( 9.89)  53.91    2.08   49216  156
essen_parts(espresso:/usr/people/guest/enjoy/008.espresso/expand.c)
66835754( 4.25)  58.15    0.89    691   76
scofactor(espresso:/usr/people/guest/enjoy/008.espresso/cofactor.c)
```

In the above listing:

- The `cycles(%)` column lists the number of program cycles (and percentage of the total) used. For example, `massive_count` used 360331656 program cycles (22.90% of the total cycles).
- The `cum%` column reports the cumulative total of all cycles used. For example, `massive_count`, `cofactor`, and `elim_lowering` used 44.01% of the cycles.
- The `seconds` column lists the time used by the procedure (the clock rate, 75 megahertz, is omitted in this example). For example, there were 4.80 seconds used by `massive_count`.
- The `cycles/call` and `bytes/line` columns report the average cycles per call, and the bytes of code per line of source text, respectively. For

example, `massive_count` used an average of 4626 cycles per call, and had 93 bytes of generated code per line of source text.

- The `procedure (file)` column lists the procedure, `massive_count`, and its source file, `cofactor.c`.

Example Using `prof -pixie -procedures -clock`

You can add absolute time information to the output by specifying the clock rate, in megahertz, with the `-clock` option. Partial output looks like this:

```
% prof -pixie -procedures -clock 20 espresso
-----
Profile listing generated Fri May 13 14:34:55 1994
with:      prof -pixie -procedures -clock 20 espresso
-----
...
* -p[rocedures] using basic-block counts.
*   Sorted in descending order by the number of cycles executed in each
*   procedure. Unexecuted procedures are not listed.
-----

      cycles(%)  cum %  seconds  cycles bytes procedure(file)
                        /call /line

360331656(22.90)  22.90   18.02   4626   93
massive_count(espresso:/usr/people/guest/enjoy/008.espresso/cofactor.c)
174376925(11.08)  33.99    8.72  15479  108
cofactor(espresso:/usr/people/guest/enjoy/008.espresso/cofactor.c)
157700711(10.02)  44.01    7.89  43817  123
elim_lowering(espresso:/usr/people/guest/enjoy/008.espresso/expand.c)
...

```

In the previous listing, the `seconds` column lists the number of seconds spent in each procedure. For example 18.02 seconds were spent in the procedure `massive_count`. The time (computed by the profiler), in seconds, is based on the machine speed specified with the `-clock` option (in megahertz). In this example, the speed specified is 20 megahertz.

Summing Basic Block Count Results

Sometimes a single run of a program does not produce the results you require. You can repeatedly run the version of a program created by *pixie* and vary the input with each run, then use the resulting *.Counts* files to produce a consolidated report.

Use the following procedure to average *prof* results:

1. Compile and link the input file. Do not use the *-p* option. For example, consider the input file *myprog.c*:

```
% cc -o myprog myprog.c
```

The *cc* compiler compiles *myprog.c* and saves the executable as *myprog*.

2. Run the profiling program *pixie*.

```
% pixie myprog -pids
```

pixie generates the modified program *myprog.pixie*.

3. Run the profiled program as many times as desired. Each time you run the program, *pixie* creates a *myprog.Counts.pid* file, with the process ID appended.

```
% myprog.pixie < input1 > output1
```

```
% myprog.pixie < input2 > output2
```

```
% myprog.pixie < input3 > output3
```

4. Create the report.

```
% prof -pixie myprog myprog.Counts*
```

prof sums the basic block data in the *.Counts* files to produce the profile report.

Using *pixstats*

Use *pixstats(1)* to get more architectural details of a program's execution than are available from *prof*. The *-op* option to *pixstats* produces low-level information on bus issue, various kinds of stalls that *prof* doesn't provide. *Prof* also requires more memory to operate, so in situations where not enough memory exists for *prof* to function correctly, you can use *pixstats*.

You can also use *pixstats* to look for write buffer stalls and to produce disassembled code listings.

Note: In subsequent releases, *pixstats* will be removed and its functionality will be moved into *prof*.

The disadvantages to using *pixstats* are that it:

- Does not provide a line-by-line count
- Profiles only one *.Counts* file at a time (no averaging)
- Provides very little documentation
- Does not show time spent in floating point exceptions

***pixstats* Syntax**

The syntax for *pixstats* is:

```
pixstats program [options]
```

program Specifies the name of the program to be analyzed.

options One of the keywords shown in Table 4-4.

***pixstats* Options**

Table 4-4 lists and briefly describes *pixstats* options. For details, see the *pixstats(1)* reference page.

Table 4-4 Options for *pixstats*

Name	Result
<i>-cycle ns</i>	Assumes an <i>ns</i> cycle time when converting cycle counts to seconds.
<i>-dis</i>	Disassembles and annotates the analyzed object code.
<i>-dso [dso_name]</i>	Analyzes only the named DSO(s).
<i>-only procedure_name</i>	Analyzes only the named procedure(s).

Table 4-4 (continued) Options for *pixstats*

Name	Result
<i>-op</i>	Produces a detailed listing about instructions and operations and procedure usage. Information includes instruction distribution, stall distribution, basic block size distribution, and register usage.
<i>-r2010</i>	Uses r2010 floating point chip operation times and overlap rules. This option is the default.
<i>-r2360</i>	Uses r2360 floating point board operation times and overlap rules.
<i>-r4000</i>	Uses the r4000 operation times and overlap rules. This is the default if the program's magic number indicates it is a mips2 or mips3 executable.

Other options are explained in the *pixstats(1)* reference page.

Use the following procedure to run *pixstats*:

1. Compile and link the input file *myprog.c*. Do not use the *-p* option. For example, using the input file *myprog.c*:

```
% cc -c myprog.c
% cc -o myprog myprog.o
```

The *cc* compiler driver compiles *myprog.c* and saves the object file as *myprog.o*. The second command links *myprog.o* and saves the executable as *myprog*.

2. Run the profiling program *pixie*.

```
% pixie myprog
```

pixie generates the modified program *myprog.pixie*.

3. Set the path, so *pixie* knows where to find the *.pixie* files.

```
% setenv LD_LIBRARY_PATH .
```

4. Execute the file generated by *pixie*, *myprog.pixie*, in the same way you would execute the original program.

```
% myprog.pixie
```


This program generates the file *myprog.Counts* which contains the basic block counts.

- Run *pixstats* to generate a detailed report.

```
% pixstats myprog
```

Examples Using *pixstats*

The following example shows the default listing generated by *pixstats*:

```
pixstats espresso:
```

```
1588254395 (1.357/inst) cycles (15.88s @ 100.0MHz)
```

```
1170355761 (1.000/inst) instructions
```

```
2397 (0.000/inst) floating point ops (0.000151 MFLOPS @ 100.0MHz)
```

```
Procedures ordered by execution time:
```

	cycles	%cycles	cum%	instrs	cycles /inst	calls	cycles procedure /call
	382093989	24.1%	24.1%	278174631	1.4	77888	4906 massive_count
	194452825	12.2%	36.3%	130750578	1.5	11265	17262 cofactor
	146765915	9.2%	45.5%	104525532	1.4	3599	40780 elim_lowering
	144704109	9.1%	54.7%	113501194	1.3	3163	45749 essen_parts
	65043668	4.1%	58.7%	51198838	1.3	96713	673 scofactor
	57256404	3.6%	62.4%	41920736	1.4	34564	1657 taut_special_cases
	54258762	3.4%	65.8%	43594130	1.2	1632626	33 full_row
	43947988	2.8%	68.5%	32692126	1.3	72095	610 sccc_special_cases
	42611632	2.7%	71.2%	27971390	1.5	2370	17980 setup_BB_CC
	35769668	2.3%	73.5%	26776245	1.3	528962	68 __malloc
	29107500	1.8%	75.3%	24012582	1.2	333396	87 cdist01
	28840766	1.8%	77.1%	23333068	1.2	235410	123 force_lower
	27458158	1.7%	78.8%	21150937	1.3	447933	61 realfree
	26682338	1.7%	80.5%	21303304	1.3	407124	66 cleanfree
	21207623	1.3%	81.9%	16338599	1.3	528945	40 __free
	19991078	1.3%	83.1%	13678106	1.5	526081	38 __malloc
	19464960	1.2%	84.3%	13152000	1.5	526080	37 __free
	17434271	1.1%	85.4%	15501189	1.1	485880	36 set_or
	14574313	0.9%	86.4%	9466949	1.5	725	20103 expandl_gasp
	13115606	0.8%	87.2%	9753942	1.3	336135	39 __smallloc
	10646822	0.7%	87.9%	7928278	1.3	316901	34 setp_implies
	9812911	0.6%	88.5%	7858695	1.2	50103	196 binate_split_select
	9487312	0.6%	89.1%	7085744	1.3	908	10449 compl_lift
	9342972	0.6%	89.7%	6531388	1.4	567	16478 cb_consensus
	8825424	0.6%	90.2%	7330700	1.2	55166	160 consensus

8353958	0.5%	90.7%	7474594	1.1	219841	38	set_diff
7858360	0.5%	91.2%	6439566	1.2	32735	240	cb_consensus_dist0
7670120	0.5%	91.7%	5122330	1.5	72095	106	sccc
7606139	0.5%	92.2%	6589200	1.2	66552	114	cactive
6833384	0.4%	92.6%	4754352	1.4	105270	65	t_delete
6561606	0.4%	93.0%	4923904	1.3	274750	24	set_clear
6065768	0.4%	93.4%	4007453	1.5	44414	137	sm_insert
5738712	0.4%	93.8%	4397726	1.3	95491	60	sf_addset
5477719	0.3%	94.1%	3639951	1.5	117592	47	t_splay
5162590	0.3%	94.5%	3351656	1.5	789	6543	essen_raising
5134827	0.3%	94.8%	3968016	1.3	1	5134827	rm_contain
4737968	0.3%	95.1%	4006370	1.2	34838	136	sccc_merge
4611671	0.3%	95.4%	3079225	1.5	2120	2175	intcopy
3868020	0.2%	95.6%	3438240	1.1	107445	36	set_and
3862752	0.2%	95.9%	3338484	1.2	66552	58	sccc_cube

In the above listing, the first line shows the total cycles used and the second line shows the total instructions. The third line shows the number of floating point operations.

You can use `pixstats -op` to generate a detailed listing about instructions and operations. Information includes instruction distribution, stall distribution, basic block size distribution, and register usage.

The following example shows a partial listing generated by executing `pixstats -op` on the C file, `espresso`.

```
% pixstats -op espresso
1588254395 (1.357/inst) cycles (15.88s @ 100.0MHz)
1170355761 (1.000/inst) instructions
  12892539 (0.011/inst) instructions annulled by untaken branch likely
  250767706 (0.214/inst) cycles lost on non-sequential fetch
  152077341 (0.130/inst) alu/shift/load result interlock cycles
    689848 (0.001/inst) multiply interlock cycles (12 cycles)
    692925 (0.001/inst) divide interlock cycles (75 cycles)
    513369 (0.000/inst) variable shift extra issue cycles (2 total issue)
    29852 (0.000/inst) floating point result interlock cycles
      48 (0.000/inst) floating point compare interlock cycles
    71323 (0.000/inst) interlock cycles due to basic block boundary

  42595822 (0.036/inst) nops
  570540263 (0.487/inst) alu (including logicals, shifts, traps)
  185728252 (0.159/inst) logicals (including moves and li)
  72264550 (0.062/inst) shifts
```

```

294103332 (0.251/inst) loads
  70309957 (0.060/inst) stores
364413289 (0.311/inst) loads+stores
120292374 (0.103/inst) load followed by load
364415843 (0.311/inst) data bus use
  77171 (0.000/inst) partial word references
117123216 (0.100/inst) sp+gp load/stores

    2397 (0.000/inst) floating point ops (0.000151 MFLOPS @ 100.0MHz)
      48 (0.000/inst) floating point compares
        0 (0.000/inst) overlapped floating point cycles
168207521 (0.144/inst) conditional branches
  35852721 (0.031/inst) branch likelies
  32916066 (0.028/inst) load in branch delay slot
  20834011 (0.018/inst) branch to branch
  13630671 (0.012/inst) branch to branch taken
    5836780 (0.005/inst) branch to branch untaken
  41296177 (0.035/inst) branch delay filled with target-1 instruction
  67419786 (0.058/inst) untaken conditional branches
  100787735 (0.086/inst) taken conditional branches
  17365744 (0.015/inst) taken conditional branches with knop
    7598403 (0.006/inst) untaken conditional branches with knop
  15749049 (0.013/inst) direction-predicted conditional branches with knop
  125383991 (0.107/inst) non-sequential fetches
  236116938 (0.202/inst) basic blocks
    8578595 (0.007/inst) calls
    8594643 (0.007/inst) non-R31 JR
    3511926 (0.003/inst) addui opportunities
      0 (0.000/inst) fp multiply/add opportunities
    2220801 (0.002/inst) skip

```

You can use *pixstats* to disassemble and annotate the analyzed object code. The next example shows *pixstats -dis[assemble]*. The file, *espresso*, was executed on an R4000 CPU; results will differ on other CPUs.

```

% pixstats -dis espresso
0  43 404618 8fbc0020      lw      gp,32(sp)
^---11265 total cycles. Executed 11265 times, avg 1 (0.00107% of inst.)---^
0  43 40461c 0040f825      move   ra,v0
1  43 404620 afa20030      sw     v0,48(sp)
2  43 404624 8f858574      lw     a1,-31372(gp)
      << 2 cycle interlock >>
5  43 404628 8ca50000      lw     a1,0(a1)
      << 2 cycle interlock >>

```

```
8 46 40462c 28a10021      slti   at,a1,33
9 46 404630 10200003      beq    at,zero,0x404640
10 46 404634 24a2ffff      addiu  v0,a1,-1
      << branch taken 11265 times (100%) >>
      << possible 2 cycles branch penalty, total 22530, avg 2 >>
^---146445 total cycles. Executed 11265 times, avg 13 (0.0075% of inst.)---^
0 46 404638 10000003      b      0x404648
1 46 40463c 24020002      li     v0,2
^--- 0 total cycles. Executed 0 times, avg 4 (0% of inst.)---^
0 46 404640 00021143      sra   v0,v0,5
1 46 404644 24420002      addiu  v0,v0,2
^---22530 total cycles. Executed 11265 times, avg 2 (0.00214% of inst.)---^
```

The second line lists the total number of cycles for basic block `8fb00020`, (listed in the previous line). Line six shows a 2-cycle interlock because of a load of register `a1` (referenced in line seven).

Line 12 shows that a branch was taken 11,265 times, and that branches were taken 100% of the time. A possible branch penalty exists for every branch. Line 13 shows an average of 2 penalties occurred for a total of 22,530 penalties (a large number because the branch was always taken).

Profiling Multiprocessed Executables

You can gather either basic block and pc sampling profile data from executables that use the *sproc* and *sprocsp* system calls, such as those executables generated by POWER Fortran and POWER C. Prepare and run the job using the same method as for uniprocessed executables. For multiprocessed executables, each thread of execution writes its own separate profile data file. View these data files with *prof* like any other profile data files.

The only difference between multiprocessed and regular executables is the way in which the data files are named. When using pc sampling, the data files for multiprocessed executables are named *process_id.prog_name*. When using *pixie*, the data files are named *prog_name.Counts.process_id*. This naming convention avoids the potential conflict of multiple threads attempting to write simultaneously to the same file.

Rearranging Procedures With *cord*

The *cord*(1) command rearranges procedures in an executable object to reduce paging and achieve better instruction cache mapping. This section describes *cord* and covers the following topics:

- “*cord* Syntax”
- “*cord* Options”
- “Example Using *cord*”

***cord* Syntax**

The syntax for *cord* is:

```
cord prog_name [reorder_file ...]
```

Use *cord* to rearrange procedures in an executable to correspond with an ordering provided in a *reorder_file*. Typically, the ordering is arranged either to minimize paging and/or to maximize the instruction cache hit rate.

The reorder file is produced by the *-feedback* option to *prof* (for information on *prof* and the *-feedback* option, see Table 4-1, Options for *prof*, or the *prof*(1) reference page). The default reorder file is named *prog.fb*, if you do not specify *reorder_file*.

You can specify multiple reorder files on the command line; the first reorder file has the highest priority in rearranging the order. Thus you can improve paging in different program phases providing that multiple feedback files are generated by executing different phases of the program or by executing the program with distinct input data that cause different regions of the program to be executed.

cord Options

Table 4-5 lists and describes *cord* options. For details, refer to the *cord(1)* reference page.

Table 4-5 Options for *cord*

Name	Result
<code>-o out_file</code>	Specifies a name for the output file. The default file is <i>prog.cord</i> .
<code>-v</code>	Prints verbose information. Lists procedures considered part of other procedures that cannot be rearranged. These procedures are typically assembler procedures that may contain relative branches to other procedures rather than relocatable ones. Lists conflicts of procedures in the binary and the reorder files.

Example Using *cord*

The example below shows how to use *pixie*, *prof*, and *cord* to rearrange the procedures in the program *xlisp* (refer to Figure 4-3).

```
% pixie xlisp                # generates xlisp.pixie
% xlisp.pixie li-input.lsp   # generates xlisp.Counts
% prof xlisp -pixie -feedback # generates xlisp.fb and
                               # libc.so.1.fb
% cord xlisp                # generates xlisp.cord
```

First, the program *xlisp* is executed by *pixie*, which generates an instrumented executable, *xlisp.pixie*. Next, the instrumented executable is run (with an input file to *xlisp*, *li-input.lsp*). Then *prof* is used to produce feedback files from the output data. Finally, *cord* is executed (and uses the order in the feedback file) to reorder the procedures in *xlisp*, generating a new binary, *xlisp.cord*. Figure 4-3 shows this procedure.

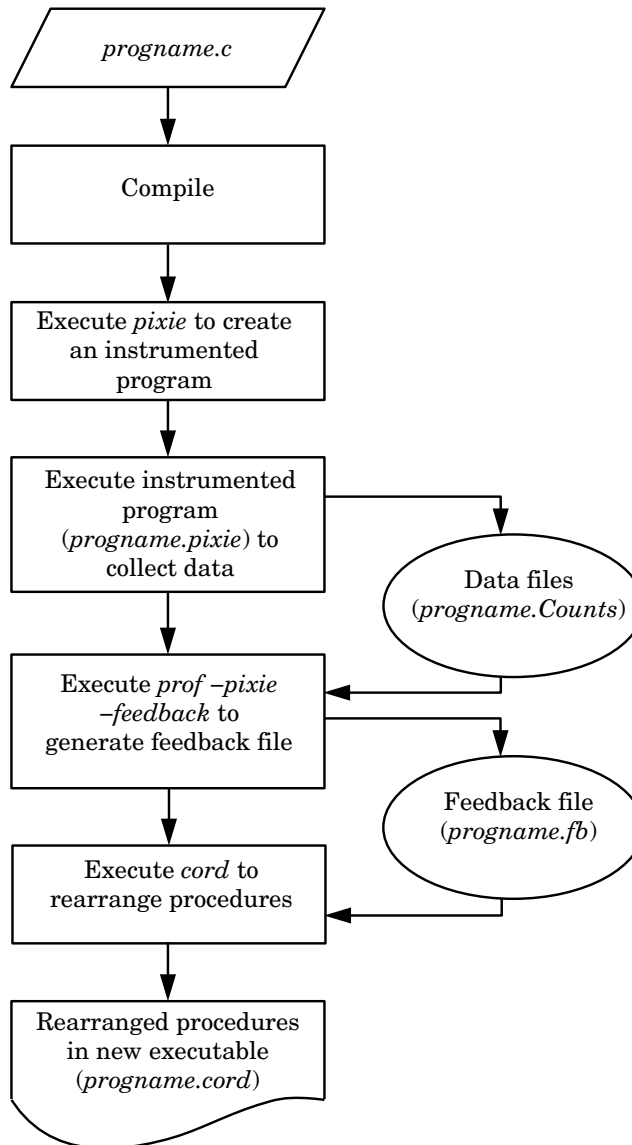


Figure 4-3 How *cord* Works

The procedure rearrangement depends on the data produced by the profiled runs of the executable. If these profiled runs approximate the actual use of

the executable, the resultant binary is close to being optimally rearranged. Design your profiled runs accordingly.

You can also manually optimize your reorder file by rearranging the procedure entries in the reorder file.

For example, after running `cord xisp -pixie -feedback`, the feedback file `xisp.fb` looks like this:

```
$magic 0x10130000
$version 2
$name xisp
$kind procedure
$start
# generated by prof -feedback
#  procedure_name      file_name      freq
      xlminit          xldmem.c      651846882
      xlxgetvalue      xlsym.c       564706014
      xlabind           xlevel.c      368782916
      xlevel            xlevel.c      360302271
      mark              xldmem.c      353045832
      xlgetvalue        xlsym.c       341400298
      xlsend            xlobj.c       306873567
      sweep             xldmem.c      232575506
      evalhook          xlevel.c      227803590
      gc                 xldmem.c      216458905
      addseg            xldmem.c      174118911
      evform            xlevel.c      161070071
      xlygetvalue        xlsym.c       133714210
      xlevlist           xlevel.c      119441482
      xlmakesym          xlsym.c       117704318
      xldinit            xldbug.c      117010681
      newvector          xldmem.c      113412102
      iskeyword          xlevel.c      105730347 ...
```

The `procedure_name` column indicates the name of the procedure and the `file_name` column lists the name of the file that contains the procedure. The `freq` column can be the number of cycles spent in the procedure, the number of times the procedure was executed, or the density (total cycles divided by the procedure size). The `cord` command places procedures based on the order specified in the feedback file and does not use frequency to determine procedure placement.

Chapter 5

Optimizing Program Performance

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

Optimizing Program Performance

This chapter describes the compiler optimization facility and how to use it. The section also contains examples demonstrating optimization techniques.

- “Optimization Overview” describes the global optimizer, benefits of optimization, and debugging and optimization
- “Using the Optimization Options” lists compiler optimization options and provides examples of full optimization
- “Loop Optimization” explains how you can decrease execution time by optimizing loops.
- “Optimizing Separate Compilation Units” covers optimization of modules
- “Optimizing Frequently Used Modules” shows how optimizing frequently used modules reduces the compile and optimization time required when the modules are called
- “Ucode Object Libraries” covers building and using *ucode* object libraries
- “Improving Global Optimization” provides tips on improving optimization with examples in Fortran and C.

Optimization Overview

This section describes the compiler optimization facilities and explains their benefits, the implications of optimizing and debugging, and the major optimizing techniques. Specifically, this section explains:

- “Global Optimizer”
- “Benefits of Optimization”
- “Optimization and Debugging”

Global Optimizer

The global optimizer is a single program that improves the performance of object programs by transforming existing code into more efficient coding sequences. The optimizer distinguishes between C, Pascal, and Fortran programs to take advantage of the various language semantics involved.

Silicon Graphics compilers perform both machine-independent and machine-dependent optimizations. Silicon Graphics machines and other machines with reduced instruction set computing (RISC) architectures provide a good target for both machine-independent and machine-dependent optimizations. The low-level instructions of RISC machines provide more optimization opportunities than the high-level instructions in complex instruction set computing (CISC) machines. Even optimizations that are machine-independent have been found to be more effective on machines with RISC architectures. Although most optimizations performed by the global optimizer are machine-independent, they have been specifically tailored to the Silicon Graphics environment.

Benefits of Optimization

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

Optimization and Debugging

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

Using the Optimization Options

This section lists compiler options you can use for optimization and provides examples of full optimization. Specifically, this section covers:

- “Compiler Optimization Options”
- “Examples of Full Optimization”

Compiler Optimization Options

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

Table 5-1 Optimization Options

Option	Result
<code>-O0</code>	No optimization. Prevents all optimizations, including the minimal optimization normally performed by the code generator and assembler. <code>uld</code> , <code>umerge</code> , and <code>uopt</code> are bypassed, and the assembler bypasses certain optimizations it normally performs.
<code>-O1</code>	(Default) The assembler and code generator perform as many optimizations as possible without affecting compile time performance. Bypasses <code>uld</code> , <code>umerge</code> , and <code>uopt</code> . However, the code generator and the assembler perform basic optimizations in a more limited scope.

Table 5-1 (continued) Optimization Options

Option	Result
-O2	Specifies global optimization. Optimizes within the bounds of individual compilation units. This option executes the global optimizer (<i>uopt</i>) phase. <i>uld</i> and <i>umerge</i> are bypassed, and only the <i>uopt</i> phase executes. It performs optimization only within the bounds of individual compilation units.
-O3	Specifies using all optimizations, including procedure inlining. This option must precede all source file arguments. It creates a ucode object file, which remains a <i>.u</i> file, for each source file. The run-time start-up routine, run-time libraries, and ucode versions of the run-time libraries are linked, as well as newly created ucode object files and any ucode object files specified on the command line. Procedure inlining is done on the resulting linked file. This file is then compiled as usual into an executable.

The *uld* and *umerge* phases process the output from the compilation phase of the compiler, which produces symbol table information and the program text in an internal format called *ucode*. The *uld* phase combines all the ucode files and symbol tables, and passes control to *umerge*. *umerge* reorders the ucode for optimal processing by *uopt*. Upon completion, *umerge* passes control to *uopt*, which performs global optimizations on the program.

Note: Refer to the applicable *cc(1)*, *CC(1)*, *pc(1)*, or *f77(1)* reference pages for details on the -O3 option and the input/output files related to this option.

Figure 5-1 shows the major processing phases of the compiler and how the compiler -On option determines the execution sequence.

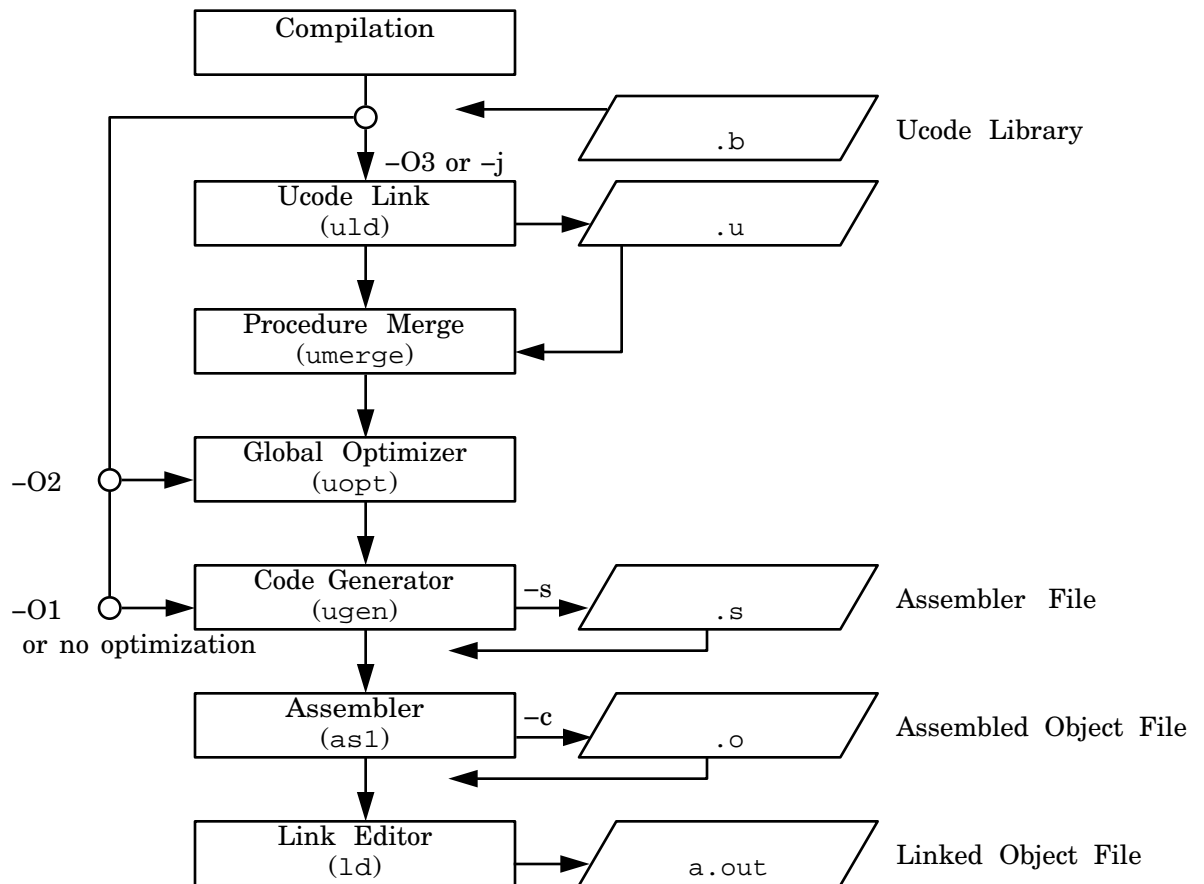


Figure 5-1 Optimization Phases of the Compiler

Examples of Full Optimization

This section provides examples of full optimization using the `-O3` option. Although the examples are in C, you can substitute the C files and driver command for another source language. The following examples assume that the program *foo* consists of three files: *a.c*, *b.c*, and *c.c*.

To perform procedure merging optimizations (`-O3`) on all three files, enter the following:

```
IRIS% cc -O3 -non_shared -o foo a.c b.c c.c
```

If you normally use the `-c` option to compile the `.o` object file, follow these steps:

1. Compile each file separately using the `-j` option by typing in the following:

```
IRIS% cc -j a.c
```

```
IRIS% cc -j b.c
```

```
IRIS% cc -j c.c
```

The `-j` option produces a `.u` file (the standard compiler front-end output made up of ucode; ucode is an internal language used by the compiler). None of the remaining compiling phases are executed, as illustrated in Figure 5-2.

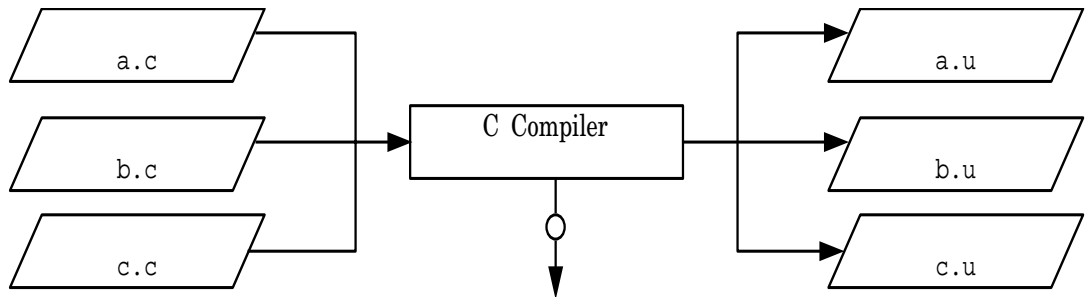


Figure 5-2 Compiling with the `-j` Option

2. Enter the following statement to perform optimization and complete the compilation process.

```
IRIS% cc -O3 -non_shared -o foo a.u b.u c.u
```

Figure 5-3 illustrates the results of executing the above statement.

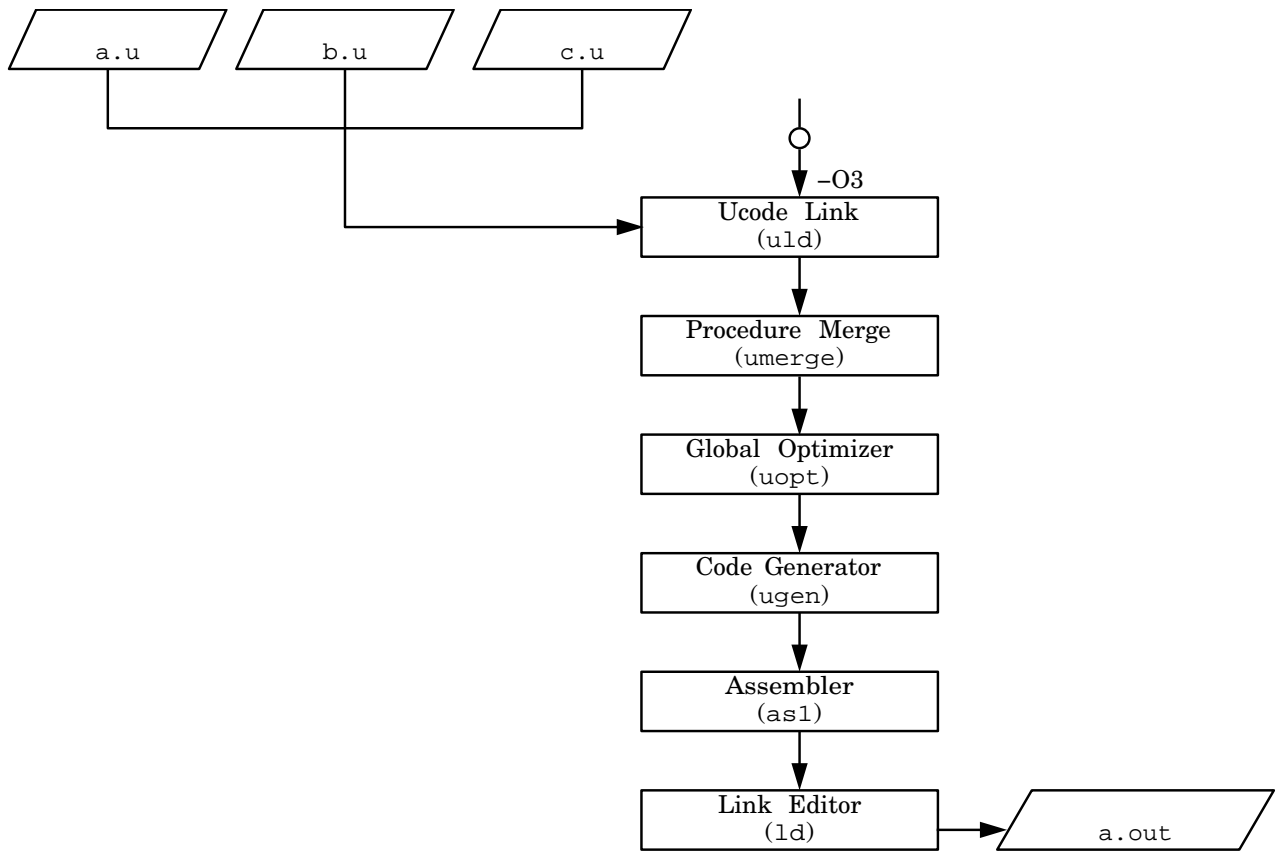


Figure 5-3 Executing Full Optimization

Loop Optimization

Optimizations are most useful in program areas that contain loops. The optimizer moves loop-invariant code sequences outside loops so that they are performed only once instead of multiple times. Apart from loop-invariant code, loops often contain loop-induction expressions that can be replaced with simple increments. In programs composed of mostly loops, global optimization can often reduce the running time by half.

Consider the source code below.

```
void left (a, distance)
    char a[];
    int distance;
{
    int j, length;
    length = strlen(a) - distance;
    for (j = 0; j < length; j++)
        a[j] = a[j + distance];
}
```

The following code samples show the unoptimized and optimized code produced by the compiler. The optimized version (compiled with the `-O` option) contains fewer total instructions and fewer instructions that reference memory. Wherever possible, the optimizer replaces load and store instructions (which reference memory) with the faster computational instructions that perform operations only in registers.

Unoptimized Code

The loop is 13 instructions long and uses eight memory references.

```
# 8          for (j=0; j<length; j++)
            sw  $0, 36($sp) # j = 0
            ble $24, 0, $33 # length >= j
$32:
# 9          a[j] = a[j+distance];
            lw  $25, 36($sp) # j
            lw  $8, 44($sp) # distance
            addu $9, $25, $8 # j+ distance
            lw  $10, 40(4sp) # address of a
            addu $11, $10, $25 # address of a[j+distance]
            lbu $12, 0($11) # a[j+distance]
            addu $13, $10, $25 # address of a[j]
            sb  $12, 0($13) # a[j]
            lw  $14, 36($sp) # j
            addu $15, $14, 1 # j+1
            sw  $15, 36($sp) # j++
            lw  $3, 32($sp) # length
            blt $15, $3, $32 # j < length
$33:
```

Optimized Code

The loop is six instructions long and uses two memory references.

```
# 8      for (j=0; j<length; j++)
        move  $5,$0      # j = 0
        ble  $4, 0, $33  # length >= j
        move  $2, $16     # address of a[j]
        addu $6, $16, $1  # address of a[j+distance]
$32:
# 9      a[j] = a[j+distance];
        lbu  $3, 0($6)    # a[j+distance]
        sb   $3, 0($2)    # a[j]
        addu $5, $5, 1    # j++
        addu $2, $2, 1    # address of next a[j]
        addu $6, $6, 1    # address of next a[j+distance]
        blt  $5, $4, $32  # j < length
$33:     # address of next a[j+distance]
```

Loop Unrolling

The optimizer performs loop unrolling to improve performance in two ways:

- Reduces loop overhead.
- Increases work performed in the larger loop body allowing more opportunity for optimization and register usage.

For example, the Fortran loop:

```
do i=1,100
  sum = sum + a(i)*b(i)
enddo
```

when unrolled four times looks like

```
do i=1,100,4
  sum = sum + a(i)*b(i)
  sum = sum + a(i+1)*b(i+1)
  sum = sum + a(i+2)*b(i+2)
  sum = sum + a(i+3)*b(i+3)
enddo
```

The unrolled version runs much faster than the original. Most of the increase in execution speed is due to the overlapping of multiplication and addition operations. The optimizer performs this transformation on its own internal representation of the program, not by rewriting the original source code.

Note: If the number of iterations of the loop is not an exact multiple of the unrolling factor (or if the number of iterations is unknown), the optimizer still performs this transformation even though the resultant code is more complicated than the original code.

Optimizing Separate Compilation Units

The *uld* and *umerge* phases of the compiler permit global optimization among code from separate files (or “modules”) in the same compilation. Traditionally, program modularity restricted the optimization of code to a single compilation unit at a time rather than over the full breadth of the program. For example, it was impossible to fully optimize calling code along with the procedures called if those procedures resided in other modules.

The *uld* and *umerge* phases of the compiler system overcome this deficiency. The *uld* phase links multiple modules into a single unit. Then, *umerge* orders the procedures for optimal processing by the global optimizer, *uopt*.

Optimizing Frequently Used Modules

Compiling and optimizing frequently used modules reduces the compile and optimization time required when the modules are called.

The following procedure explains how to compile two frequently used modules, *b.c* and *c.c*, while retaining all the necessary information to link them with future programs; *future.c* represents one such program.

1. Compile *b.c* and *c.c* separately by entering the following statements:

```
IRIS% cc -j b.c
```

```
IRIS% cc -j c.c
```

The *-j* option causes the front end (first phase) of the compiler to produce two ucode files: *b.u* and *c.u*.

2. Using an editor, create a file containing the external symbols in *b.c* and *c.c* to which *future.c* will refer. Each symbolic name must be separated by at least one blank. Consider the skeletal contents of *b.c* and *c.c*:

File <i>b.c</i>	File <i>c.c</i>
foo() { . . }	x() { . . }
bar() { . . zot() { . . }	help() { . . }
struct { . . } work;	struct { . . } ddata;
	y() { . . }

In this example, *future.c* calls or references only *foo*, *bar*, *x*, *ddata*, and *y* in the *b.c* and *c.c* procedures. A file (named *extern* for this example) must be created containing the following symbolic names:

```
foo bar x ddata y
```

The structure *work*, and the procedures *help* and *zot* are used internally only by *b.c* and *c.c*, and thus are not included in *extern*.

If you omit an external symbolic name, an error message is generated (see Step 4 below).

3. Optimize the *b.u* and *c.u* modules (created in Step 1) using the *extern* file (created in Step 2) as follows:

```
IRIS% cc -O3 -non_shared -rm_dead_code -kp extern b.u c.u  
-o keep.o -c
```

The `-rm_dead_code` option tells the linker to assume names not specified in the `extern` file as internal names. In the `-kp` option, `k` indicates that the linker option `-p` is to be passed to the ucode loader. The `-c` option suppresses the `a.out` file.

Figure 5-4 illustrates the optimization process in Step 3.

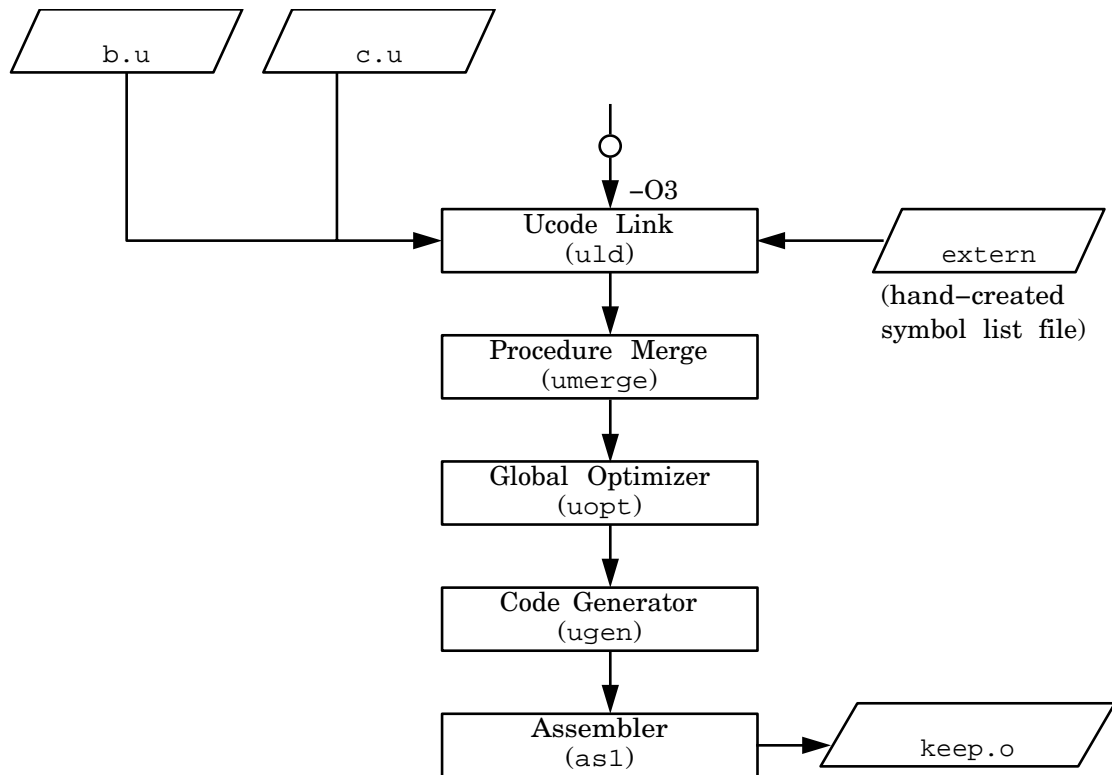


Figure 5-4 Optimization Process

4. Create a ucode file and an optimized object code file (`foo`) for `future.c` as follows:

```
IRIS% cc -j future.c
```

```
IRIS% cc -O3 -non_shared future.u keep.o -o foo
```

If the following message appears it means that the code in *future.c* is using a symbol from the code in *b.c* or *c.c* that was not specified in the file *extern* (go to Step 5 if this message appears.)

```
zot: multiply defined hidden external (should have been preserved)
```

5. Include *zot*, which the message indicates is missing, in the file *extern* and recompile as follows:

```
IRIS% cc -O3 -non_shared -kp extern b.u c.u -o keep.o
IRIS% cc -O3 -non_shared future.u keep.o -o foo
```

Ucode Object Libraries

This section describes how to build and use ucode object libraries.

Building Ucode Object Libraries

Building ucode object libraries is similar to building *coff(5)* object libraries. First, compile the source files into ucode object files using the compiler driver option *-j* and using the archiver just as you would for *coff* object libraries.

Using the above example, to build a ucode library (*libfoo.b*) of a source file, enter the following:

```
IRIS% cc -j a.c
IRIS% cc -j b.c
IRIS% cc -j c.c
IRIS% ar crs libfoo.b a.u b.u c.u
```

Conventional names exist for ucode object libraries (*libname.b*) just as they do for *coff* object libraries (*libname.a*).

Using Ucode Object Libraries

Using ucode object libraries is similar to using *coff*(5) object files. To load from a ucode library, specify a *-kname* option to the compiler driver or the ucode loader. For example, to load the file created in the previous example from the ucode library (assuming *libfoo.a* was placed in the */usr/lib* directory), enter the following:

```
IRIS% cc -O3 -non_shared file1.u file2.u -klfoo -o output
```

Remember that libraries are searched as they are encountered on the command line, so the order in which you specify them is important. If a library is made from both assembly and high-level language routines, the ucode object library contains code only for the high-level language routines. The library does not contain all the routines, as does a *coff* object library or a DSO. In this case, specify to the ucode loader first the ucode object library and then the *coff* object library or DSO to ensure that all modules are loaded from the proper library.

If the compiler driver is to perform both a ucode load step and a final load step, the object file created after the ucode load step is placed in the position of the first ucode file specified or created on the command line in the final load step.

Improving Global Optimization

This section describes coding hints that increase optimizing opportunities for the global optimizer (*uopt*). Apply these recommendations to your code whenever possible. Topics include:

- "Optimizing C and Fortran Programs"
- "Improving Other Optimization"
- "Register Allocation"

Optimizing C and Fortran Programs

The following suggestion applies to both C and Fortran programs:

Do not use indirect calls (calls that use routines or pointers to functions as arguments). Indirect calls cause unknown side effects (that is, they change global variables) that can reduce the amount of optimization possible.

C Programs Only

The following suggestions apply to C programs only:

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

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

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

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

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

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

Pointers in particular tend to cause aliasing problems, since it is often impossible for the compiler to identify their target objects. Therefore, you

can help the compiler avoid possible aliases by introducing local variables to store the values obtained from dereferenced pointers. Indirect operations and calls affect dereferenced values, but do not affect local variables. Therefore, local variables can be kept in registers. The following example shows how the proper placement of pointers and the elimination of aliasing produces better code.

Example of Pointer Placement and Aliasing

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

Consider the following source code:

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

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

The generated assembly code looks like this:

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

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

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

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

The generated assembly code looks like this:

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

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

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

```
extern int ext1;
static int stat1;

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

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

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

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

Ada® Programs

This suggestion applies to Ada programs:

Use of pragma inline. Use pragma inline to inline short subroutines and avoid the overhead associated with procedure calls.

Improving Other Optimization

The global optimizer processes programs only when you specify the `-O2` or `-O3` option at compilation. However, the code generator and assembler phases of the compiler always perform certain optimizations (certain assembler optimizations are bypassed when you specify the `-O0` option at compilation).

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

C and Fortran Programs

The following suggestions apply to both C and Fortran programs:

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

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

Instead of:

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

Use:

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

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

C Programs Only

The following suggestions apply to C programs only:

- Rely on *libc.so* functions (for example, *strcpy*, *strlen*, *strcmp*, *bcopy*, *bzero*, *memset*, and *memcpy*). These functions were carefully coded for efficiency.
- Use the *unsigned* data type for variables wherever possible (see next bulleted item for an exception to this rule, though). The compiler generates fewer instructions for multiplying and dividing unsigned numbers by a power of two. Consider the following example:

```
int i;
unsigned j;
...
return i/2 + j/2;
```

The compiler generates six instructions for the signed $i/2$ operation:

```
000000 20010002 li      r1,2
000004 0081001a div     r4,r1
000008 14200002 bne    r1,r0,0x14
00000c 00000000 nop
000010 03fe000d break   1022
000014 00001812 mflo   r3
```

The compiler generates only one instruction for the unsigned $j/2$ operation:

```
000018 0005c042 srl     r24,r5,1 # j / 2
```

In this example, $i/2$ is an expensive expression, while $j/2$ is inexpensive.

- Use a signed data type, or cast to a signed data type, for any variable which must be converted to floating-point.

```
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.

Register Allocation

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

The optimizer allocates registers for the most suitable data items, taking into account their frequency of use and their locations in the program structure. Also, the optimizer assigns values to registers in such a way as to minimize movement of values among registers during procedure invocations.

Appendix A

Position-Independent Coding in Assembly Language

This appendix describes assembler directives that support generation of PIC.

Position-Independent Coding in Assembly Language

Several new assembler directives have been added to support generation of PIC. For more information on PIC, refer to the *MIPS ABI Supplement* and the PIC coding model it describes. For information on assembly language, refer to the *MIPSpro Assembly Language Programmer's Guide*.

The assembler generates PIC if either of two things occur:

- the directive **.option pic2** appears in the assembler file
- the assembler, *as*, is invoked with the **-KPIC** argument in the absence of an explicit **.option pic0** directive in the file

Unless PIC is being generated, the other options in this section are ignored by the assembler, with the exception of **.gpword**, which becomes **.word**. Thus, you may easily use the same assembler file for generating PIC and non-PIC (i.e., non-shared) objects by *not* placing **.option pic0** or **.option pic2** in the assembler file and invoking the assembler without **-KPIC** (for non-shared) or with **-KPIC** (for PIC).

- **.option pic2**

This directive forces the assembler to mark the output object file "PIC" and activates the following directives. It overrides the command line argument. Normally, you don't need to specify this directive.

Instead, you should use the **-KPIC** or **-non_shared** options to toggle between generating PIC or non-PIC.

Even though **-KPIC** will be made the default for the high-level language drivers (such as *cc*, *f77*, and *pc*) in future releases, it will *not* be the default for assembly sources. You must explicitly specify **-KPIC** for compiling *.s* files.

- **.cpload reg**

This directive expands into three instructions that sets the **gp** register to the context pointer value for the current function. The three instructions are:

```
lui    gp, _gp_disp
addui  gp, gp, _gp_disp
addu   gp, gp, reg
```

`_gp_disp` is a reserved symbol that the linker sets to the distance between the `lui` instruction and the context pointer. This directive is required at the beginning of each subroutine that uses the **gp** register.

You must add this directive at the beginning of every procedure, with the exception of leaf-procedures that do not access any global variables, and procedures that are static (i.e., not marked **.globl** or **.extern**).

- **.cprestore offset**

This directive causes the assembler to issue:

```
sw    gp, offset(sp)
```

at the point where it appears. Additionally, it causes the assembler to emit:

```
lw    gp, offset(sp)
```

after every jump-and-link (`jal`) operation (but *not* after a branch-and-link (`bal`) operation), thereby restoring the **gp** register after function calls. The programmer is responsible for allocating the stack space for the **gp**. This space should be in the saved register area of the stack frame to remain consistent with MIPS' calling and debugger conventions.

- **.gpword local-sym**

This directive is similar to **.word** except that the relocation entry for **local-sym** has the `R_MIPS_GPREL32` type. After linkage, this results in a 32-bit value that is the distance between **local-sym** and the context pointer (that is, the **gp**). **local-sym** must be local. It is currently used for PIC switch tables.

- **.cpadd reg**

This directive adds the value of the context pointer (**gp**) to `reg`.

Examples

This following is a simplified version of the *hello world* program.

```
.option pic2
.data
.align      2
$$5:
.ascii      "hello world\x0A\x00"
.text
.align      2
main:
.set        noreorder
.cpload     $25
.set        reorder
subu        $sp, 40
sw          $31, 36($sp)
.cprestore 32
la          $4, $$5
jal         printf
move        $2, $0
lw          $31, 36($sp)
addu        $sp, 40
j           $31
```

The actual instructions generated by the assembler are:

```
lui        gp,0           #
addiu      gp,gp,0        # generated by .cpload
addu       gp,gp,t9       #
lw         a0,0(gp)       # gp-relative addressing used
lw         t9,0(gp)       # t9 is used for func. call
addiu      sp,sp,-40
sw         ra,36(sp)
sw         gp,32(sp)      # from .cprestore
jalr       ra,t9         # jal is changed to jalr
addiu      a0,a0,0
lw         ra,36(sp)
lw         gp,32(sp)      # activated by .cprestore
move       v0,zero
jr         ra
addiu      sp,sp,40
nop
```

Note: The MIPS ABI requires that register **t9** (\$25) be used for indirect function call, so **.cpload** should always use \$25. No reorder mode should also be used. Also, programmers should make sure that **t9** is dead before any function call.

If your program uses an indirect jump (**jalr**), you must also use **t9** as the jump register.

If you have an unconditional jump to an external label:

```
j    _cerror
```

you have to rewrite it into indirect jump via *t9*:

```
la   t9, _cerror
j    t9
```

If you use branch-and-link (**bal**) instruction, and if the target procedure begins with a **.cpload**, you have to specify an alternate entry point:

```
foo: .set      noreorder # callee
     .cpload  $25
     .set      reorder
$$1: ...      # alternate entry point
     ...
     j        $31      # foo returns
bar: ...      # caller
     ...
bal  $$1      # by-pass the .cpload
     ...
```

This is very important because **.cpload** assumes register \$25 contains the address of `foo`, but in this case \$25 is not set up. Note that since both `foo` and `bar` reside in the same file, they must have the same value for **gp**. So the **.cpload** instructions can be and must be bypassed. However, since `foo` can still be called from outside, the **.cpload** is still required.

Alternatively (and less efficiently), if you don't want to have an alternate entry point, you can set up register \$25 before the **bal**:

```
la   t9, foo
bal  foo
```

.gpword and **.cpadd** are used together to implement position-independent jump table (or any table of text addresses). Entries of the address table created by **.gpword** are converted into displacement from the context pointer. To get the correct text address, use **.cpadd** to add the value of **gp** back to them. Since the **gp** is updated by the run-time linker, the correct text address can be reconstructed regardless of the location of the DSO.

Index

A

accom_mp preprocessor, 2
acpp preprocessor, 2
Ada
 optimization, 130
addresses, optimization, 130
address space, 71
aliasing, optimization, 127
analyzer, parallel, 2
-ansi option, 17
a.out files, 19
archive libraries, 49
archiver. *See ar* command
ar command, 42-46
 command syntax, 43
 options, 43
as1 assembler, 2, 22
assembler, 2, 135-139
assembly language programs
 linking, 22
 position-independent coding, 135

B

bal operation, 136
basic block counting. *See profiling*
BLOCK DATA, 66
branch-and-link instruction, 138

-Bsymbolic, compiling, 66
building ucode object library, 125

C

C++
 building DSOs, 61
 compiler, 3
 language definitions, 15
 ld options, 61
cache mapping, improve, 107
cc compiler. *See drivers*
-cckr option, 17
ccom_mp preprocessor, 2
cfe preprocessor, 2
code generator, 2
COFF, 10
Common Object File Format, 10
COMMON symbols, 66
compiler drivers. *See drivers*
compiler front end, 2
compiler options. *See drivers*
compiler system
 components, 1
 overview, 1
compiling with -Bsymbolic, 66
conventions, syntax, xv
copt optimizer, 2

cord, 107-110
 command options, 108
 command syntax, 107
 example, 108
 feedback files, 107
 -*feedback* option, 108
 -*o out_file* option, 108
 -*v* option, 108
.Counts file, 91
.cpadd reg directive, 136
.cpload reg directive, 136
cpp preprocessor, 2
.cprestore offset directive, 136
.cpword local-sym directive, 136

D

data type
 signed, 132
 unsigned, 131
dbx. *See* debugging
debugging
 and include files, 14
 and optimization, 119
 driver options, 28
disassemble object file, 28
dis command, 28, 29
 command syntax, 29
 options, 30
dlclose(), 70
dLError(), 70
dlopen(), 69
dlsym(), 70
do while, optimization, 127
drivers
 as1 assembler, 22
 bypassing, 3

drivers
 C++ compiler, 3
 cc compiler, 2
 cfe preprocessor, 3
 defaults, 16
 f77 compiler, 2
 file name suffixes, 12
 input file suffixes, 12
 -*KPIC*, 11, 135
 -*nocpp*, 3
 -*non_shared*, 11
 options, 16, 135
 -*KPIC*, 135
 -*non_shared*, 135
 passing options to *ld*, 20
 pc compiler, 2
 -*v* option, 3
DSOs, 1, 10, 11, 49-73
 archive libraries, 51
 building new DSOs, 58
 C++, 61
 converting libraries, 67
 creating DSOs, 58
 dlclose(), 70
 dLError(), 70
 dlopen(), 69
 dlsym(), 70
 dynamic loading diagnostics, 70
 exporting symbols, 60
 guidelines, 53
 hiding symbols, 60
 libraries, shared, 53
 linking, 25
 loading dynamically, 69
 mmap() system call, 71
 munmap() system call, 71
 naming conventions, 59
 QuickStart, 56-58
 QuickStart registry file, 61
 registry files, 61-64
 search path, 64

DSOs

- shared libraries, 53
 - starting quickly, 56
 - unloading dynamically, 70
 - versioning, 71
- dynamic linking, 1, 10, 69
- Dynamic Shared Objects. *See* DSOs

E

- elfdump* command, 28, 31
- command syntax, 31
 - options, 31
- ELF. *See* executable and linking format
- executable and linking format, 1, 10
- exporting symbols, 60

F

- f77* compiler. *See* drivers
- fcom* preprocessor, 2
- file* command, 29, 33
- command syntax, 33
- files
- header, 13
 - include, 13
 - listing properties, 29
 - naming conventions, 12
- file type, determining, 33
- floating point data, 102
- format
- object file, 1, 10
- Fortran
- optimization, 130

G

- global offset table, 11
- global optimizer, 114-132
- g option, 18
- GOT, 11
- .gpword*, 136
- .gpword* directive, 135

H

- header files, 13
- multiple languages, 14

I

- if-then-else statements
- optimization, 130
- include files, 13
- debugging, 14
 - multiple languages, 14
- indirect
- calls, using, 127
 - function call, 138
 - jump instruction, 138
- internationalization
- C++, 15
 - multilanguage programs, 24

J

- jal* operation, 136

K

`-KPIC`

See also drivers

`-KPIC` option, 11, 18

L

ld

and assembly language programs, 22

C++, 61

command syntax, 21

DSOs, 61

dynamic linking, 1, 10

example, 22

libraries, default search path, 24

libraries, specifying, 23

link editor, 2

multilanguage programs, 26

options, 61

registry files, 61

`-shared` option, 59

`LD_BIND_NOW`, 66

`LD_LIBRARY_PATH`, 91

libdl, 69

libraries

and multilanguage programs, 24

archive, 49

global data, 54

libdl, 69

locality, 54

non-shared, converting to DSOs, 67

paging, 54

routines to exclude, 53

routines to include, 53

self-contained, 53

shared, 1, 10

shared, static, 11, 25, 49

specifying, 23

libraries

static data, 53

tuning, 54

lib.so functions

optimization, 131

linking

dynamic. *See ld*

linking. *See ld*

loader

runtime. *See rld*

loading

symbols, 60

local variables

optimization, 127

loop unrolling, 121

M

machine instructions, 28

macro preprocessors, 2

mmap() system call, 71

mon.out file. *See* profiling, pc sampling

multilanguage programs

and *ld*, 26

and libraries, 24

header files, 14

multiprocessed executables, profiling, 106

munmap() system call, 71

N

naming source files, 12

nm command, 29, 33-38

command syntax, 34

example, 36

example of undefined symbol, 27

undefined symbol, 27

– *non_shared* option, 135

O

–O0 compiler option, 115

–O1 compiler option, 115

–O2 compiler option, 116

–O3 compiler option, 116
example, 118

object code library
building, 125

object file information
disassemble, 28
format, 1, 10
listing section sizes, 29, 40
symbol table information, 29, 33
tools, 28
using *elfdump*, 28, 31
using *odump*, 29, 38

odump command, 29, 38-39
command syntax, 38

optimization
Ada, 130
addresses, 130
aliasing, 127
and debugging, 119
and loop unrolling, 121
and register allocation, 132
C, 127-132
do while, 127
example, 128
Fortran, 127, 130
frequently used modules, 122
full, 118
function return values, 127
global, 114-132
if-then-else statements, 130
libc.so functions, 131
loop, 119

optimization
machine-dependent, 114
machine-independent, 114
–O0 compiler option, 115
–O1 compiler option, 115
–O2 compiler option, 116
–O3 compiler option, 116, 118
options, 115
pointers, 127
pragma inline, 130
separate compilation units, 122
signed data types, 132
STDARG, 130
subscripts, 129
switch statements, 130
tables, 130
tips for improving, 126
unions, 127
unsigned data type, 131
value parameters, 127
VARARG, 130
variables, global vs. local, 127

optimizer, 2
copt optimizer, 2

optimizing programs
benefits, 114
debugging, 114

.option pic0 directive, 135
.option pic2 directive, 135

P

page size, 54
paging
alignment, 54
paging, reduce, 107
parallel analyzer, 2
parameters
optimization, 127

- pca* analyzer, 2
- pc* compiler. *See* drivers
- pc* sampling. *See* profiling
- pfa* analyzer, 2
- PIC. *See* position-independent code
- pixie*, 89-100
 - and *prof-clock* example, 99
 - and *prof-heavy* example, 95
 - and *prof-i* example, 94
 - and *prof-lines* example, 96
 - and *prof-pids*, 100
 - and *prof-procedures* example, 98
 - autopixie* option, 90
 - command options, 89
 - command syntax, 89
 - .Counts* file, 91, 100
 - counts* option, 89
 - examples, 90
 - liblist* option, 90
 - output size, 92
 - pids* option, 90
 - restricting output, 92
 - setting search path, 91
 - verbose* option, 90
- pixstats*, 100-106
 - command syntax, 101
 - disassemble* option, example, 105
 - example, 103
 - op* option, example, 104
 - profiling, 102
- pointers
 - optimization, 127
- position-independent code, 1, 10, 11, 58
 - assembly language, 135-139
 - branch-and-link instruction, 138
 - .cpadd reg* directive, 136
 - .cpload reg* directive, 136
 - .cprestore offset* directive, 136
 - .cpword local-sym* directive, 136
 - examples, 137
 - position-independent code
 - indirect function call, 138
 - indirect jump instruction, 138
 - .option pic0* directive, 135
 - .option pic2* directive, 135
 - register *t9*, 138
 - switch tables, 136
- pragma inline
 - optimization, 130
- preprocessing, 2
- preprocessors
 - macro, 2
- procedures, rearrange, 107
- prof*
 - Also see* profiling
 - clock* example, 99
 - heavy* example, 95
 - invocations* example, 94
 - lines* example, 96
 - procedures* example, 98
- PROFDIR environment variable, 83
- profiling, 78-106
 - Also see prof*, 79
 - basic block counting, 88-100
 - clock* option, 79, 80
 - example, 99
 - command options, 79
 - command syntax, 79
 - dis* option, 80
 - dso* option, 79, 80
 - exclude* option, 79, 80, 82, 92
 - floating point, 102
 - g* option, 92
 - heavy* option, 79, 80
 - example, 95
 - instruction distribution, 104
 - invocations* option, 79, 80
 - example, 94
 - lines* option, 79, 80
 - example, 96

profiling

- merge option, 79, 80
- multiple data files, 83
- multiprocessed executables, 106
- only option, 80, 82, 92
- overview, 78
- pcsample option, 81
- pc sampling, 81-84
 - example, 85
- pids option, 100
- pixie option, 81
- pixstats, 102
- p option, 82
- procedure invocation example, 93
- procedures option, 81
 - example, 98
- quit option, 81, 92, 97
- register usage, 104
- stall distribution, 104
- summing results, 100
- testcoverage option, 81
- zero option, 81

Q

QuickStart DSOs. *See* DSOs, QuickStart

R

- rearrange procedures, 107
- reduce paging, 107
- register
 - allocation, 132
 - usage, 104
- register t9, 138
- registry file. *See* DSOs
- relocation bits, removing, 29

remove

- relocation bits, 29
- symbol table, 29
- reorder procedures, 107
- resolve text symbols, 66
- return values, optimization, 127
- rld, 50
 - dynamic linking, 69
 - libdl, 69
 - search path, 64, 91
- runtime linker. *See* rld

S

- scalar optimizer, *copt*, 2
- scalar variables
 - word size, 131
- search path
 - rld, 64, 91
- shared libraries, static, 25, 49
- shared library, 1, 10
- shared objects, dynamic, 49
- signed data type
 - optimization, 132
- size command, 29, 40, 40-41
 - command syntax, 40
 - example, 41
- source file names, 12
- STDARG. *See* optimization
- strip command, 29, 41
 - command syntax, 41
- subscripts
 - optimization, 129
- switch statements
 - optimization, 130
- switch tables, 136
- symbol resolution, 66

symbols
 exporting, 60
 loading, 60
symbol table
 removing, 29
symbol table information, listing, 29
syntax, conventions, xv

T

tables, switch, 136
tools
 basic block counting, 88
 optimization, 115
 performance, 77
 procedure rearranger, 107
 profiling, 78
 ucode, 2
type, determining for files, 33
typographical conventions, xv

U

ucode
 object library, building, 125
 object library, using, 126
 tools, 2
ugen code generator, 2
ujoin, 2
uld, 2
umerge, 2
unassigned data type
 optimization, 131
unions
 optimization, 127
uopt optimizer, 2
upas preprocessor, 2

V

VARARG. *See* optimization
variables
 scalar, 131
virtual address space, 71

W

.word directive, 135
word-size scalar variables, 131

X

-xansi option, 19

We'd Like to Hear From You

As a user of Silicon Graphics documentation, your comments are important to us. They 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 to comment on:

- 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 include the title and part number of the document you are commenting on. The part number for this document is 007-2479-001.

Thank you!

Three Ways to Reach Us



The **postcard** opposite this page has space for your comments. Write your comments on the postage-paid card for your country, then detach and mail it. If your country is not listed, either use the international card and apply the necessary postage or use electronic mail or FAX for your reply.



If **electronic mail** is available to you, write your comments in an e-mail message and mail it to either of these addresses:

- If you are on the Internet, use this address: techpubs@sgi.com
- For UUCP mail, use this address through any backbone site:
[your_site]!sgi!techpubs



You can forward your comments (or annotated copies of manual pages) to Technical Publications at this **FAX** number:

415 965-0964