IRIS-4D Series Compiler Guide

IRIS-4D Series

SiliconGraphics
Computer Systems

Document number: 007-0905-010

IRIS-4D Series Compiler Guide

Version 1.0

Document Number 007-0905-010

Technical Publications:
Robert Reimann

Engineering:
Greg Boyd
Deborah Ryan

IRIS-4D Series Compiler Guide
Version 1.0
Document Number 007-0905-010

Silicon Graphics, Inc.
Mountain View, California

Contents

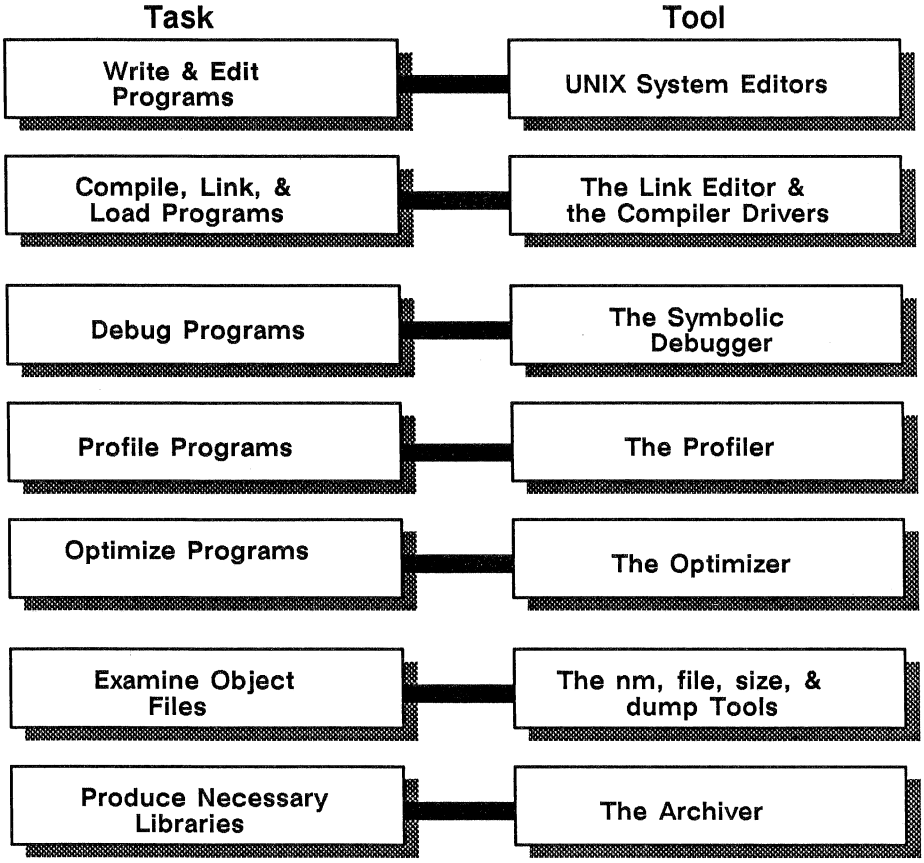
2. Improving Program Performance

1. The Compiler System

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

1.1 Overview

The components that comprise the compiler system and the task each performs are summarized below:


Task
Tool
Write & Edit Programs
UNIX System Editors
Compile, Link, & Load Programs
The Link Editor & the Compiler Drivers
Debug Programs
The Symbolic Debugger
Profile Programs
The Profiler
Optimize Programs
The Optimizer
Examine Object Files
The nm, file, size, & dump Tools
Produce Necessary Libraries
The Archiver

1.2 The Drivers

Intelligent programs called *drivers* actually invoke the following major components of the compiler system: the macro preprocessor (*cpp*), the compiler (*cc* or *f77*), the assembler (*as*), and the link editor (*ld*). This section gives an overview of driver operations and commands.

1.2.1 Driver Commands

The commands *f77*(1) and *cc*(1) run the drivers that cause your programs to be compiled, optimized, assembled, and link edited.

Each command knows the appropriate libraries associated with the main program and passes only those libraries to the link editor.

1.2.2 Files

The drivers recognize the contents of an input file by the suffix assigned to the filename, as shown below.

Suffix	Description
.e	efl source
.r	ratfor source
.s	assembly source
.i	source is assumed to be that of the that of the processing driver. For example: `f77 -c source.i` *source.i* is assumed to contain FORTRAN source statements.
.c	C source
.f	Fortran 77 source
.u	ucode object file
.b	ucode object library
.o	object file
.a	object library

NOTE: The assembly driver *as* assumes that any file, regardless of the suffix, contains assembly language statements; *as* accepts only one input source file.

1.2.3 Operational Overview

Figure 1–1 on the next page shows the relationship between the major components of the compiler system and their primary inputs and outputs.

Note that FORTRAN uses preprocessors (see Figure 1–2) that the other languages do not use. For more information, see the *efl*(1), *ratfor*(1), and *m4*(1) manual pages in the *IRIS–4D User's Reference Manual.*

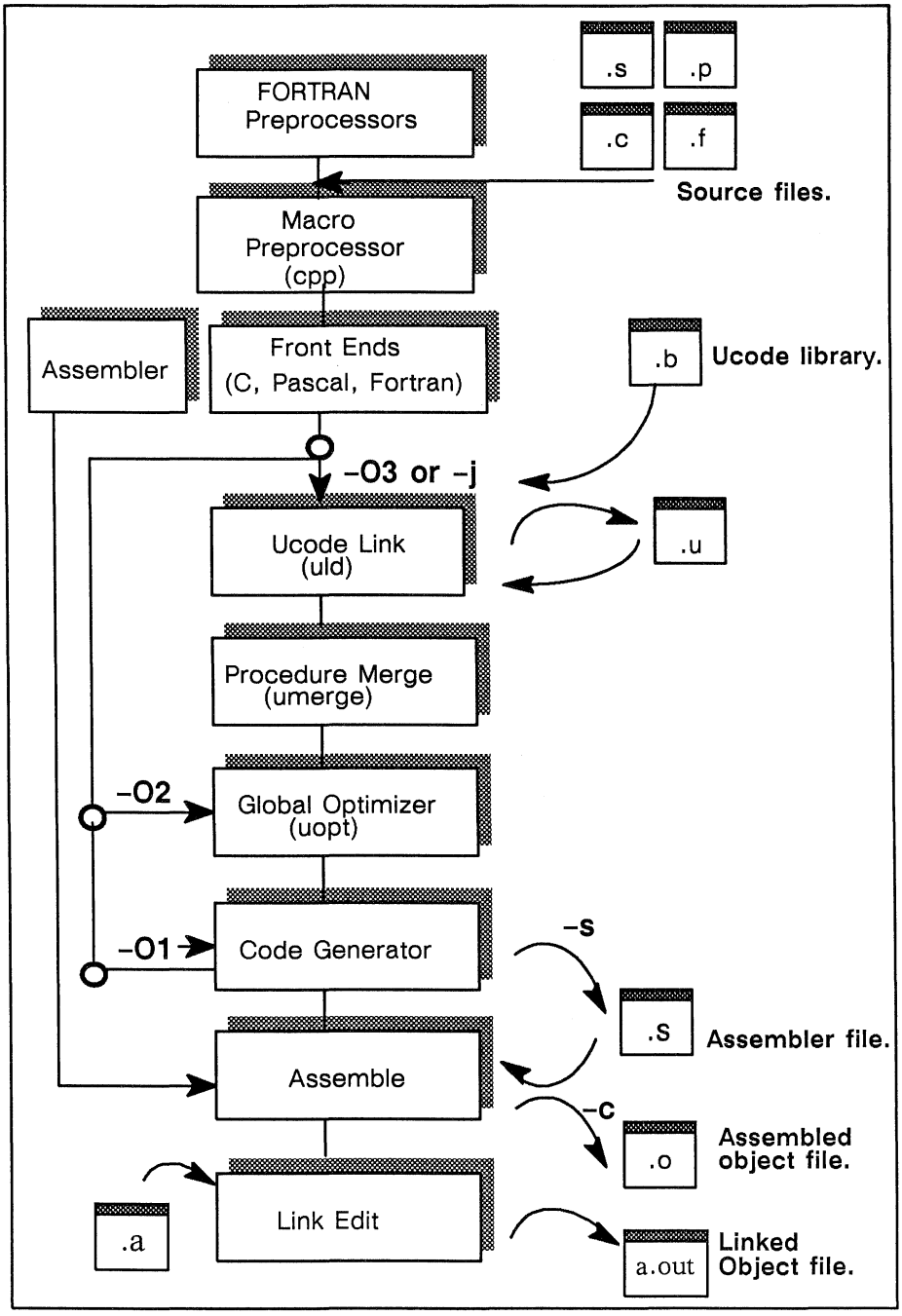


Figure 1-1. The FORTRAN Compiler System Driver

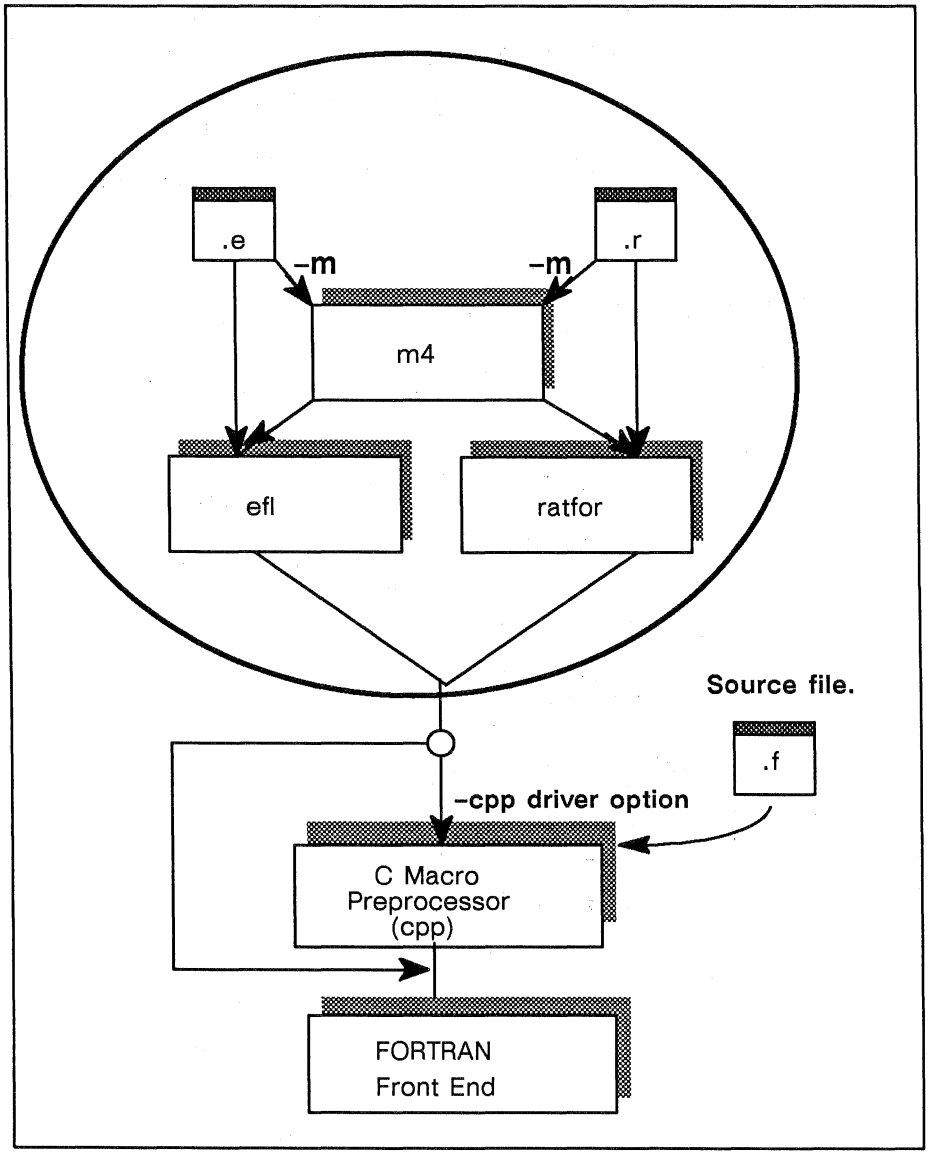


Figure 1-2. The FORTRAN Preprocessors

1.2.4 Default Options

At compilation, you can select one or more options that affect a variety of program development functions, including debugging, optimization, and profiling facilities, and the names assigned to output files.

Some options have defaults, which apply even if you don't specify them. For example, the default names for output files are *filename*.o for object files, where *filename* is the name of the source file; the default name for executable program objects is *a.out*. The following example uses the defaults in compiling source files *foo.c* and *bar.c*:

```
% cc foo.c bar.c
```

1.2.5 Compiling Multi-Language Programs

When the source language of the main program differs from that of a subprogram, you should compile each program module separately with the appropriate driver and then link them in a separate step. You can create objects suitable for link editing by specifying the **-c** option, which stops the driver immediately after the assembler phase. For example:

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

The Figure 1–3 below shows the compilation control flow for these two commands.

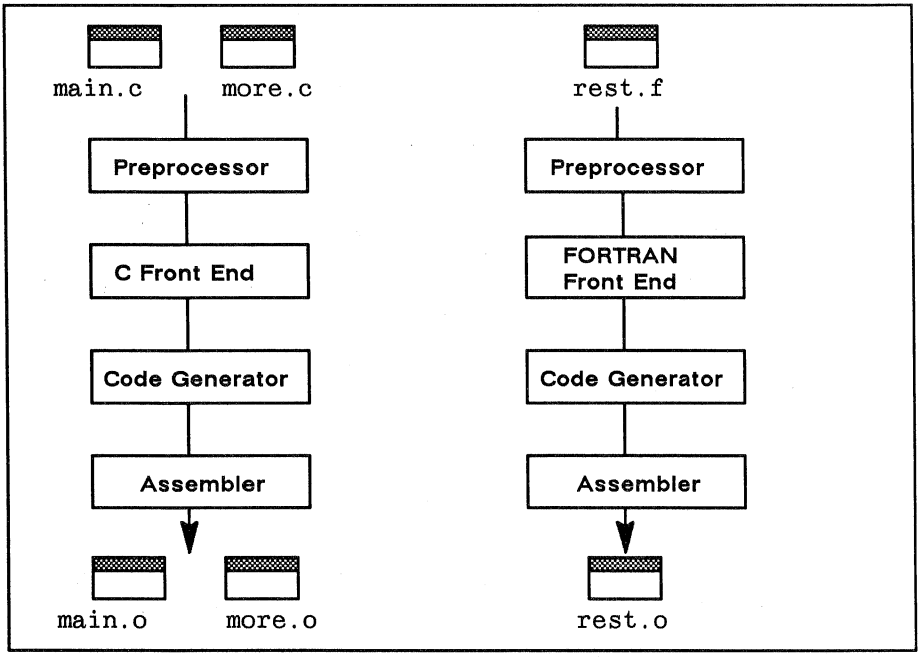


Figure 1-3. Compilation Control Flow

1.2.6 Linking Objects

You can also use a driver command to link edit separate objects into one executable program. The driver recognizes the *.o* suffix as the name of a file containing object code suitable for link editing and immediately invokes the link editor.

For more information on the link editor and on specifying link libraries, see the "Link Editor" section of this chapter. For a detailed listing of the default libraries see the *cc*(1) and *f77*(1)manual pages in the *IRIS-4D User's Reference Manual.*

1.3 Compilation Options

The tables on the following pages summarize the options you can specify for the compilation phases, which include the preprocessing phase through the assembly phase (Figure 1–1 shows these phases); the options summaries are divided into the following major groups:

- General options.
- Debugging options.
- Profiling and compiling options. See Chapter 2 for information the advantages of profiling and debugging, and how to use the profiler.
- Compiler development options.

NOTE: The tables list only the most frequently used options; they don't list all available options. See the *cc*(1) and *f77*(1) manual pages in the *IRIS–4D User's Reference Manual* for a complete list of options available.

1.3.1 General Options

The general options are listed in alphabetical order in the tables that follow.

Option Name	Purpose
-c	Prevents the link editor from linking your program after compilation. This option forces the compiler to produce an .o file even when you compiler only one program.
-C	C driver only. Used with the **-P** or **-E** options. Prevents the macro processor from stripping comments. Use this option when you suspect the preprocessor is not emitting the intended code and you wish examine the code with its comments.
-C	Pascal and FORTRAN drivers only. Generates code that causes range checking for subscripts during program execution.
-E	C driver only. Runs only the C macro preprocessor and sends results to the standard output. Specify also **-C** to retain comments. Use **-E** when you suspect the the preprocessor isn't emitting the intended code.
-D *name* **-D** *name=def*	Defines a macro*name* as if you specified a *#define* in your program. Unless you specify a definition with *name=def*, the compiler defines the name to be "1".

Option Name	Purpose
-G *num*	*num* is a decimal number that specifies the maximum size in bytes of an item to be placed in the global pointer area. The default is 512 bytes. You can raise or lower *num* to control the number of data items placed in these sections. Note: If you receive the link editor message "Too much data in the gp area . . .", you must recompile the module lowering the value currently specified for *num*.
-I *dirname*	Compiler searches the current directory, *dirname*, and the default directory, */usr/include*, in that order for the include file.
-I	When specified in addition to **-I** *dirname*, the compiler searches only *dirname* file (does not search the default directory).
-nocpp	Do not run the C macro preprocessor on FORTRAN, C and assembly source files before processing.

Option Name	Purpose
-o *filename*	Assigns the name *filename* to the program object. When used with the **-c** option, tells where to leave the .o file. The default filename is *a.out*.
-P	Same as **-E** options, except puts results in an .i file. Specify both **-P** and **-C** to retain comments.
-p1 or -p	Permits program counter (pc) sampling. This option provides operational statistics for use in improving program performance. See Chapter 2 for details. Note: This option affects only the link editor and is ignored by the compiler front-ends. When link editing as a separate step from compilation, be sure to specify this option if pc sampling is desired.
-S	Similar to **-c**, except produces assembly code in an .s file instead of object code in an .o file.
-Zv	Issues a warning message when the compiler finds a non-standard feature in the programming language of your source program.
-U *name*	Overrides a definition of a macro name that you specified with the**-D** option, or that is defined automatically by the driver.
-v	Lists compiler phases as they are executed. Use this option when you suspect a phase isn't being run as you intended. For example, the option might reveal that you failed to specify a library required by the link editor.
-V	Prints the version number of the driver and its phases. When reporting a suspected compiler program, you must include this number.
-w	Suppresses warning messages.

1.3.2 Debugging Options

The table below lists the compiler options available for debugging source code using *dbx*.

Option Name	Purpose
-g0*	Produces a program object without debugging information. Reduces the size of the program object and should be used when debugging is no longer required. Retains all optimizations.
-g1	Permits accurate, but limited, source-level debugging. This option does most optimizations.
-g or **-g2**	Permits full source-level debugging. These options often suppress optimizations that might interfere with full debugging.
-g3	Permits full, but inaccurate, debugging on fully optimized code. Debugger output may be confusing or misleading. Specify this option for programs that malfunction only after you attempt to optimize them.

*Default option.

1.3.3 Profiling Option

The compiler system permits the generation of profiled programs that, when executed, provide operational statistics. This is done through compiler option **-p** (which provides pc sampling information) and the **pixie** program (which provides profiles of basic block counts). See Chapter 2 for details.

1.3.4 Optimizer Options

The table below summarizes the options available for program optimization. However, to fully understand the benefits of optimization and how the compiler achieves optimization, you should read the "Optimization" section in Chapter 2 of this manual.

Option Name	Purpose
-O or -O2	Global optimization. Optimizes within the bounds of individual compilation units. This option executes global optimizer (uopt) phase.
-O0	No optimization. Prevents all optimizations, including the minimal optimization normally performed by the code generator and assembler.
-O1 *	The assembler and the code generator perform as many optimizations as possible without affecting compile-time performance.
-O3	All optimizations, including procedure inlining. This option must precede all source file arguments. With this option, a ucode object file is created for each source file and is left in a '.u' file. If the -c option is not present, the runtime startup routine, runtime libraries, and ucode versions of the runtime libraries are linked, as well as newly created ucode object files and ucode object files specified on the command line. If the -c option is present, only the newly created ucode files and those specified on the command line are linked. Procedure inlining is done on the resulting linkred file. This file is compiled and the object file is left in 'u.out.o' by default. If **-ko** *output* is specified, the object file is left in *output* with a suffix of '.o'.

* Default option.

1.3.5 Compiler Development Options

In addition to the standard options, each driver also has options that you normally won't use. These options primarily aid compiler development work. For information about how to use these options, consult the *cc*(1) and *f77*(1) manual pages in the *IRIS-4D User's Reference Manual.*

1.4 Including Common Files (Definition Files)

When you write programs, often you have common definition files that you share among a program's modules. Common files define things like known constants or the parameters for system calls (for example, the files that define the object file formats).

Because globally shared things should go in one place, you need a way to put these things in a common place. Definition files (often called header files in the C programming language) let you share common information between many files in a program.

Many people call these files *#include* or "header" files. These files have a "*.h*" suffix. Typically, a manual page from the *IRIS-4D User's Reference Manual* tells you to include a specific definition file.

Each supported language handles these files the same way, and you specify these files in your program's source code.

NOTE: If you intend to debug your program using *dbx*, you should not place executable code in an include file. The debugger recognizes an include file as one line of source code; none of the source lines in the file appears during the debugging session.

To specify an include file in your program, put a line like this at the beginning of your program:

```
#include "test.h"
```

You can include files in your program source files in either of two ways:

1. In column 1 of your source file, type:

   ```
   #include "filename "
   ```

   where *filename* is the name of the include file. Because you placed *filename* within double quotation marks ("), the C macro preprocessor searches in sequence the current directory and the default directory */usr/include*.

2. In column 1 of your source file, type:

   ```
   #include < filename >
   ```

filename is the name of the include file. Because you placed *filename* between the greater–than and less–than signs (< >), the C macro preprocessor skips the current directory and searches only the default directory */usr/include* for the include file.

C, FORTRAN 77, and assembly code can reside in the same include files, and then can be conditionally included in programs as required. To set up a shareable include file, you must create an *.h* file and enter the respective code as indicated in Figure 1–4:

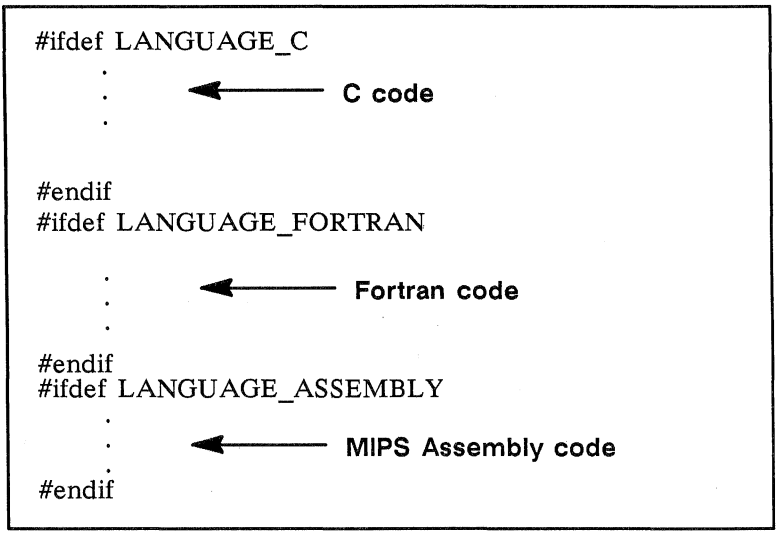


Figure 1–4.

NOTE: When you write your program, you need to include the ".*h*" file that you created.

1.5 Link Editor

This section gives summarizes the functions of the link editor and how it works. Refer to the *ld*(1) manual page in the *IRIS–4D User's Reference Manual* for complete information on the link editor options and libraries.

The link editor combines one or more object files (in the order specified) into one program object file, performing relocation, external symbol resolutions, and all the other processing required to make object files ready for execution. Unless you specify otherwise, the link editor names the

program object file *a.out*. You can execute the program object or use it as input for another link editor run.

The link editor supports all the standard command line features of other UNIX system link editors except System V *ifiles*. (An *ifile* holds a description of a load module.)

1.5.1 Running the Link Editor

You can run the link editor by typing *ld* on the command line of your shell or by using one of the driver commands as described in this chapter in the section "Linking Objects". In most cases, the driver commands should be used to call the link editor so that the object is linked with system libraries and important link editor switches. The syntax of the *ld* command is as follows:

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

NOTE: The assembler driver *as* does not run the link editor. To link edit a program written in assembly language, do either of the following:

- Assemble and link edit 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 procedures.
- Assemble the file using *as*, then link edit the resulting object file with the *ld* command.

1.5.2 Specifying Libraries

If you compile multi-language programs, be sure to explicitly load any required runtime libraries. For a list of the libraries that a language uses, see the manual pages for *cc*(1) and *f77*(1) in the *IRIS-4D User's Reference Manual*.

You may need to specify libraries when you use UNIX system packages that are not part of a particular language. Most of the manual pages for these packages list the required libraries.

NOTE: The link editor searches libraries in the order you specify.

1.5.3 Link Editor Options

Table 1–1 summarizes the link editor options. Refer also to the list of general options earlier in this chapter and to the *ld*(1) manual page in the *IRIS–4D User's Reference Manual* for complete information on options and libraries that affect link editor processing.

Option Name	Purpose
-b	Tells *ld* not to merge symbolic information entries from the same file into one entry for that file. Use this option when a file compiled for debugging has variables with the same names but different attributes. This can occur when compiling two object files that use the same include file, and variables with the same name differ because of conditional statements within the file.
-B*num*	Sets the starting address of the uninitialized data segment (bss) to the hexadecimal address *num*. This option is valid only when you've also specified the **-N** link editor option described later in this table.
-B*string*	Append*string*to the libarary name created by the **-l***x* or **kl***x* option.
-d	Forces the definition of common storage and link editor–defined symbols even if **-r** is specified.
-D*num*	Sets the starting address of the data segment (data) to the hexadecimal address *num*. This option is valid only when you've also specified the -N link editor option.
-e *epsym*	Sets the default entry point address for the output file to the specified symbol *epsym*.

Table 1–1. Link Editor Options

Option Name	Purpose
-f *fill*	Sets the fill pattern for "holes" within an output section of an object file; *fill* is a four-byte hexadecimal constant that defines the fill pattern.
-G*num*	Specifies the maximum size (in decimal bytes) of a *.comm* item that should be allocated in the small uninitialized data (sbss) section for reference by the global pointer.
-bestGnum	calculates the best-G *num* to use when compiling and linking.
-count **-nocount** **-countall**	These options control which objects are counted as recompilable for the best -G *num* calculation. By default, the **-bestGnum** option assumes that all files can be recompiled with a different -G *num* option. If you can not recompile certain object files or libraries, use these options to tell the link editor so that it will calculate the proper-G *num* value. **-nocount** states that the object file following it on the command line can not be recompiled; **-countall** overrides all **-nocount** options following it on the command line.

Table 1-1. Link Editor Options (continued)

Option Name	Purpose
-l*x*	Specifies the name of a link library, where x is the library name. The link editor searches for libx.a in /lib, /usr/lib, and /usr/local/lib directories respectively. For example, if you specify /curses, the library pathnames can be: */lib/curses.a* */usr/lib/curses.a* */usr/local/lib/curses.a* If a library relies on procedures or data from another library, specify that library's name first. If a library resides in a directory other than /lib, /usr/lib, or /usr/local/lib, use the **-L** option to specify the appropriate directory for that library.
-L *dirname*	Searches *dirname* for libraries specified in the **-l** option before searching directories */lib, /usr/lib, or /usr/local/lib.* This option must precede the **-l** option.
-L	If the link editor doesn't find the library in *dirname,* then /lib, /usr/lib, and /usr/local/lib are NOT searched. An **-L** *dirname* option must be specified with **-L.**
-m	Produces a link editor memory map in System V format.
-M	Produces a link editor memory map in BSD format.

Table 1-1. Link Editor Options (continued)

Option Name	Purpose
-n	* Creates an NMAGIC file. The text segment is read-only and shareable by all users of the file.
-N	* Creates an OMAGIC file. The text segment isn't readable and shareable by other users. The data segment follows immediately after after the text segment.
-o *filename*	Specifies a name for your object file. If you don't specify a name, the link editor uses *a.out* as the default.
-p*file*	Preserve the symbol names listed in file when loading ucode object files. The symbol names in file are separated by blanks, tabs, or new lines. See Optimizing Frequently Used Modules in Chapter 2 for an example.
-r	Performs a partial link-edit, retaining relocation entries. This is required if the object is to be re-link edited with other objects in the future. The option causes the link editor not to define common symbols and to suppress messages on unresolved references.
-s	Strips Symbol table information from the program object, reducing its size. This option might be useful when linking routines that are frequently linked into other program objects.
-T *num*	Sets the origin for the text segment to the specified hexadecimal number. The default origin is 0X 400000. The contents and format of the text segment are described in Chapter 9 of the *Assembly Language Programmer's Guide*.
-u *symname*	Makes symname undefined so that library components that define *symname* are loaded.

* See Chapter 9 of the Assembly Language Programmer's Guide for more information on NMAGIC and OMAGIC files.

Table 1-1. Link Editor Options (continued)

Option Name	Purpose
-v	Prints the name of each file as it is processed by the link editor.
-V	Prints the link editor version number. You might need this number, for example, when reporting a suspected bug in the link editor.
-VS *num*	Puts the specified decimal version stamp numin the object file that the link editor produces.
-x	Retains external and static symbols in the Symbol table to allow some debugging facilities. Doesn't retain local (non-global) symbols.

Table 1-1. Link Editor Options (continued)

1.6 Object File Tools

The following tools provide information on object files as indicated:

- **odump**: lists the contents (including the symbol table and header information) of an object file.
- **nm**: lists only symbol table information.
- **file**: provides descriptive information on the general properties of the specified file (for example, the programming language used).
- **size**: prints the size of the *text, data, rdata, sdata, bss,* and *sbss* sections. The format of these sections is described in Chapter 9 of the *Assembly Language Programmer's Guide.*

The sections that follow describe these tools in detail.

1.6.1 Dumping Selected Parts of Files (odump)

The *odump* tool lists headers, tables, and other selected parts of an object or archive file.

odump *options filename1[filname2..filenamen]*

In the above syntax description, *options* is one or more of the options and suboptions listed in Tables 1–2 and 1–3; *filename* is the name of one or more object files whose contents are to be dumped. For more information, see the *odump*(1) manual page in *IRIS–4D User's Reference Manual.*

Option Name	Purpose
-a	Dumps the archive header of each member of the specified archive library file.
-c	Dumps the string table.
-f	Dumps each file header.
-F	Dumps the file descriptor table.
-g	Dumps the global symbols in the symbol table of an archive library file.
-h	Dumps the section headers.
-i	Dumps the symbolic information header.
-l	Dumps line number information.
-o	Dumps each optional header.
-P	Dumps the procedure descriptor table.
-r	Dumps relocation information.
-R	Dumps the relative file index table.
-s	Dumps the section contents.
-t	Dumps symbol table entries.

Table 1–2. Main *odump* Options

Option Name	Purpose
-d *number*	Dumps the section number, or a range of section numbers, that starts at the specified number and that ends with the last section number or the number you specify with the **+d** auxiliary option.
+d *number*	Dumps the sections in a range that starts with the first section or with the section you specify with the **-d** option.
-n *name*	Dumps information only for the named entry name. Use this option with the **-h**, **-s**, **-r**, **-l**, and **-t** options.
-p	Suppresses the printing of headers.
-t *index*	Dumps only the indexed symbol table entry. You can specify a range of table entries by using the **+t** option with the **-t** option.
+t *index*	Dumps 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	Dumps information in symbolic rather than numeric representation (for example, in Static rather than 0X02). Use this option with all **dump** options except **-s**.
-z *name,number*	Dumps the line number entry or a range of entries that start at the specified number for the named function.
+z *number*	Dumps the line number that starts at the function name or the number specified by the **-z** option and that ends at the the number specified by the **+z** option.

Table 1-3. Auxilliary *odump* Options

1.6.2 Listing Symbol Table Information (nm)

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

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

In the above syntax description, *options* is one or more of characters (listed in Table 1–5) that specify the type of information to be printed; *filename* specifies the object file(s) or archive file(s) from which symbol table information is to be extracted. If you don't specify a file, **nm** assumes *a.out*.

The meanings of the character keys shown in an **nm** listing are described in Table 1–4.

Key	Description
N	Nil storage class, which avoids loading of unused external references.
T	External text.
t	Local text.
D	External initialized data.
d	Local initialized data.
B	External zeroed data.
b	Local zeroed data.
A	External absolute data.
a	Local absolute data.
U	External undefined data.
G	External small initialized data.
S	External small zeroed data.
s	Local small zeroed data.
R	External read-only data.
r	Local read-only data.
C	Common data.
E	Small common data.
V	External small undefined.

Table 1-4. Character Key Meanings

Option Name	Purpose
-A	Prints the listing in System V format. (Default).
-B	Prints the listing in BSD format.
-a	Prints debugging information (turns BSD output into System V format).
-b	Prints the value field in octal.
-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 filename 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 1-5. Symbol Table Dump (*nm*) Options

Option Name	Purpose
-T	Truncates characters in exceedingly long symbol names; inserts an asterisk as the last character of the truncated name. This 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 *nm*
-x	Prints the value field in hexadecimal.

Table 1-5. Symbol Table Dump (*nm*) Options (continued)

1.6.3 Determining a File's Type (file)

The **file** tool lists the properties of *program source, text, object,* and other files.

This tool often erroneously recognizes command files as C programs. For more information, see the **file(1)** manual page in the *IRIS-4D User's Reference Manual.*

Syntax:

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

Example:

```
% file test.o a.out
test.o:mipsel demand paged pure executable not stripped
a.out: mipsel demand paged pure executable not stripped
%
```

1.6.4 Determining a File's Section Sizes (size)

The **size** tool prints information about the *text, rdata, data, sdata, bss,* and *sbss* sections of the specified object or archive file(s). The contents and format of section data are described in Chapter 9 of the *Assembly Language Programmer's Guide.*

Syntax:

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

In the above syntax description, *options* is an alphabetic character (listed in Table 1-6) that specifies the format of the listing; *filename* specifies the object or archive file(s) whose properties are to be listed. If you don't specify a file, **size** assumes *a.out.*

Below is an example of a **size** statement and the listing it produces.

```
% 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  166965  360    136    40    31232      7a00
%
```

Option Name	Purpose
-A	Prints data section headers in System V format. (Default)
-B	Prints data section headers in Berkeley format.
-d	Prints the section sizes in decimal.
-o	Prints the section sizes in octal.
-V	Prints the version of size that you're using.
-x	Prints the section sizes in hexadecimal.

Table 1-6. Size Options

1.7 Archiver

An archive library is a file that contains one or more routines in object (*.o*) file format; the term *object* as used in this chapter refers to an *.o* file that is part of an archive library file. When a program calls an object not explicitly included in the program, the link editor (*ld*) looks for that object in an archive library. The editor then loads only that object (not the whole 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 about the library.
- Copying individual objects from the library into individual object file.

The sections that follow describe the syntax of the *ar* (archiver) command and some examples of how to use it. See the *ar*(1) manual page in the *IRIS-4D User's Reference Manual* for additional information.

Syntax:

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

The following explains the parameters in the above syntax description:

- *options* is one or more characters (listed in Tables 1–7 and 1–8) that specify the action that the archiver is to take. When you specify more than one option character, group the characters together with no spaces between; don't place a dash (–) character before the option characters.

- *posObject* is the name of an object within an archive library. It specifies the relative placement (either before or after *posObject*) of an object that is to be copied into the library or moved within the library. A *posObject* is required when the **m** or **r** options are specified together with the **a**, **b**, or **i** suboptions. Example 4 below shows the use of a *posObject* parameter.

- *libName* is the name of the archive library you are creating, updating, or extracting information from.

- *object* is the name object(s) or object file(s) that you are manipulating.

1.7.1 Examples

1. Create a new library and add routines to it.

```
% ar cr libtest.a mcount.o mon1.o string.o
```

   Options **c** suppresses archiver messages during the creation process. Options **r** creates the library *libtest.a* and adds *mcount.o, mon1.o, and string.o.*

2. Add or replace an object *(.o)* file to an existing library.

```
% ar r libtest.a mon1.o
```

   Option **r** replaces *mon1.o* in the library *libtest.a.* If *mon1.o* didn't already exist, the new object *mon1.o* would be added.

CAUTION: If you specify the same file twice in an argument list, it appears twice in the archive.

3. Update the library's *symdef* table.

```
% ar ts libtest.a
```

Option **s** creates the *symdef* table and **t** lists the table of contents.

NOTE: After you create or change a library, you must always use the **s** option to update the *symdef* (symbol definition) table of the archive library. The link editor uses the symdef table to locate objects during the link process.

4. Add a new file immediately before a specified file in the library.

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

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

1.7.2 Archiver Options

Table 1–7 lists the archiver options. You must specify at least one and *only* one of the following options: **d**, **m**, **p**, **q**, **r**, or **x**. In addition, you can optionally specify the **c**, **l**, **s**, **t**, and **v** options, and any of the archiver suboptions listed in the following tables.

Option Name	Purpose
c	Suppresses the warning message that the the archiver issues when it discovers that the archive you specified doesn't already exist.
d	Deletes the specified objects from the archive.
l	Puts the archiver's temporary files in the the current working directory. Ordinarily, the archiver puts those files in */tmp*. This option is useful when */tmp* is full.
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 the *posObject* parameter.
p	Prints the specified object(s) in the archive on the standard output device (usually the terminal screen).
q	Adds the specified object files to the end of the archive. An existing object file with the same name isnot deleted, and the link editor will continue to use the old file. This option is similar to the **r** option (described below), but is faster. Use it when creating a new library.

Table 1-7. Archiver Options

Option Name	Purpose
r	Adds the specified object files to the archive. This option deletes duplicate objects in the archive. If you want to add the object at a specific position in the archive library, specify an **a**, **b**, or **i** suboption together with the *posObject* parameter. See Example 4 in the preceding section for an example of using the *posObject* parameter. See also the **u** suboption. Use the **r** option when updating existing libraries.
s	Creates a *symdef* table in the archive. You must use this option each time you create or change the archive library. At least one of the following options must be specified with the **s** option: **m**, **p**, **q** **r**, or **t**.
t	Prints a table of contents on the standard output (usually the screen) for the specified object or archive file.
v	Lists descriptive information during the process of creating or modifying the archive. When specified with the **t** produces a verbose table of contents.
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. Then, the date stamp on the archive file is the last modified date. If no object are specified, copies all the library objects into the current directory.

Table 1-7. Archiver Options (continued)

The archiver has these suboptions:

Suboption Name	Use with option...	Purpose
a	**m** or **r**	[1]Specifies that the object file follow the *posObject* file you specify in the *ar* statement.
b	**m** or **r**	[1]Specifies that the object file precede the *posObject* file you specify in the *ar* statement.
i	**m** or **r**	[1] Same as **-b**
o	**x**	Used when extracting a file from the archive to the current directory. Forces the last modified date of the extracted to match that of the archive file.
u	**r**	The archiver replaces the existing object file when the last modified date is earlier (precedes) that of the new object file.

[1] See Example 4 in the preceding section for an example of using the *posObject* parameter.

Table 1-8. Archiver Suboptions

2. Improving Program Performance

This chapter describes facilities that can help reduce the execution time of your programs; it contains the following major sections:

- Profiling, which describes the advantages of the profiler and how to use it. The profiler isolates those portions of your code where execution is concentrated and provides reports that indicate where you should devote your time and effort for coding improvements.
- Optimization, which describes the compiler optimization facility and how to use it. The section also gives examples showing optimization techniques.
- Limiting the Size of Global Pointer Data, which describes the global pointer area and how, through controlling the size of variables and constants that the compiler places in this area, you can improve program performance.

2.1 Introduction

The best way to produce efficient code is to follow good programming practices:

- Choose good algorithms and leave the details to the compiler.
- Avoid tailoring your work for any particular release or quirk of the compiler system.

As technological advances cause MIPS to make changes to the current compiler system, anything you tailor now might negatively affect future program performance. Moreover, tailored code might not work at all with new versions of the system. To take action on possible compiler inefficiencies, report them directly to MIPS.

2.2 Profiling

This section describes the concept of profiling, its advantages and disadvantages, and how to use the profiler.

2.2.1 Overview

Profiling helps you find the areas of code where most of the execution time is spent. In the typical program, execution time is confined to a relatively few sections of code; it's profitable to concentrate on improving coding efficiency in only those sections. The compiler system provides the following profile information:

- pc sampling (*pc* stands for *program counter*), which highlights the execution time spent in various parts of the program.

  You obtain pc sampling information by link editing the desired source modules using the **-p** option and then executing the resulting program object, which generates profile data in raw format.

- Invocation counting, which gives the number of times each procedure in the program is invoked.

- Basic block counting, which measures the execution of basic blocks (a basic block is a sequence of instructions that is entered only at the beginning and which exits only at the end). This option provides statistics on individual lines.

  You obtain invocation counting and basic block counting information using the **pixie** program. **Pixie** takes your source program and creates an equivalent program containing additional code that counts the execution of each basic block. Executing **pixie** and the equivalent program generate the profile data in raw format.

Using the **prof** program, you can create a formatted listing of the raw profile data. The listings can indicate where to correct sub-optimal coding, substitute better algorithms, or substitute assembly language. The listings also indicate if your program has exercised all portions of the code.

Figure 2-1 gives an example of a pc sampling listing produced from a program compiled with the **-p** compiler option. The **prof** program produced the listing from the raw profile data using the **-procedure** option.

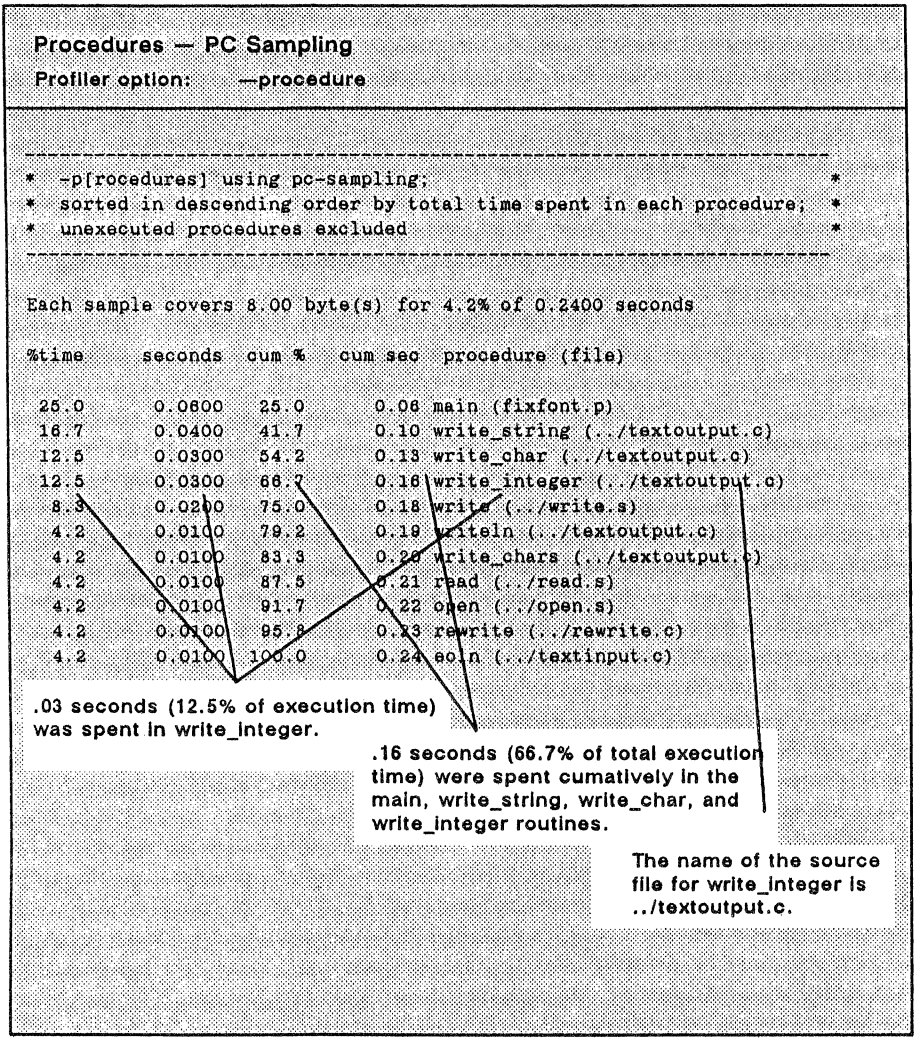


Figure 2–1. Profiler Listing for PC Sampling

Figures 2–2 through 2–6 show listings from raw data produced by **pixie**. The **prof** option used is given at the top of each figure.

Procedures — Invocation Counting

Profiler option: —pixie -invocation

```
*  -i[nvocations] using basic-block counts;                              *
*  the called procedures are sorted in descending order by number of     *
*  calls; a '?' in the columns marked '#calls' or 'line' means that data *
*  is unavailable because part of the program was compiled without       *
*  profiling.                                                            *
```

called procedure	#calls	%calls	from line, calling procedure (file):	
eoln	4017	81.51	37	main (pix.p)
	453	9.19	45	main (pix.p)
	428	8.69	19	main (pix.p)
	30	0.61	17	main (pix.p)
write_char	4014	81.75	43	main (pix.p)
	453	9.23	45	main (pix.p)
	442	9.00	42	main (pix.p)
	1	0.02	47	main (pix.p)
	0	0.00	160	write_string (../textoutput.c)
	0	0.00	225	write_integer (../textoutput.c)
	0	0.00	257	write_cardinal (../textoutput.c)
	0	0.00	284	write_real (../textoutput.c)
	0	0.00	286	write_real (../textoutput.c)
write_string	453	24.59	31	main (pix.p)
	453	24.59	29	main (pix.p)
	453	24.59	31	main (pix.p)
	453	24.59	31	main (pix.p)
	30	1.63	23	main (pix.p)
	0	0.00	189	write_enum (../textoutput.c)
write_integer	453	50.00	31	main (pix.p)
	453	50.00	31	main (pix.p)
eof	453	93.40	45	main (pix.p)
	30	6.19	23	main (pix.p)
	1	0.21	28	main (pix.p)
	1	0.21	14	main (pix.p)
writeln	453	93.60	29	main (pix.p)
	30	6.20	23	main (pix.p)
	1	0.21	47	main (pix.p)
readln	453	93.79	39	main (pix.p)
	30	6.21	21	main (pix.p)
sbrk	4	66.67	207	morecore (../malloc.c)
	1	16.67	110	malloc (../malloc.c)
	1	16.67	115	malloc (../malloc.c)
close	4	100.00	108	fclose (../flsbuf.c)
fflush	4	100.00	107	fclose (../flsbuf.c)
	0	0.00	49	_filbuf (../filbuf.c)

eoln was called 4017 times from line 37 of main. This presented 81.51% of the calls to eoln.

The source code for main is the file pix.p.

Figure 2-2. Profiler Listing for Procedure Invocations

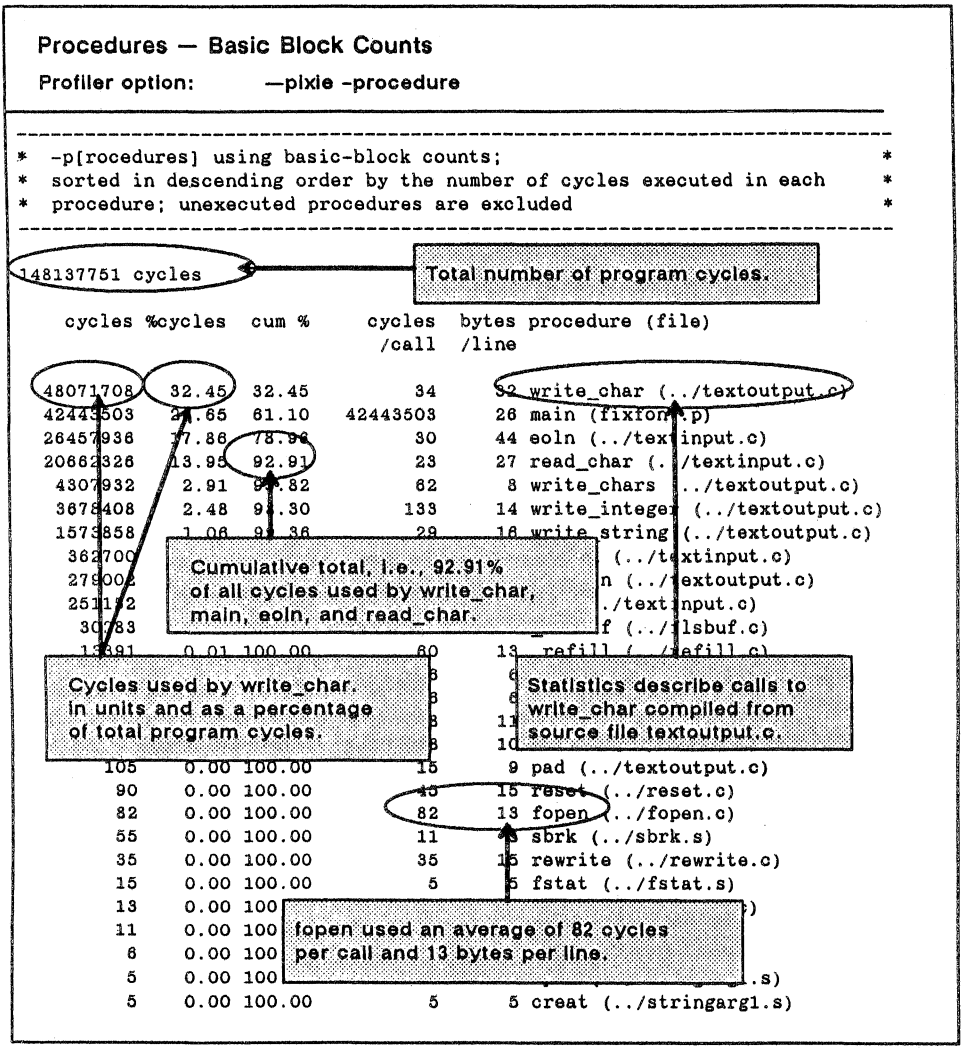


Figure 2-3. Profiler Listing for Procedures Based on Basic Blocks Counts

Procedures — Basic Block Counts (with clock time)

Profiler option: —pixie—procedure —clock

```
-------------------------------------------------------------------------------
*  -p[rocedures] using basic-block counts;                                    *
*  sorted in descending order by the number of cycles executed in each        *
*  procedure; unexecuted procedures are excluded                              *
-------------------------------------------------------------------------------

148137751 cycles (18.5172 seconds at 8.00 megahertz)

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

  48071708   32.45  32.45   6.0090                                        .c)
  42443503   28.65  61.10   5.3054
  26457936   17.86  78.96   3.3072
  20662326   13.95  92.91   2.5828                                        )
   4307932    2.91  95.82   0.5385                                        t.c)
   3678408    2.48  98.30   0.4598    133    14 write_integer (../textoutput.c)
   1573858    1.06  99.36   0.1967     29    16 write_string (../textoutput.c)
    362700    0.24  99.61   0.0453     26    67 readln (../textinput.c)
    279002    0.19  99.80   0.0349     20    30 writeln (../textoutput.c)
    251152    0.17  99.97   0.0314     19    44 eof (../textinput.c)
     30283    0.02  99.99   0.0038     63    11 _flsbuf (../flsbuf.c)
     13391    0.01 100.00   0.0017     60    13 _refill (../refill.c)
      2923    0.00 100.00   0.0004      6     6 write (../write.s)
      1356    0.00 100.00   0.0002      6     6 read (../read.s)
       735    0.00 100.00   0.0001    368    11 morecore (../malloc.c)
       116    0.00 100.00   0.0000     58    10 malloc (../malloc.c)
       105    0.00 100.00   0.0000     15     9 pad (../textoutput.c)
        90    0.00 100.00   0.0000     45    15 reset (../reset.c)
                                              n (../fopen.c)
                                                (../sbrk.s)
                                             ite (../rewrite.c)
                                              t (../fstat.s)
                                             ty (../isatty.c)
        11    0.00 100.00   0.0000     11    11 gtty (../gtty.c)
         6    0.00 100.00   0.0000      6     5 ioctl (../simple.s)
         5    0.00 100.00   0.0000      5     5 open (../stringarg1.s)
         5    0.00 100.00   0.0000      5     5 creat (../stringarg1.s)
```

The profiler computes the time in seconds based on the megahertz machine speed specified in the —clock option.

This listing contains the same information as the listing shown in Figure 4.3, except that this listing also contains the number of seconds spent in each procedure.

Figure 2–4. Profiler Listing for Procedures Based on Basic Blocks Counts (with clock times)

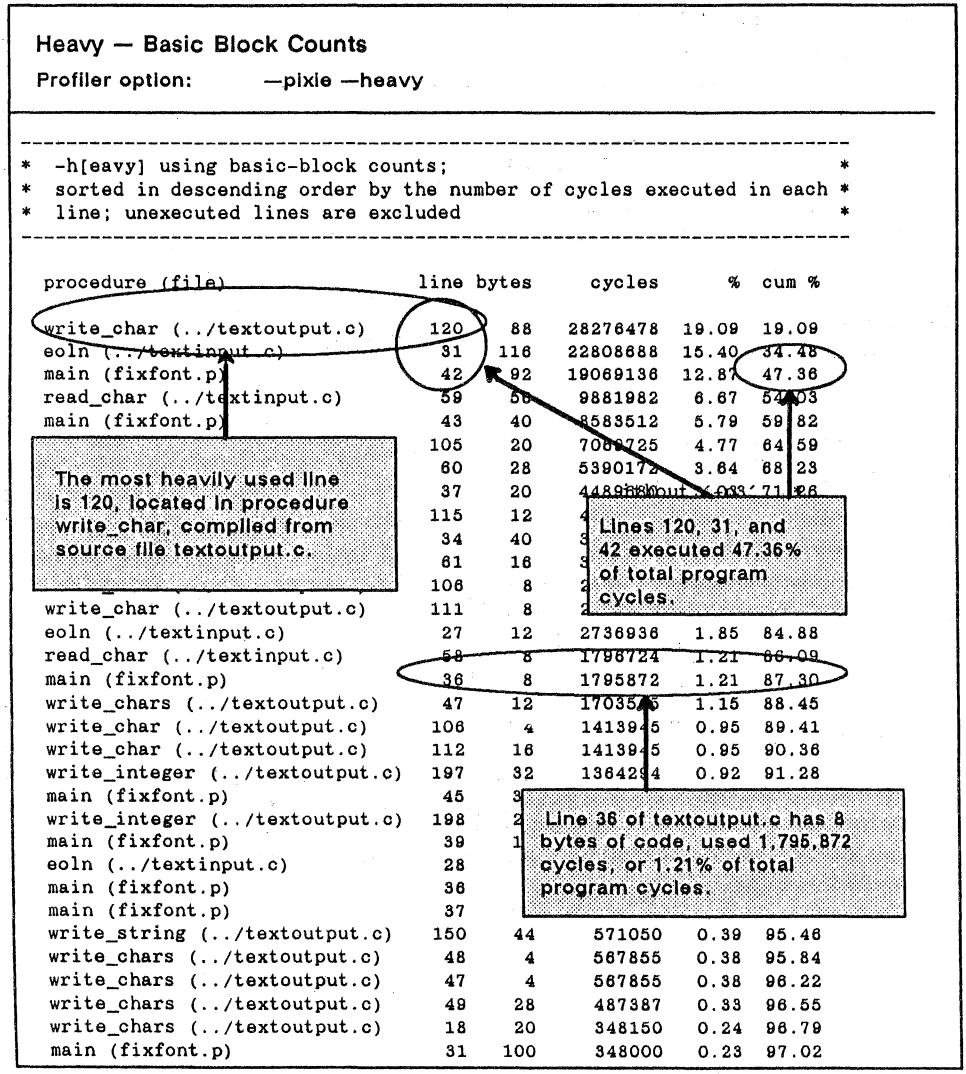


Figure 2-5. Profiler Listing for Heavy Line Usage

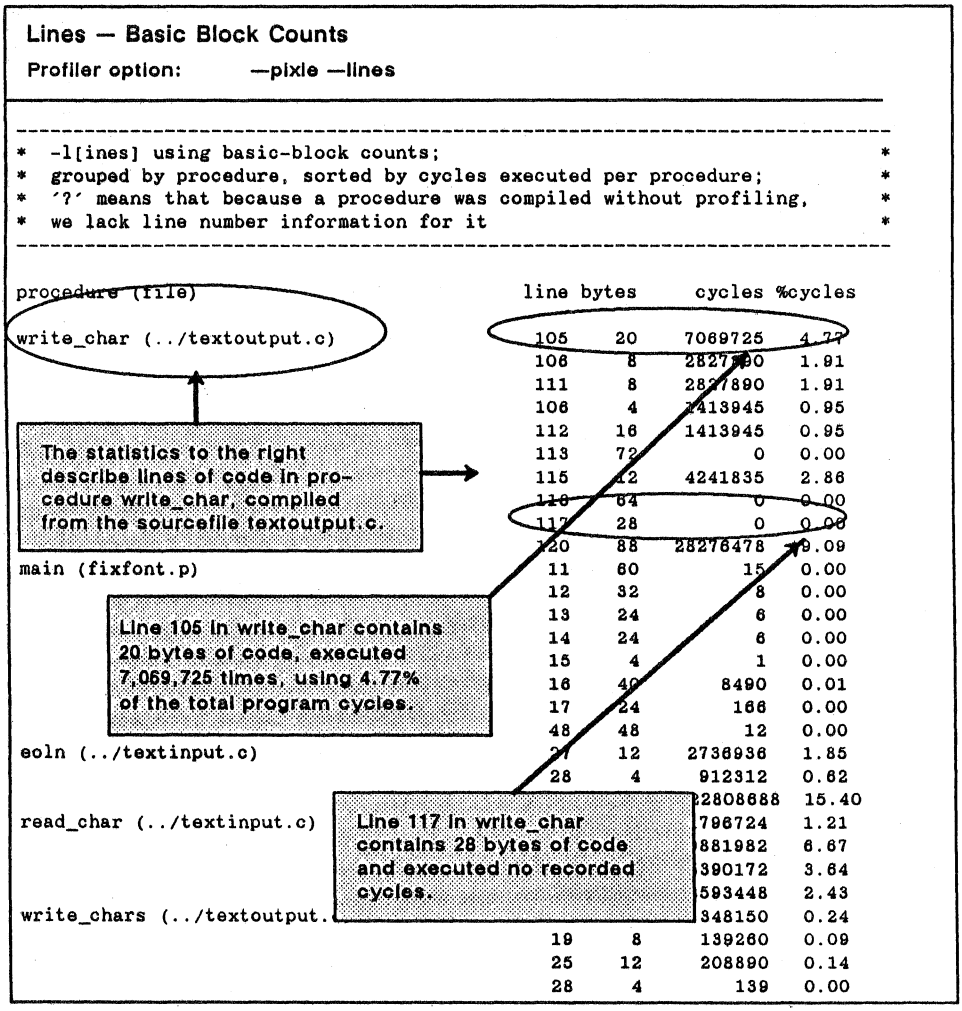


Figure 2-6. Profiler Listing for Line Information

2.2.2 How Basic Block Counting Works

Figure 2–7 on the next pages gives the steps to follow in obtaining basic block counts. Details of the steps shown in the figure are as follows:

1. Compile and link–edit. Do *not* use the *–p* option. For example:

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

2. Run the profiling program **pixie**. For example:

```
pixie -o myprog.pixie myprog
```

   Pixie takes *myprog* and writes an equivalent program containing additional code that counts the execution of each basic block. **Pixie** also generates a file (*myprog.Addrs*) that contains the address of each of the basic blocks. For more information, see the *pixie*(1) section in the *IRIS–4D User's Reference Manual.*

3. Execute *myprog.pixie*, which was generated by **pixie**. For example:

```
myprog.pixie
```

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

4. Run the profile formatting program **prof**, which extracts information from *myprog.Addr* and *myprog.Counts,* and prints it in an easily readable format. For example:

```
prof -pixie myprog myprog.Addrs myprog.Counts
```

   NOTE: Specifying *myprog.Addrs* and *myprog.Counts* is optional; **pixie** searches by default for with names in having the formal *program_name.Addrs* and *program_name.Counts.*

   You can run the program several times, altering the input data, and create multiple profile data files, if you desire. See the section **Averaging Prof Results** later in this chapter for an example.

You can include or exclude information on specific procedures within your program by using the **–only** or **–exclude prof** options (Table 2–1).

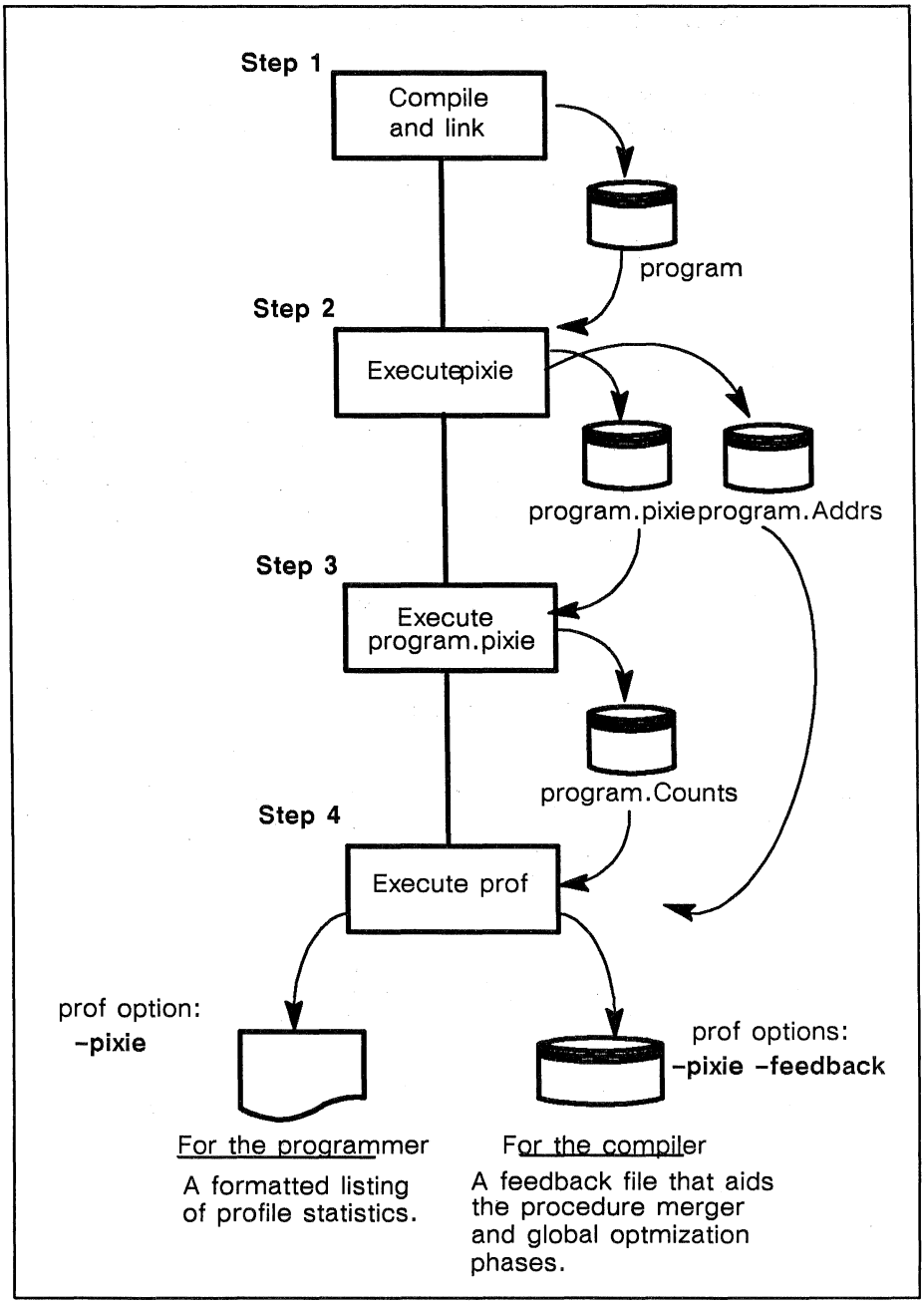


Figure 2-7. How Basic Block Counting Works

2.2.3 Averaging Prof Results

A single run of a program may not produce the typical results you require. You can repeatedly run the version of your program created by **pixie**, varying the input with each run,; then, you can then use the resulting .Counts files to produce a consolidated report. For example:

1. Compile and link-edit. Do *not* use the **-p** option. For example:

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

2. Run the profiling program **pixie**. For example:

```
pixie -o myprog.pixie myprog
```

   This step produces the *myprog.Addrs* file to be used in Step 4, as well as the modified program *myprog.pixie*.

3. Run the profiled program as many times as desired. Each time you run the program, a *myprog.Counts* file is created; rename this file before executing the next sample run. For example:

```
myprog.pixie < input1 > output1
mv myprog.Counts myprog1.Counts
myprog.pixie < input2 > output2
mv myprog.Counts myprog2.Counts
myprog.pixie < input3 > output3
mv myprog.Counts myprog3.Counts
```

4. Create the report as shown below.

```
prof -pixie myprog myprog.Addrs myprog[123].Counts
```

   prof takes an average of the basic block data in the *myprog1.Counts*, *myprog2.Counts*, and *myprog3.Counts* files to produce the profile report.

2.2.4 How PC-Sampling Works

Figure 2-8 gives the steps to follow in obtaining pc-sampling information.

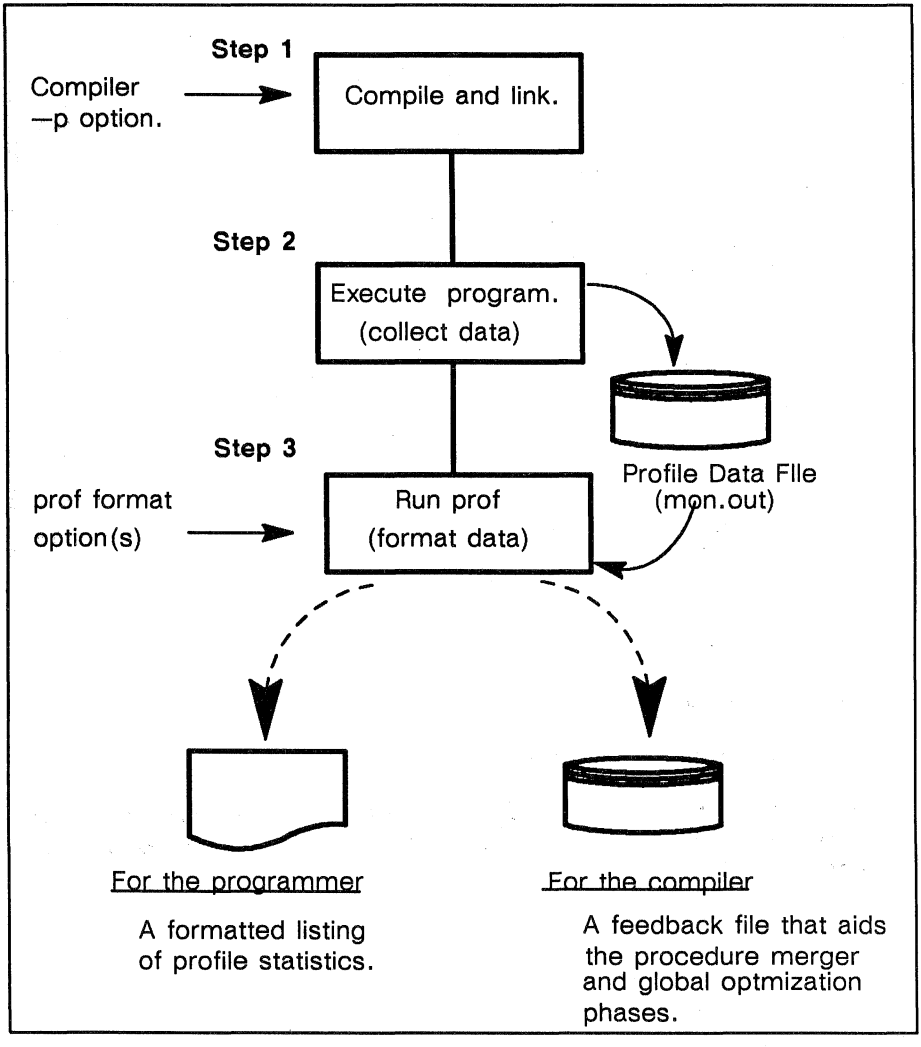


Figure 2-8. How PC-Sampling Works

Details of the steps shown in Figure 2–8 are as follows:

1. Compile and link–edit using the **–p** option. For example:

```
cc -c myprog.c
cc -p -o myprog myprog.o
```

   Note that the **-p** profiling option must be specified during the link editing step to obtain pc sampling information.

2. Execute the profiled program. During execution, profiling data is saved in the *profile data file* (the default is *mon.out*).

```
myprog
```

   You can run the program several times, altering the input data, and create multiple profile data files, if you desire. See the section **Averaging Prof Results** later in this chapter for an example.

3. Run the profile formatting program **prof**, which extracts information from the profile data file(s) and prints it in an easily readable format.

```
prof -procedure myprog mon.out
```

   For more information on **prof**, see the *prof*(1) section in the *IRIS–4D User's Reference Manual.*

You can include or exclude information on specific procedures within your program by using the **–only** or **–exclude** profiler options (Table 2–1).

2.2.5 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*. If you wish to collect profile data in several files, or specify a different name for the profile data file, set the environment variable PROFDIR as follows:

C Shell	**Bourne Shell**
`setenv PROFDIR` *string*	PROFDIR = *string* ; export PROFDIR

This causes the results to be saved in the file *string/pid.progname*, where *pid* is the process id of the executing program and *progname* is its name as it appears in *argv[0]*; *string* is the name of a directory you must create before you run the program.

2.2.6 Running the Profiler (prof)

The profiler program converts the raw profiling information into either a printed listing or an output file for use by the compiler. To run the program, type in **prof** followed by the optional parameters indicated below:

```
prof [options] [pname] { [profile_filename ... ] |
      [pname.Addrs pname.Counts] }
```

The **prof** parameters are summarized below:

options is one of the keyword or keyword abbreviations shown in Table 2–1. (You can specify either the entire name or the initial character of the option, as indicated in the table.)

pname specifies the name of your program. The default file is *a.out*.

profile_filename specifies one or more files containing the profile data gathered when the profiled program executed. If you specify more than one file, **prof** sums the statistics in the resulting profile listings.

pname.Addrs (produced by running **pixie**) and *pname*.Counts (produced by running the **pixie**–modified version of the program).

The prof program takes defaults for *profile_filename* as follows:

- If you don't specify *profile_filename,* the profiler looks for the *mon.out* file; if this file doesn't exist, it looks for the profile input data file(s) in the directory specified by the PROFDIR environment variable (see the preceding section **Creating Multiple Profile Data Files**).
- If you don't specify *profile_filename,* but do specify **–pixie**, then **prof** looks for *pname*.Addrs and *pname*.Counts and provides basic block count information if these files are present.

You might wish to consider using the **—merge** option when you have more than one profile data file; this option merges the data from several profile files into one file. See of Table 2–1 for information on the **—merge** option.

Name	Result
-p[rocedures]	Lists the time spent in each procedure. See Figure 2-3 for a sample output listing.
-pixie	Basic block counting. Indicates that information is to be generated on basic block counting, and that the program. Addrsand program. Counts files produced by **pixie** are to be used by default. See Figures 2-3 through 2-6 for sample output.
-i[nvocations]	Basic block counting. Lists the number of times each procedure is invoked. The -exclude and -only options described below apply to callees, but not to callers. See Figure 2-2 for sample output.
-l[ines]	Basic block counting. List statistics for each line of source code. See Figure 2-6 for sample output.
-o[nly] procedure_name	Reports information on only the procedure specified by *procedure_name* rather than the entire program. You may specify more than one -o option. If you specify upper-case O, prof uses only the named procedure(s), rather than the entire program, as the base upon which it calculates percentages.
-e[xclude] procedure_name	Excludes information on the procedure(s) (and their descendants) specified by *procedure_name*. If you specify upper-case E for Exclude, **prof** also omits that procedure from the base upon which it calculates percentages. The -exclude option overrides the -include option.
-z[ero]	Basic block counting. Prints a list of procedures that are never invoked.

Table 2-1. Options for the Profile List Program (prof)

Name	Result
-h[eavy]	Basic block counting. Same as the -lines option, but sorts the lines by their frequency of use. See Figure 2-5 for a sample output listing.
-c[lock] n	Basic block countingLists the number of seconds spent in each routine, based on the CPU clock frequency n, expressed in megaHertz. If you omit n, it defaults to 8.0. Never use the default if the next argument program_name or profile_filename begins with a digit. See Figure 2-4 for a sample output listing.
-t[estcoverage]	Basic block counting. Lists line numbers that contain code that is never executed.
-m[erge] filename	This option is useful when multiple input files of profile data (normally in mon.out) are used. It causes the profiler to merge the input files into filename, making it possible to specify the name of of the merged file (instead of several file names) on subsequent profiler runs.

Table 2-1. Options for the Profile List Program (prof) (continued)

Name	Result
-q[uit] n -q[uit] n% -q[uit] ncum%	Allows you to condense output listings by truncating unwanted lines. You can truncate by specifying n in three different ways: **n** — n is an integer. All lines after n lines are truncated. **n%** — n is an integer followed by the percentage sign. All lines after the line containing n% calls in the %calls column are truncated. **ncum%** — n is an integer followed by cum and a percentage sign. All lines after the line containing ncum% calls in the cum% column are truncated. For example, to eliminate the lines in the shaded portion of the sample listing below, any one of the following could be specified: -prof -q 4 -prof -q 13% -prof -q 92cum% calls %calls cum% 48071708 32.45 32.45 6.0090 42443503 28.65 61.10 5.3054 26457936 17.86 78.96 3.3072 20662326 13.95 92.91 2.5828 4307932 2.91 95.82 0.5385 3678408 2.48 98.30 0.4598 1573858 1.06 99.36 0.1967 362700 0.24 99.61 0.0453 279002 0.19 99.80 0.0349 251152 0.17 99.97 0.0314 30283 0.02 99.99 0.0038 13391 0.01 100.00 0.0017 2923 0.00 100.00 0.0004

Table 2-1. Options for the Profile List Program (prof) (continued)

2.3 Optimization

This section gives background on the compiler optimization facilities and describes their benefits, the implications of optimizing and debugging, and the major optimizing techniques.

2.3.1 Overview

Global Optimizer

The global optimizer is a single program that improves the performance of C and FORTRAN object programs by transforming existing code into more efficient coding sequences. Although the same optimizer processes C and FORTRAN optimizations, it does distinguish between C and FORTRAN programs to take advantage of the different language semantics involved.

Today, most compilers perform certain code optimizations, although the extent to which they perform these optimizations varies widely. The MIPS compilers perform more extensive optimizations compared with the average compiler available. These advanced optimizations are the results of the latest research into better and more powerful compiler techniques.

The MIPS compiler performs both machine-independent and machine-dependent optimizations. MIPS machines and other machines with RISC architectures provide a better target for machine-dependent optimizations. This is because the low-level instructions of RISC machines provide more optimization opportunities than the high-level instructions in other machines. Even optimizations that are machine-independent have been found to be effective on machines with RISC architectures. Although most of the optimizations performed by the global optimizer are machine-independent, they have been specifically tailored to the MIPS environment.

Benefits

The primary benefits of optimization, of course, 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. This frees you up to 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. Although the optimizer doesn't alter the flow of control within a program, it may move operations around so that the object code doesn't correspond to the source code. These changed sequences of code may create confusion when using the debugger.

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.

The examples in Figure 2-9 show the results of loop optimization. The source code below was compiled with and without the **-O** compiler optimization option:

```
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];
  }
```

Figure 2-9. Example of loop Optimization

Figure 2-10 shows the unoptimized and optimized code produced by the compiler. Note that the optimized version 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:

loop is 13 instructions long using 8 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      $10, 40($sp)  # address of a
        addu    $11, $10, $9  # 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:

loop is 6 instructions long using 2 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, $17 # 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 nexta[j+distance]
```

Figure 2-10. Unoptimized and Optimized Code

Register Allocation

MIPS architecture emphasizes the use of registers. Therefore, register usages have significant impact on program performance. For example, fetching a value from a register is significantly faster than fetching a value from storage. Thus, to perform its intended function, the optimizer must make the best possible use of registers.

In allocating registers, the optimizer selects those data items most suited for registers, taking into account their frequency of use and their location in the program structure. In addition, the optimizer assigns values to registers so that their contents move minimally within loops and during procedure invocations.

Optimizing Separate Compilation Units

The optimizer processes one procedure at a time. Large procedures offer more opportunities for optimization, since more inter–relationships are exposed in terms of constructs and regions. However, because of their size, large procedures require more time than smaller ones.

The *uload* and *umerge* phases of the compiler permit global optimization among separate units in the same compilation. Often, programs are divided into separate files, called modules or compilation units, which are compiled separately. This saves compile time during program development, since a change requires recompilation of only one compilation unit rather than the entire program.

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, calls to procedures that reside in other modules couldn't be fully optimized together with the code that called them.

The *uload* and *umerge* phases of the compiler system overcomes this deficiency. The *uload* phase links multi–compilation units into a single compilation unit. Then, *umerge* orders the procedures for optimal processing by the global optimizer (uopt).

2.3.2 Optimization Options

Figure 2-11 on the next page shows the major processing phases of the compiler and how the compiler —O*n* option determines the execution sequence. The table belows summarizes the functions of each of the —O options.

Option	Result
—O3	The *ulink* and *umerge* phases process the output from the compilation (C or FORTRAN) phase of the compiler, which produces symbol table information and the program text in an internal format called *ucode*. The *ulink* 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.
—O2	*Ulink* and *umerge* are bypassed, and only the global optimizer (*uopt*) phase executes. It performs optimization only within the bounds of individual compilation units.
—O1	*Ulink, umerge,* and *uopt* are bypassed. However, the code generator and the assembler perform basic optimizations in a more limited scope.
—O0	*Ulink, umerge,* and *uopt* are bypassed, and the assembler bypasses certain optimizations it normally performs.

NOTE: You should refer to the *cc*(1), *f77*(1) manual page, as applicable, in the *IRIS-4D User's Reference Manual* for details on the -O3 option and the input and output files related to this option.

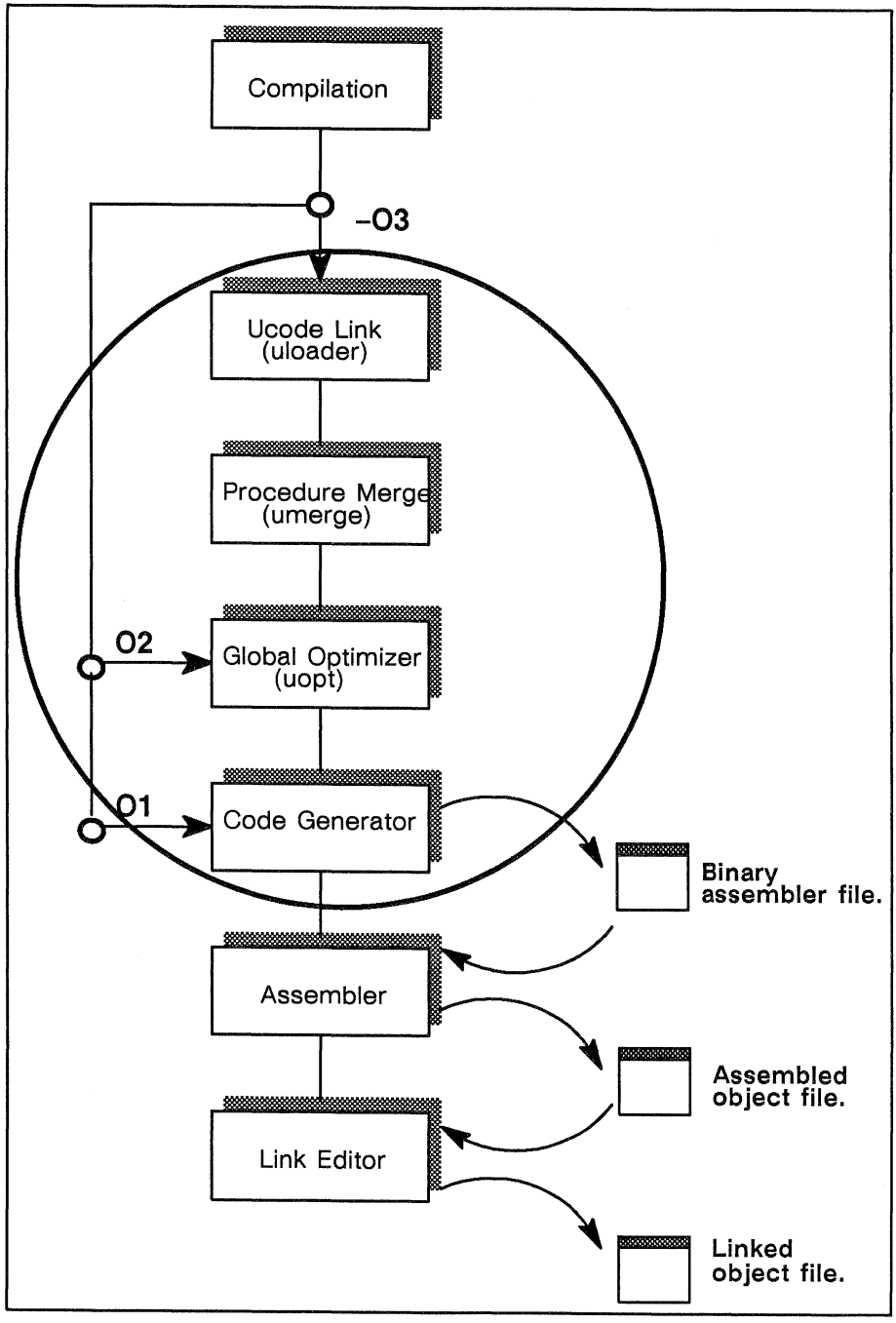


Figure 2-11. Optimization Phases of the Compiler

2.3.3 Full Optimization (-O3)

This section provides examples using the **-O3** option. The examples given assume that the program *foo* consists of three files: *a.c, b.c,* and *a.c.*

To perform procedure merging optimizations (**-O3**) on all three files, type in the following:

```
% cc -O3 -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:

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

   The **—j** option causes the compiler driver to produce a *.u* file (the standard compiler front-end output, which is made up of ucode; ucode is an internal language used by the compiler). None of the remaining compiling phases are executed, as illustrated below. The figure below illustrates the results after execution of the three commands shown above.

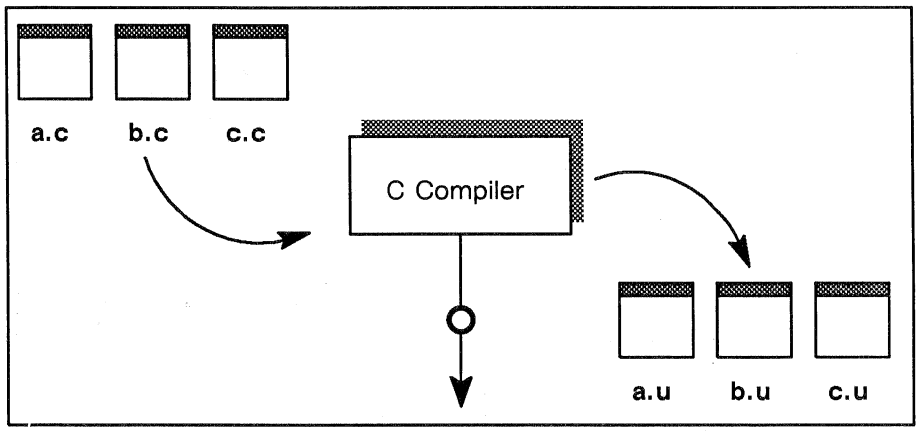


Figure 2-12.

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

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

Figure 2–13 illustrates the results of executing the above statement.

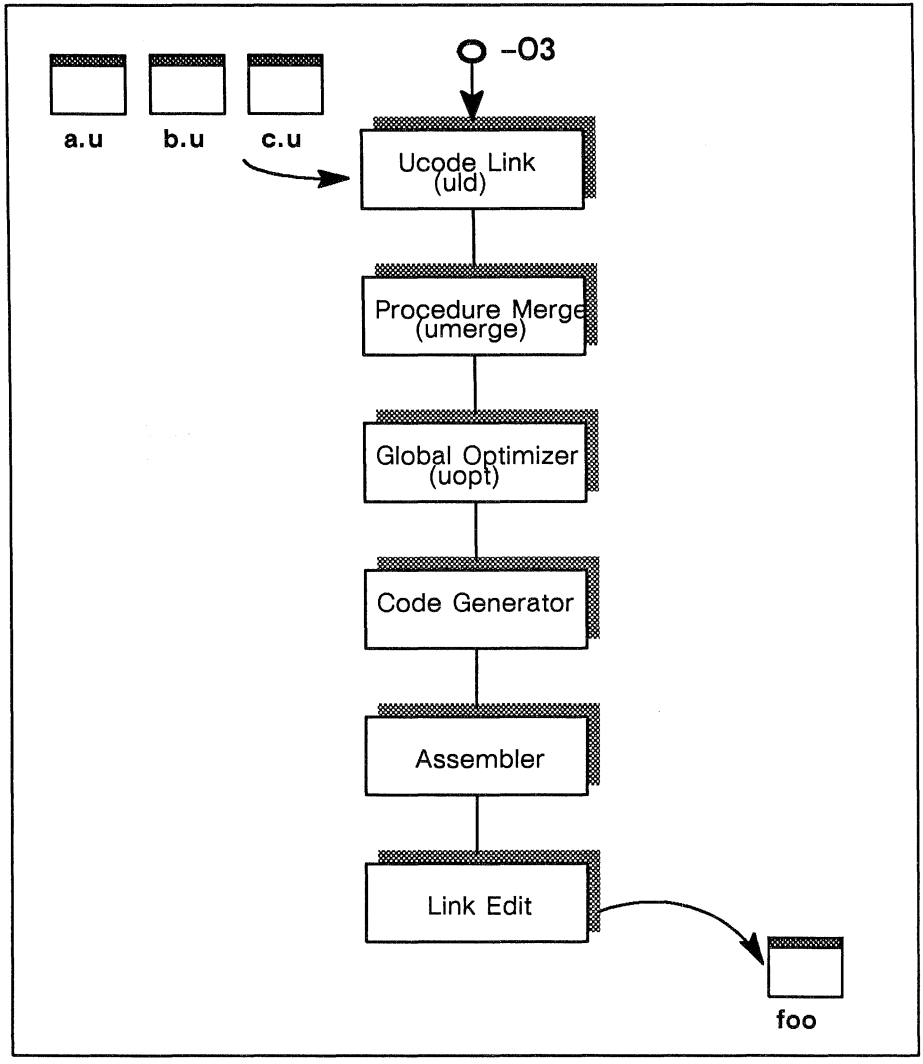


Figure 2–13.

2.3.4 Optimizing Frequently Used Modules

You may want to compile and optimize modules that are frequently called from programs written in the future. This can reduce the compile and optimization time required when the modules are needed.

In the examples that follow, *b.c* and *c.c* represent two frequently used modules that you wish to compile and optimize, 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:

```
23
% cc -j b.c
% 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. Create manually 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 following skeletal contents of *b.c* and *c.c* shown in Figure 2–14.

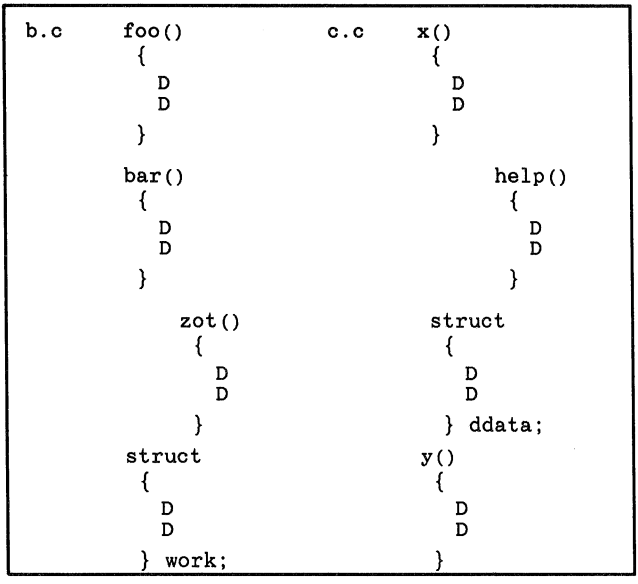


Figure 2–14.

In this example, *future.c* will call or reference 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 aren't included in *extern*.)

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

3. Now, optimize the *b.u* and *c.u* modules (Step 1) using the *extern* file (Step 2) as follows:

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

In the **—kp** option, *k* designates that the link editor option *p* is to be passed to the ucode loader.

Figure 2–15 illustrates Step 3.

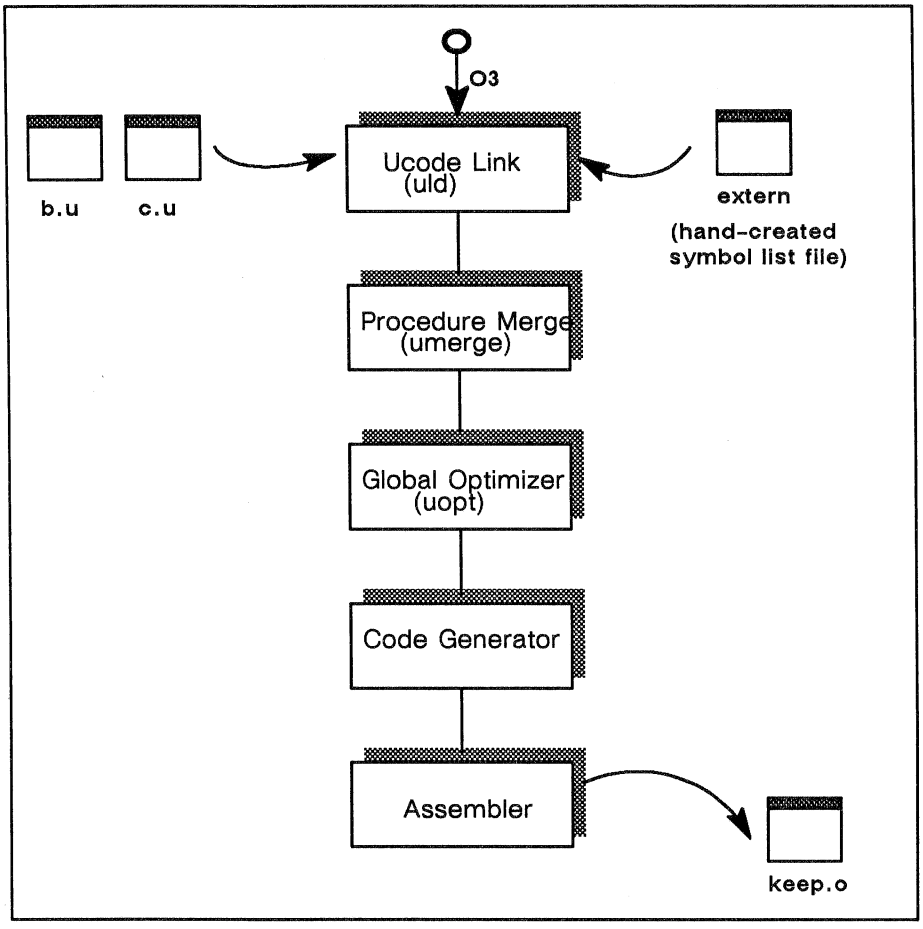


Figure 2–15.

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

```
% cc -j future.c
% cc -O3 future.u keep.o -o foo
```

The following message may appear; it means that the code in *future.c* is using a symbol from the code in *b.c* or *b.c* that was not specified in the file *extern*.

```
zot: multiply defined hidden external (should have been pre-
served)
```

Go to Step 5 if this message appears.

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

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

2.3.5 Building a Ucode Object Library

Building a ucode object library is similar to building *coff*(5) object library. 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*(5) object libraries. Using the above example, to build a *ucode* library (lib*foo*.b) of a source file, type in the following:

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

Conventional names exist for *ucode* object libraries (lib*x.b*) just as they do for *coff*(5) object libraries (libx.*a*).

2.3.6 Using Ucode Object Libraries

Using ucode object libraries is similar to using *coff(5)* object files. To load from a ucode library, specify a -kl*x* option to the compiler driver or the ucode loader. For example, to load from the ucode library the file created in the previous example, type in the following:

```
% cc -O3 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

and not all the routines as the *coff*(5) object library. In this case, you must specify to the *ucode* loader both the *ucode* object library and the *coff*(5) object library, in that order 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.

2.3.7 Improving Global Optimization

This section contains coding hints recommended to increase optimizing opportunities for the global optimizer (uopt). You should read through the recommendations in this section and, where possible, apply them to your code.

C and FORTRAN Programs

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

C Programs Only

Function return values. Use functions to return values instead of reference parameters.

Do while and repeat. Use **do while** (for C) instead of **while** or **for** when possible. For **do while**, the optimizer doesn't have to duplicate the loop condition in order to move code from within the loop to outside the loop.

Unions and variant records. Avoid unions (in C) that cause overlap between integer and floating point data types. This keeps the optimizer from assigning the 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. Use value parameters instead of reference parameters or global variables. Reference parameters have the same degrading effects as the use of pointers (see below).

Pointers and aliasing. Aliases can often be avoided by introducing local variables to store dereferenced results. (A *dereferenced* result is the value obtained from a specified address.) Dereferenced values are affected by indirect operations and calls, whereas local variables aren't. Therefore, they can be kept in registers. Figure 2–16 shows how the proper placement of pointers and the elimination of aliasing lets the compiler produce better code.

Consider the following example, which uses pointers. Because the statement *p++ = 0 might modify len, the compiler cannot place it in a register for optimal performance, but instead, must load it from memory on each pass through the loop.

Source Code:

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

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

Generated Assembly Code:

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

Figure 2–16.

Two different methods can be used to increase the efficiency of this example: using subscripts instead of pointers and using local variables to store unchanging values.

Using subscripts instead of pointers. The use of subscripting in the procedure *azero* eliminates aliasing; the compiler keeps the value of *len* in a register, saving two instructions, and still uses a pointer to access *a* efficiently, even though a pointer isn't specified in the source code. See Figure 2–17.

Source Code:

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

Generated Assembly Code:

```
          for (i = 0; i != len; i++) a[i] = 0;
        move    $2, $0          # i = 0
        beq     $4, 0, $37      # len != 0
        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:
```

Figure 2–17.

Using local variables. Specifying *len* as a local variable or formal argument (as shown in Figure 2–18) ensures that aliasing can't take place and permits the compiler to place *len* in a register.

Source Code:

```
char a[10];
void
lpzero(len)
  int len;
  {
  char *p;
  for (p = a; p != a + len; ) *p++ = 0;
  }
```

Generated Assembly Code:

```
#  8   for (p = a; p != a + len; ) *p++ = 0;
       move            $2, $6          # p = a
       addu            $5, $6, $4
       beq             $5, $6, $33     # a + len != a
$32:
       sb              $0, 0($2)       # *p = 0
       addu            $2, $2, 1       # p++
       bne             $5, $2, $32     # a + len != p
$33:
```

Figure 2-18.

In the previous example, the compiler generates slightly more efficient code for the second method.

C Programs Only

Write straightforward code. For example, don't use ++ and -- operators within an expression. When you use these operators for their values rather than for their side-effects, you often get bad code.

For example:

Bad	Good

`while (n--) {`
`  . . .`
`}`
	`while (n != 0 ) {`
	`  n--;`
	`  . . .`
	`}`

Use register declarations liberally. The compiler automatically assigns variables to registers. However, specifically declaring a **register** type lets the compiler make more aggressive assumptions when assigning register variables.

Addresses. Avoid taking and passing addresses (& values). This can create aliases, make the optimizer store variables from registers to their home storage locations, and significantly reduce optimization opportunities that would otherwise be performed by the compiler.

VARARGs. Avoid functions that take a variable number of arguments. This causes the optimizer to unnecessarily save all parameter registers on entry.

2.3.8 Improving Other Optimization

The global optimizer processes programs *only* when you explicitly 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, when followed, increase optimizing opportunities for the other passes of the compiler.

C and FORTRAN Programs

1. Use tables rather than **if–then–else** or **switch** statements.

   For example:

OK	More Efficient
`if ( i == 1) c = "1";` `else c = "0";`	`c = "01"[i];`

2. As an optimizing technique, the compiler puts the first four parameters of a parameter list into registers where they remain during execution of the called routine. Therefore, you should always declare as the first four parameters those variables that are most frequently manipulated in the called routine with floating point parameters preceding non-floating point.

3. Use word-size variables instead of smaller ones if enough space is available. This may take more space but it is more efficient.

C Programs Only

1. Rely on *libc* functions (for example, *strcpy, strlen, strcmp, bcopy, bzero, memset,* and *memcpy*). These functions were hand-coded for efficiency.

2. Use the **unsigned** data type for variables wherever possible for the following reasons: (1) because it knows the variable will always be greater than or equal to zero (>=0), the compiler can perform optimizations that would not otherwise be possible, and (2) the compiler generates fewer instructions for multiply and divide operations that use the 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* operations:

```
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 the example, *i/2* is an expensive expression; however, *j/2* is inexpensive.

2.4 Limiting the Size of Global Pointer Data

Global pointer data are constants and variables that the compiler places in the *.sdata* and *.sbss* portions of the data and *bss* segments shown in Figure 2–19. This area is referred to as the *global pointer area.*

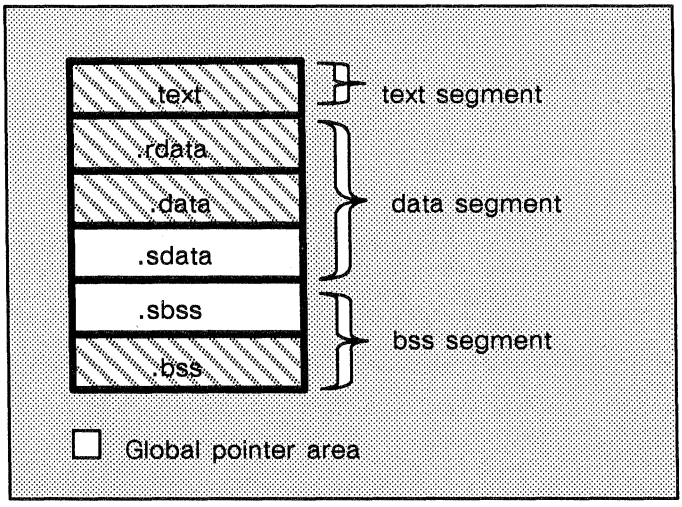


Figure 2–19.

(The *.rdata, .data,* and *.sdata* sections contain initialized data, and the *.sbss* and *.bss* sections reserve space for uninitialized data that is created by the kernel loader for the program before execution and filled with zeros. For more information on section data, see **Chapter 9** of the *Assembly Language Programmer's Guide.*)

2.4.1 Purpose of Global Pointer Data

In general, the compiler system emits two machine instructions to access a global datum. However, by using a register as a global pointer (called $gp), the compiler creates the 65536-byte global pointer area where a program can access any datum with a single machine instruction—only half the number of instructions required without a global pointer.

To maximize the number of individual variables and constants that a program can access in the global pointer area, the compiler first places those variables and constants that take the fewest bytes of memory. By default, the variables and constants occupying 512 or fewer bytes are placed in the global pointer area, and those occupying more than 512 bytes are placed in the *.data* and *.bss* sections.

2.4.2 Controlling the Size of Global Pointer Data

The more data that the compiler places in the global pointer area, the faster a program executes. However, if the data to be placed in the global pointer area exceeds 65536 bytes, the link editor prints an error message and doesn't create an executable object file. For most programs, the 512-byte default produces optimal results. However, the compiler provides the **-G** option to let you change the default size. For example, the specification

```
-G 8
```

causes the compiler to place only those variables and constants that occupy eight or fewer bytes in the global pointer area.

2.4.3 Obtaining Optimal Global Data Size

The compiler places some variables in the global pointer area regardless of the setting of the **—G** option. For example, a program written in assembly language may contain *.sdata* directives that cause variables and constants to be placed into the global pointer area regardless of size. Moreover, the **—G** option doesn't affect variables and constants in libraries and objects compiled beforehand. To alter the allocation size for the global pointer area for data from these objects, you must recompile them specifying the **—G** option and the desired value.

Thus, two potential problems exist in specifying a maximum size in the –G option:

- Using a value that is too small can reduce the speed of the program.
- Using a value that is too large can cause more than the maximum 65536 bytes to be placed in the data area, creating an error condition and producing an unexecutable object module.

The link editor **–bestGnum** option helps overcome these problems by predicting an optimal value to specify for the **–G** option. The next sections give examples of using the **–bestGnum** option and the related **–nocount** and **–count** options.

2.4.4 Examples (Excluding Libraries)

When using the **–bestGnum** option exclusive of **–nocount** and **–count**, the compiler driver assumes that you cannot recompile any libraries associated with the program; the driver causes the link editor not to consider libraries when predicting the optimal maximum size.

If you specify the option as shown below:

```
cc –bestGnum bogus.c
```

the compiler produces a message giving the best value for **–G**; if all program data fits into the global pointer area, a message indicates this. For example:

```
All data will fit into the global pointer area
Best -G num value to compile with is 80 (or greater)
```

Because all data fits into the global pointer area, no recompilation is necessary. Consider the following example, which specifies 70000 as the maximum size of a data item to be placed in the global pointer area:

```
cc bogus.c -G 70000 –bestGnum
```

The above example produces the following messages:

```
Too much data in the gp area, recompile all objects with a
smaller -G num variable than 70000
Best -G num value to compile with is 1024
```

In this example, the link editor doesn't produce an executable load module and recommends a recompilation as specified below:

```
cc bogus.c -G 1024
```

2.4.5 Example (Including Libraries)

You can explicitly specify that the link editor either include or exclude specific libraries in predicting the **—G** value. Consider the following example:

```
cc -o plotter -bestGnum plotter.o -nocount libieee.a -count
liblaser.a
```

In the above example, the link editor assumes that *libieee.a* cannot be recompiled and will continue to occupy the same space in the global pointer area. It assumes that *plotter.o* and *liblaser.a* can be recompiled and produces a recommended **—G** value to use upon recompilation.

Silicon Graphics, Inc.

Date _____

Your name _____

Title _____

Department _____

Company _____

Address _____

_____

_____

Phone _____

COMMENTS

Manual title and version _____

_____

Please list any errors, inaccuracies, or omissions you have found in this manual

_____

_____

_____

_____

Please list any suggestions you may have for improving this manual

_____

_____

_____

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 45 MOUNTAIN VIEW, CA

POSTAGE WILL BE PAID BY ADDRESSEE

Silicon Graphics, Inc.
Attention: Technical Publications
2011 Stierlin Road
Mountain View, CA 94043-1321