Chapter 3

Improving Program Performance

This chapter explains how to reduce the execution time of your programs using the compiler system. The techniques described here comprise only a minor part of performance tuning; other areas which can be tuned, but which are outside the scope of this chapter, include graphics, I/O, the kernel, memory, and *REACT* (real-time) system calls. This chapter contains the following sections:

- "Profiling with prof" describes how to use the profiler, *prof*.
- "Optimization" describes the compiler optimization facility and how to use it. The section also contains examples demonstrating optimization techniques.

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 well break in future releases of the operating 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.

Profiling with *prof*

Profiling produces detailed information about program execution. 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.

Overview of Profiling

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

The compiler system provides two kinds of profiling:

- Program counter (pc) sampling—measures the amount of execution time spent in various parts of the program.
- Basic block counting—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). Basic block counting provides statistics on individual lines.

Running the Profiler

The profiler program, *prof*, converts raw profiling information to either a printed listing or an output file for use by the compiler.

Syntax

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

options — One of the keywords or keyword abbreviations shown in Table 3-1. (Specify either the entire name or the initial character of the option, as indicated in the table.)

pname — Specifies the name of the program to be profiled. The default file is *a.out*.

profile_filename — 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, *prof* sums the statistics in the resulting profile listings.

pname.Addrs — Output file produced by running *pixie*.

pname.Counts — Output file produced by running the *pixie*-modified version of the program *pname.*

The *prof* program has these defaults:

- If you do not specify *profile_filename*, the profiler looks for the *mon.out* file; if that file does not exist, *prof* looks for profile input data files in the directory specified by the PROFDIR environment variable (refer to "Creating Multiple Profile Data Files").
- If you do not 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.

Note: *prof* and *pixie* now work on programs linked with shared libraries. For details refer to the *pixie*(1) manual page. ♦

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 Table 3-1 for information on the *–merge* option.

Table 3-1 Options for *prof*

Name	Result
–c[*lock*] *n*	A basic-block-counting option. Lists the number of seconds spent in each routine, based on the CPU clock frequency *n*, expressed in megahertz.
–e[*xclude*] *procedure_name*	Excludes information on the procedures (and their descendants) specified by *procedure_name*. If you specify uppercase *–E* instead of lowercase, *prof* also omits that procedure from the base upon which it calculates percentages. This option overrides the *–include* option.
–h[*eavy*]	A basic-block-counting option. Same as the *–lines* option, but sorts the lines by their frequency of use.
–i[*nvocations*]	A basic-block-counting option. Lists the number of times each procedure is invoked. The *–exclude* and *–only* options described below apply to callees, but not to callers.
–l[*ines*]	A basic-block-counting option. Lists statistics for each line of source code.
–m[*erge*] *filename*	Merges the input files into *filename*, allowing you to specify the name of the merged file (instead of several file names) on subsequent profiler runs. This option is useful when using multiple input files of profile data (normally in *mon.out*).

Table 3-1 (continued) Options for *prof*

Name	Result
–*o*[*nly*] *procedure_name*	Reports information on only the procedure specified by *procedure_name* rather than the entire program. You can specify more than one –*o* option. If you specify uppercase –*O*, *prof* uses only the named procedures, rather than the entire program, as the base upon which it calculates percentages.
–*pixie*	Indicates that information is to be generated on basic block counting, and that the *pname.Addrs* and *pname.Counts* files produced by *pixie* are to be used by default.
–*p*[*rocedures*]	Lists the time spent in each procedure.
–*q*[*uit*] *n*	Condense output listings by truncating unwanted lines. You can specify *n* in three different ways: *n*, an integer, truncates everything after *n* lines; *n*%, an integer followed by a percent sign, truncates everything after the line containing *n*% calls in the %calls column; and an integer followed by "cum%" truncates everything after the line containing *ncum*% calls in the cum% column.
–*t*[*estcoverage*]	A basic-block-counting option. Lists line numbers that contain code that is never executed.
–*z*[*ero*]	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
- Memory access time
- Time spent in user routines
- Time spent in system routines

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

This section explains how to obtain pc sampling and provides an example for clarification. It also explains how to create multiple profile data files.

Pc Sampling

Obtain pc sampling information by link editing 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 (refer to Figure 3-1).

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

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

2. Link-edit the object file created in Step 1.

   ```
   IRIS% 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. (During execution, profiling data is saved in the file *mon.out*.)

   ```
   IRIS% myprog
   ```

   You can run the program several times, altering the input data, to create multiple profile data files. Use the environment variable PROFDIR as explained later in this section.

4. Run the profile formatting program *prof*.

   ```
   IRIS% prof myprog mon.out
   ```

   prof extracts information from *mon.out* and prints it in an easily readable format. For more information see the *prof*(1) manual page.

Include or exclude information on specific procedures within your program by using the *–only* or *–exclude* profiler options (refer to Table 3-1).

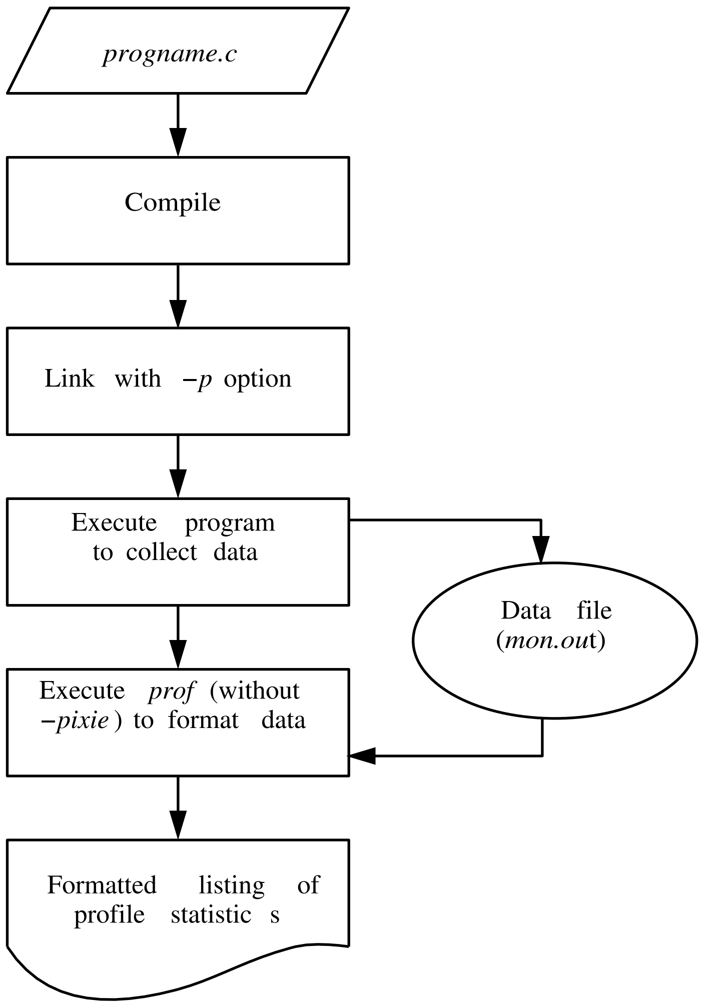

Figure 3-1 How pc Sampling Works

Example

The following listing is an example of pc sampling output from a profiled version of the ANSI C preprocessor, *acpp*:

```
Profile listing generated Sat Feb 23 15:00:10 1991 with:
  prof acpp
-------------------------------------------------------------
--
*  -p[rocedures] using
pc-sampling                            *
*  sorted in descending order by total time spent in
each     *
*  procedure; unexecuted procedures
excluded                  *
-------------------------------------------------------------
--

Each sample covers 8.00 byte(s) for 1.1% of 0.9000 seconds

%time  seconds  cum %  cum sec  procedure (file)

 28.9  0.2600   28.9   0.26     rescan (cccp.c)
 27.8  0.2500   56.7   0.51    write (sys/write.s)
 10.0  0.0900   66.7   0.60    handle_directive (cccp.c)
  8.9  0.0800   75.6   0.68    read (sys/read.s)
  5.6  0.0500   81.1   0.73    malloc (malloc.c)
  5.6  0.0500   86.7   0.78    collect_expansion (cccp.c)
  3.3  0.0300   90.0   0.81    open (sys/open.s)
  2.2  0.0200   92.2   0.83    hashf (cccp.c)
  2.2  0.0200   94.4   0.85    skip_if_group (cccp.c)
  1.1  0.0100   95.6   0.86    main (cccp.c)
  1.1  0.0100   96.7   0.87    do_define (cccp.c)
  1.1  0.0100   97.8   0.88    macroexpand (cccp.c)
  1.1  0.0100   98.9   0.89    skip_to_end_of_comment
(cccp.c)
  1.1  0.0100  100.0   0.90    strncmp (strncmp.c)
```

In the above listing:

- 0.09 seconds (10.0% of execution time) was spent in *handle_directive*.
- 0.6 seconds (66.7% of total execution time) were spent cumulatively in the *rescan*, *write*, and *handle_directive* routines.
- The name of the source file containing the *handle_directive* routine is *cccp.c* .

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 3-2:

Table 3-2 Setting the PROFDIR environment variable

C Shell	Bourne Shell
setenv PROFDIR *dirname*	PROFDIR=*dirname*; export PROFDIR

Setting this environment variable places the raw profile data of each invocation of *progname* in files named *dirname/pid.progname*. (You must create a directory called *dirname* before you run the program.) *pid* is the process ID of the executing program, and *progname* is the name of the program when invoked.

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. *pixie* takes a source program and creates an equivalent program containing additional code that counts the execution of each basic block.

Using *pixie*

Before obtaining basic block counts with *prof* and the *–pixie* option, use *pixie* to translate your source program into a profiling version and generate a file of block addresses.

Syntax

```
pixie in_prog_name [options]
```

in_prog_name Name of the input program.

options One of the keywords listed in Table 3-3. (For a complete list of options refer to the *pixie*(1) manual page.)

Table 3-3 Options for *pixie*

Name	Result
–*o out_prog_name*	Specifies a name for the equivalent program. The default is to remove any leading directory names from the *in_prog_name* and append *.pixie*.
–*bbaddrs name*	Specifies a name for the file of basic block addresses. The default is to remove any leading directory names from the *in_prog_name* and append *.Addrs*.
–[*no*]*quiet*	[Permits] or suppresses messages summarizing the binary-to-binary translation process. Default: –*noquiet*.
–[*no*]*textdata*	Controls whether *pixie* puts the original text into the translated output. This option is required to correctly translate programs with data in the text section (for example, Fortran 77 format statements in some compiler releases). Default: –*textdata* (include original text).

Obtaining Basic Block Counts

Use the procedure below to obtain basic block counts (refer to Figure 3-2).

1. Compile and link edit your program normally. Do not use the –*p* option. For example, using the input file *myprog.c*:

   ```
   IRIS% 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.

   ```
   IRIS% pixie myprog –o myprog.pixie
   ```

   pixie takes *myprog* and writes an equivalent program containing additional code that counts the execution of each basic block. *pixie* also generates a file called *myprog.Addrs* which contains the address of each basic block. For more information, refer to the *pixie*(1) manual page.

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

```
IRIS% myprog.pixie
```

This program generates a list of basic block counts in a file named *myprog.Counts*.

4. Run the profile formatting program *prof* specifying the *–pixie* option and the name of the original program.

   ```
   prof –pixie myprog myprog.Addrs myprog.Counts
   ```

   prof extracts information from *myprog.Addrs* and *myprog.Counts* and prints it in an easily readable format.

Note: Specifying *myprog.Addrs* and *myprog.Counts* is optional; *pixie* searches by default for names having the form *program_name.Addrs* and *program_name.Counts*. ♦

You can run the program several times, altering the input data, to create multiple profile data files. See “Averaging Basic Block Count Results” later in this section for an example.

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” on page 52 for details about *prof* options.

Include or exclude information on specific procedures within your program by using the *–only* or *–exclude prof* options (explained in Table 3-3). *prof* timings reflect only time spent in a specific procedure, not time spent including procedures called by that procedure. To show the time spent including called procedures specify the *–g* option to *prof*.

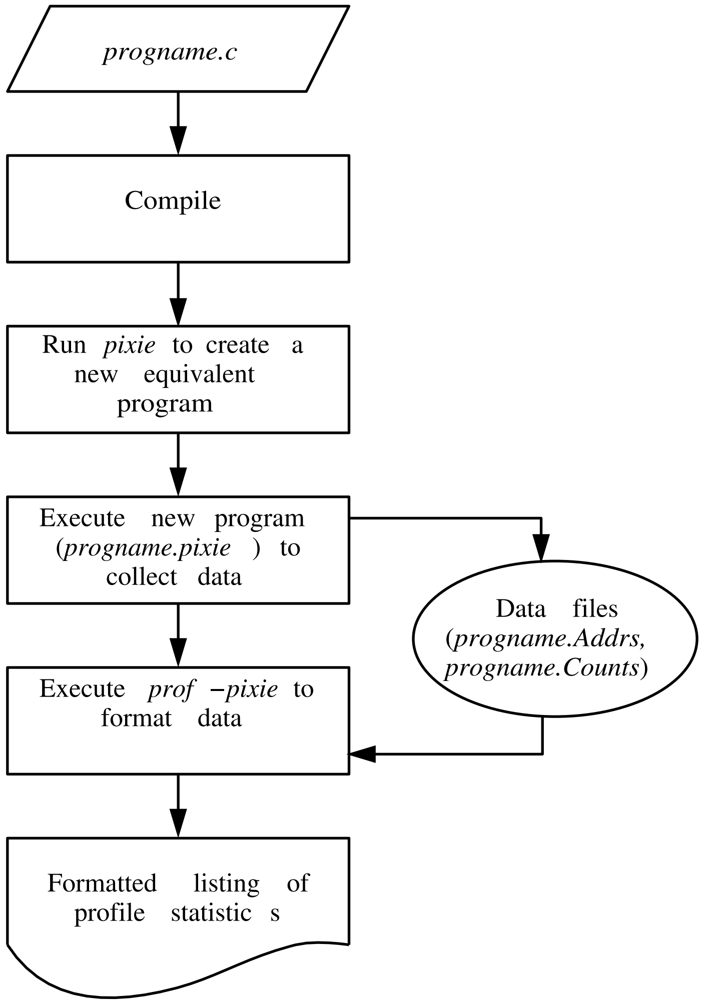

Figure 3-2 How Basic Block Counting Works

Examples

The following listings are examples of basic block counting output using *prof –pixie*, from a profiled version of the ANSI C preprocessor, *acpp*.

The first listing illustrates the use of the *–invocations* option. For each procedure invoked, *prof* produces a list of the procedures that called it:

```
IRIS% prof -pixie -invocations acpp


Profile listing generated Sat Feb 23 15:00:01 1991 with:
  prof -pixie -invocations acpp
--------------------------------------------------------------
-
*  -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.
*
--------------------------------------------------------------

called procedure #calls %calls from line, calling procedure
(file):

strncmp          1112  45.22   4022  skip_if_group (cccp.c)
                  963  39.16   2213  handle_directive
(cccp.c)
                  362  14.72   3368  collect_expansion
(cccp.c)
                   22   0.89   5246  lookup (cccp.c)
                    0   0.00   3530  do_line (cccp.c)
                    0   0.00      2  parse_number (cexp.c)
                    0   0.00   3661  do_pragma (cccp.c)
                    0   0.00   3112  compare_defs (cccp.c)
                    0   0.00   3014  do_define (cccp.c)
                    0   0.00      2  parse_number (cexp.c)
malloc           1604  73.21   5690  xmalloc (cccp.c)
                  585  26.70   5714  xcalloc (cccp.c)
                    2   0.09    244  realloc (malloc.c)
                    0   0.00    303  setcftime (_cftime.c)
                    0   0.00    268  _findbuf (flsbuf.c)
                    0   0.00    256  realloc (malloc.c)
```

In the above listing:

- *strncmp* was called 1,112 times from line 4,022 of *skip_if_group*.

- Calls from *skip_if_group* made up 45.22% of the calls to *strncmp*.
- The file *cccp.c* contains the source code for *skip_if_group*.

The following listing shows the source code lines responsible for the largest portion of execution time produced with the *–heavy* option:

```
IRIS% prof -pixie -heavy acpp

Profile listing generated Sat Feb 23 15:00:03 1991 with:
  prof -pixie -heavy acpp
--------------------------------------------------------------
--
*  -h[eavy] using basic-block
counts;                          *
*  sorted in descending order by the number of
cycles         *
*  executed in each line; unexecuted lines are
excluded       *
--------------------------------------------------------------
--

procedure (file)            line bytes     cycles      %
cum %

rescan (cccp.c)             1431    40     940652  12.80
12.80
handle_directive (cccp.c)   2291    68     428379   5.83
18.63
rescan (cccp.c)             1678    16     369193   5.02
23.65
skip_if_group (cccp.c)      4067    68     280109   3.81
27.46
rescan (cccp.c)             1679    28     277812   3.78
31.24
rescan (cccp.c)             1429     8     189746   2.58
33.82
rescan (cccp.c)             1659     8     186960   2.54
36.36
rescan (cccp.c)             2014    92     149390   2.03
38.40
rescan (cccp.c)             1720    40     143226   1.95
40.34
rescan (cccp.c)             1721    12     139773   1.90
42.25
```

```
rescan (cccp.c)            1723    12     139773   1.90
44.15
handle_directive (cccp.c)   2375    72     137623   1.87
46.02
rescan (cccp.c)            1832    16     114445   1.56
47.58
handle_directive (cccp.c)   2372    20     110966   1.51
49.09
bzero (cccp.c)             5595    20     100660   1.37
50.46
collect_expansion (cccp.c)  3309    44      97100   1.32
51.78
collect_expansion (cccp.c)  3357    72      95715   1.30
53.08
rescan (cccp.c)            1428     8      94988   1.29
54.37
```

In the above listing:

- The most heavily used line is line 1431 in procedure *rescan*, compiled from the source file *cccp.c*.
- Lines 1431, 2291, and 1678 executed 23.65% of the total program cycles.
- Line 1431 of *cccp.c* has 40 bytes of code and used 940652 cycles, which is 12.80% of the total program cycles.

The following listing, produced using the *–lines* option, shows the execution time spent on each line of code, grouped by procedure:

```
IRIS% prof -pixie -lines acpp
```

```
Profile listing generated Sat Feb 23 15:00:05 1991 with:
  prof -pixie -lines acpp
-------------------------------------------------------------
--
*  -l[ines] using basic-block
counts;                          *
*  grouped by procedure, sorted by cycles executed
per        *
*  procedure;  '?' means that line number information is
not  *
*
available.                                                 *
-------------------------------------------------------------
--
```

```
procedure (file)                  line bytes     cycles
%cycles

rescan (cccp.c)                   1355    52       1495
0.02
                                  1360     4        115
0.00
                                  1363     4        115
0.00
                                  1383     4        115
0.00
                                  1415    52        230
0.00
                                  1416     8          0
0.00
                                  1418    32        920
0.01
                                  1419    64       1612
0.02
                                  1420     4        115
0.00
                                  1424    16        460
0.00
                                  1425    24        460
0.00
                                  1428     8      94988
1.29
                                  1429     8     189746
2.58
                                  1431    40     940652
12.80
                                  1433    12          0
0.00
                                  1434    12          0
0.00
                                  1435    16          0
0.00
                                  1438     4          0
0.00
                                  1439    12          0
0.00
                                  1440     4          0
0.00
                                  1441     8          0
0.00
```

```
                                  1446    16          0
0.00
                                  1447    12          0
0.00
```

In the above listing:

- The statistics describe the lines of code in procedure *rescan* compiled from the source file *cccp.c*.
- Line 1355 in *rescan* contains 52 bytes of code, executed 1,495 times, using 0.02% of the total program cycles.
- Line 1360 in *rescan* contains 4 bytes of code and did not execute any recorded cycles.

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:

```
calls    %call    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   100.0  0.0017
2923       0.00   100.00 0.0017
```

Any one of the following commands will eliminate everything from the line starting with 4307932 to the end of the listing:

```
prof –quit 4
prof –quit 13%
prof –quit 92cum%
```

The following listing, produced with the *–procedures* option, shows the percentage of execution time spent in each procedure:

```
IRIS% prof -pixie -procedures acpp

Profile listing generated Sat Feb 23 15:00:01 1991 with:
  prof -pixie -procedures acpp
-------------------------------------------------------------
--
*  -p[rocedures] using basic-block
counts;                    *
*  sorted in descending order by the number of
cycles         *
*  executed in each procedure; unexecuted procedures
are      *
*
excluded                                                   *
-------------------------------------------------------------
--

7350305 cycles

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

3536769 48.12  48.12 30755   17   rescan (cccp.c)
1231455 16.75  64.87  1671   18   handle_directive (cccp.c)
684976   9.32  74.19  1616   19   collect_expansion (cccp.c)
453805   6.17  80.36  4777   17   skip_if_group (cccp.c)
258087   3.51  83.88   118   14   malloc (malloc.c)
234150   3.19  87.06   445   17   skip_to_end_of_comment
(cccp.c)
111560   1.52  88.58    46   21   strncmp (strncmp.c)
109313   1.49  90.07   258   18   do_define (cccp.c)
101035   1.37  91.44  1348   14   bzero (cccp.c)
 83969   1.14  92.58   124   13   skip_quoted_string (cccp.c)
 71584   0.97  93.56   309   21   macroexpand (cccp.c)
 52005   0.71  94.26    91   27   hashf (cccp.c)
 50685   0.69  94.95   114   17   install (cccp.c)
 49438   0.67  95.63   634   18   _doprnt (doprnt.c)
 43520   0.59  96.22    75   13   xcalloc (cccp.c)
 30143   0.41  96.63    63   16   bcopy (cccp.c)
```

In the above listing:

- The total number of program cycles is 7,350,305.

- *rescan* used 3,536,769 cycles, which is 48.12% of the total number of program cycles.
- The cumulative total of all cycles used by *rescan*, *handle_directive*, and *collect_expansion* is 74.19% (see column 3 of the third row).
- *rescan* used an average of 30,755 cycles per call and consisted of 17 bytes of generated code per line of source text.
- The procedure *rescan* is in the source file *cccp.c* .

You can add absolute time information to the output by specifying the clock rate, in megahertz, with the *–clock* option. The following listing shows the output:

```
IRIS% prof -pixie -procedures -clock 20 acpp

Profile listing generated Sat Feb 23 15:00:01 1991 with:
  prof -pixie -procedures -clock acpp
-------------------------------------------------------------
--
*  -p[rocedures] using basic-block
counts;                    *
*  sorted in descending order by the number of
cycles        *
*  executed in each procedure; unexecuted procedures
are      *
*
excluded                                                  *
-------------------------------------------------------------
--

7350305 cycles (0.3675 seconds at 20.00 megahertz)

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

3536769 48.12  48.12 0.176 30755 17  rescan (cccp.c)
1231455 16.75  64.87 0.0616 1671 18  handle_directive
(cccp.c)
684976  9.32  74.19  0.0342 1616  19  collect_expansion
(cccp.c)
453805  6.17  80.36  0.022  4777  17  skip_if_group (cccp.c)
258087  3.51  83.88  0.0129  118  14  malloc (malloc.c)
111560  1.52  88.58  0.0056   46  21  strncmp (strncmp.c)
109313  1.49  90.07  0.0055  258  18  do_define (cccp.c)
101035  1.37  91.44  0.00   1348  14  bzero (cccp.c)
```

```
71584  0.97  93.56  0.0036   30  21  macroexpand (cccp.c)
52005  0.71  94.26  0.0026   91  27  hashf (cccp.c)
50685  0.69  94.95  0.0025  114  17  install (cccp.c)
49438  0.67  95.63  0.0025  634  18  _doprnt (doprnt.c)
43520  0.59  96.22  0.0022   75  13  xcalloc (cccp.c)
30143  0.41  96.63  0.0015   63  16  bcopy (cccp.c)
26577  0.36  96.99  0.0013   55  15  alloca (alloca.c)
```

The above listing contains the same information as the previous listing except that this listing also contains the number of seconds spent in each procedure. The profiler computes the time, in seconds, based on the machine speed specified with the *–clock* option (in megahertz.) In this example the speed specified is 20 megahertz.

Averaging 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-edit the input file. Do not use the *–p* option. For example, using the input file *myprog.c*:

   ```
   IRIS% cc –c myprog.c
   ```

   ```
   IRIS% cc –o myprog myprog.o
   ```

   The *cc* compiler compiles *myprog.c* and saves the object file as *myprog.o*. The second command link-edits *myprog.o* and saves the executable as *myprog*.

2. Run the profiling program *pixie*.

   ```
   IRIS% pixie myprog –o myprog.pixie
   ```

   pixie generates the file *myprog.Addrs* that contains the address of each basic block. It also 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* file. Rename this file before executing the next sample run.

   ```
   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.

   ```
   IRIS% 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.

Using *pixstats*

Use the *pixstats* command to analyze a program's execution characteristics. *prof*-generated reports ignore time spent in floating point operations and time spent accessing memory. *pixstats* provides accurate floating point information by analyzing the *.Addrs* and the *.Counts* files created through *pixie*. 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

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

Syntax

```
pixstats program [options]
```

program Specifies the name of the program to be analyzed.

options One of the keywords shown in Table 3-4

Table 3-4 Options for *pixstats*

Name	Result
–cycle ns	Assumes an *ns* cycle time when converting cycle counts to seconds.
–r2010	Uses r2010 floating point chip operation times and overlap rules. This option is the default.
–r2360	Uses r2360 floating point board operation times and overlap rules.
–disassemble	Disassembles and lists the analyzed object code.

Use the following procedure to run *pixstats*:

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

   IRIS% **cc –c myprog.c**

   IRIS% **cc –o myprog myprog.o**

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

2. Run the profiling program *pixie*.

   IRIS% **pixie –o myprog.pixie myprog**

   pixie generates the file *myprog.Addrs* that contains the address of each basic block. It also generates the modified program *myprog.pixie*.

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

   IRIS% **myprog.pixie**

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

4. Run *pixstats* to generate a detailed report.

```
IRIS% pixstats myprog
```

Example

The following example shows part of a listing generated by running *pixstats* on the file *acpp*:

```
pixstats acpp:
   7670434 (1.043) cycles (0.307s @ 25.0MHz)
   7350710 (1.000) instructions
   2351262 (0.320) basic blocks
     15581 (0.002) calls
   1069316 (0.145) loads
    499584 (0.068) stores
   1568900 (0.213) loads+stores
   1568932 (0.213) data bus use
    686354 (0.093) partial word references
   1941521 (0.264) branches
   1041239 (0.142) nops
         0 (0.000) load interlock cycles
    319724 (0.043) multiply/divide interlock cycles (12/35 cycles)
         0 (0.000) flops (0 mflop/s @ 25.0MHz)
         0 (0.000) floating point data interlock cycles
         0 (0.000) floating point add unit interlock cycles
         0 (0.000) floating point multiply unit interlock cycles
         0 (0.000) floating point divide unit interlock cycles
         0 (0.000) other floating point interlock cycles
         0 (0.000) 1 cycle interlocks (2 cycle stalls -- not counted
                   in total)
         0 (0.000) overlapped floating point cycles
     18400 (0.003) interlock cycles due to basic block boundary
0.272 load nops per load
0.318 stores per memory reference
0.437 partial word references per reference
  3.1 instructions per basic block
  3.8 instructions per branch
0.630 backward branches per branch
0.300 branch nops per branch
492 cycles per call
472 instructions per call
.
.
.
 cycles %cycles  cum%   instrs  c/i   calls  c/call name
```

```
3828673  49.9%  49.9%  3536769  1.1   115    33293 rescan
1231455  16.1%  66.0%  1231455  1.0   737     1671 handle_directive
 684976   8.9%  74.9%   684976  1.0   424     1616 collect_expansion
.
.
.
```

In the above listing the first line shows a total of 7,670,434 cycles used by *acpp*. This total includes floating point calculations. The second line shows a total of 7,350,710 instructions. This total does not include floating point calculations. By comparing the two totals, you can analyze floating point versus integer calculations.

Profiling Multiprocessed Executables

You can gather both *pixie* and pc sampling profile data from executables that use the *sproc* system call, such as POWER Fortran and POWER C executables. Prepare and run the job using the same process 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.program_name*. When using *pixie*, the data files are named *program_name.Countsprocess_id*. This naming convention avoids the potential conflict of all the threads attempting to write simultaneously to the same file.

Optimization

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

Overview

This section contains an overview of optimization. It explains the global optimizer, the benefits of optimization, and other general topics.

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 better target for machine-dependent optimizations. 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 optimizations performed by the global optimizer are machine-independent, they have been specifically tailored to the Silicon Graphics environment.

Benefits

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.

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 6 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 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 because the multiplication and addition operations are overlapped.

The optimizer performs a similar sort of transformation, although the optimizer does this 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 result is more complicated than the original code. ♦

Register Allocation

The Silicon Graphics architecture emphasizes the use of registers. Therefore, register usage has a significant impact on program performance. For example, fetching a value from a register is significantly faster than fetching a value from RAM. The optimizer must therefore make the best possible use of registers.

The optimizer allocates registers for the most suitable data items, taking into account their frequency of use and their locations in the program structure. In addition, the optimizer assigns values to registers in such a way as to minimize shifting around 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 interrelationships are exposed in terms of constructs and regions. However, large procedures require more time to optimize than smaller ones.

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*.

Optimization Options

Invoke the optimizer by specifying a compiler driver, such as *cc*(1), with any of the options listed in Table 3-5. Figure 3-3 shows the major processing phases of the compiler and how the compiler *–On* option determines the execution sequence.

Table 3-5 Optimization Options

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

Table 3-5 (continued) Optimization Options

Option	Result
–O2	Specifies global optimization. Optimizes within the bounds of individual compilation units. This option executes the global optimizer (*uopt*) phase. *uld* and *umerge* are bypassed, and only the *uopt* phase executes. It performs optimization only within the bounds of individual compilation units.
–O3	This option cannot be used to compile with DSOs; it is only available for non-shared programs. The *–non_shared* option must therefore be used whenever *–O3* is used. *–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 *.u* file, for each source file. The runtime start-up routine, runtime libraries, and ucode versions of the runtime 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 *cc*(1), *pc*(1), or *f77*(1) manual pages, as applicable, for details on the *–O3* option and the input and output files related to this option. ♦

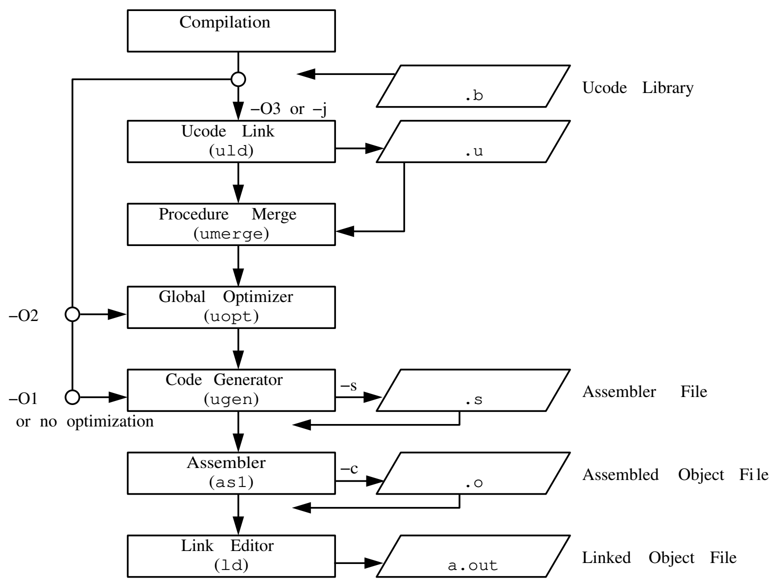

Figure 3-3 Optimization Phases of the Compiler

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 3-4.

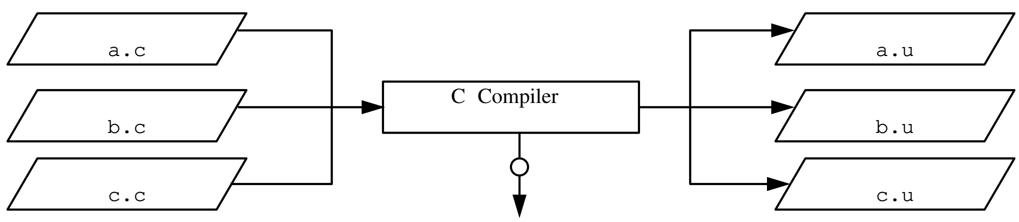

Figure 3-4 Compiling with the *–j* Option

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

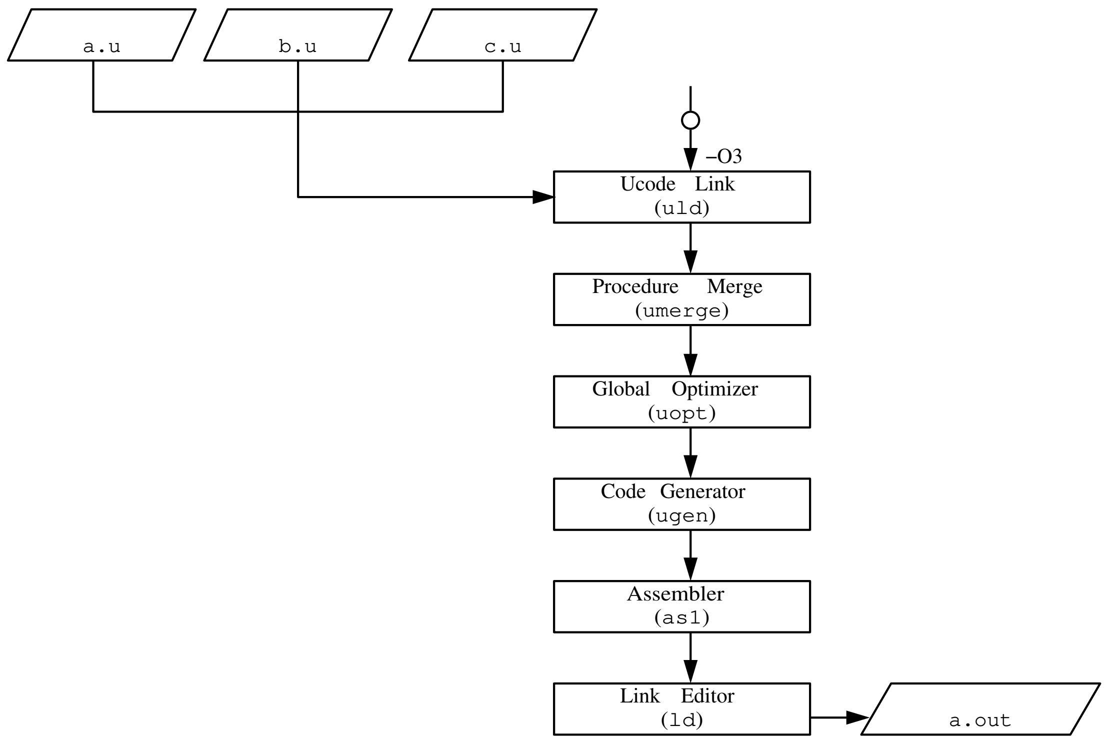

Figure 3-5 Executing Full Optimization

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, manually 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 b.c                 File c.c

foo()                    x()
 {                        {
  .                        .
  .                        .
 }                        }

bar()                    help()
 {                        {
  .                        .
  .                        .
  zot()                   }
   {
     .                   struct
     .                    {
   }                       .
                           .
  struct                  } ddata;
   {
     .                   y()
     .                    {
   } work;                  .
 }                          .
                          }
```

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 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 -kp extern b.u c.u -o keep.o
   ```

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

   Figure 3-6 illustrates Step 3.

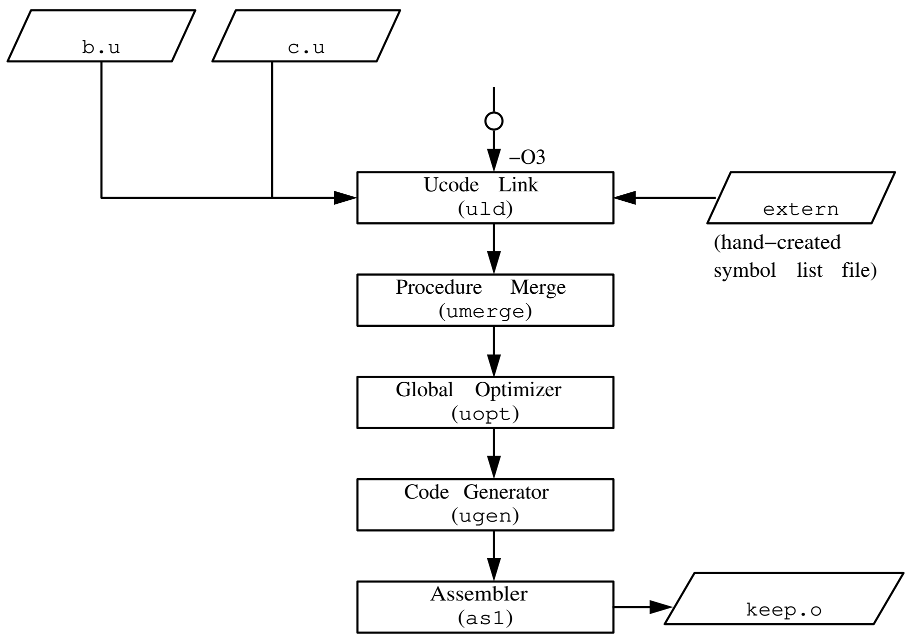

Figure 3-6 Optimization Process

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
   ```

Building a Ucode Object Library

Building a ucode object library is similar to building a *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* 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 (*lib*name.*b*) just as they do for *coff* object libraries (*lib*name.*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 *–klname* 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 contains coding hints recommended to increase optimizing opportunities for the global optimizer (*uopt*). Apply these recommendations to your code whenever possible.

C and Fortran Programs

The following suggestion applies to both 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, 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 which return values instead of pointer parameters.

Do while. Use *do while* instead of *while* or *for* when possible. 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 will 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. You can often avoid 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

In this example, because the statement **p++ = 0* might modify *len*, 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.

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
       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:
```

Two methods for increasing the efficiency of this example might be: using subscripts instead of pointers; and 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.

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 != 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. Specifying *len* as a local variable or formal argument (as shown below) prevents aliasing and allows 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
       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:
```

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

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. For example, the following code uses the value of *n--* as a condition for the *while* loop, which is a convoluted way of performing the loop while *n* is non-zero:

```
while (n--) {
    ...
}
```

In the following code it is obvious that the loop is performed when *n* is non-zero:

```
while (n != O) {
    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). 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.

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:

```
/* OK: */
if ( i == 1 ) c = '1';
    else c = '0';

/* More efficient: */
```

```
c = "01"[i];
```

- 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, always declare as the first four parameters those variables that are most frequently manipulated in the called routine.
- Use word-size variables instead of smaller ones if enough space is available. This practice can take more space, but it is more efficient.

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). There are two reasons for this. First: since the compiler knows such a variable will always be greater than or equal to zero, it performs optimizations that would not otherwise be possible. Second: 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.