Chapter 4

Using the Performance Tools

This chapter explains how to use performance tools to reduce the execution time of your programs. For information about the compiler optimization facilities, see Chapter 5, “Optimizing Program Performance.”

This chapter covers the following topics:

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

Although it may be possible to obtain short-term speed increases by relying on unsupported or undocumented quirks of the compiler system, it’s a bad idea to do so. Any such “features” may break in future releases of the compiler system. The best way to produce efficient code that can be trusted to remain efficient is to follow good programming practices; in particular, choose good algorithms and leave the details to the compiler.

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

Overview of Profiling

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

The compiler system provides the following types of profiling:

- *Program counter (pc) sampling*, which measures the amount of execution time spent in various parts of the program. This statistical data is obtained by periodically sampling the program counter. By default, *prof* generates pc sampling data. For more information, see “pc Sampling.”
- *Basic block counting*, which counts the execution of basic blocks (a basic block is a sequence of instructions that is entered only at the beginning and exits only at the end). It produces an exact count of the number of times each basic block is executed. For more information, see “Basic Block Counting.”
- *Inclusive basic block counting*, which counts basic blocks and generates a call graph. By propagating regular counts to callers of a routine, *prof* provides inclusive basic block counting. For more information, “Inclusive Basic Block Counting.”

Profiling With *prof*

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

The topics covered below include:

- “Running the Profiler”
- “prof Options”

Running the Profiler

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

prof Syntax

The syntax for *prof* is:

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

options	One of the keywords or keyword abbreviations shown in Table 4-1. (Specify either the entire name or the initial character of the option, as indicated in the table.)
prog_name	Specifies the name of the program whose profile data is to be profiled.
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.

prof Defaults

The *prof* program has these defaults:

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

  You may want to use the **–merge** option when you have more than one profile data file. This option merges the data from several profile files into one file.
- If you specify **–pixie** and do not specify *profile_filename*, then *prof* looks for *prog_name.Counts* and provides basic block count information if this file is present.
- If you specify *profile_filename*(s), *prof* determines the file type based on its content: a *prof*- or *pixie*-mode file.

prof Options

Table 4-1 lists *prof* options. Options that apply to basic block counting are indicated as such. For more information, see the *prof*(1) reference page. Click the word prof to view the page.

Table 4-1 Options for *prof*

Name	Result
–c[**lock**] *n*	A basic-block-counting option. This option lists the number of seconds spent in each routine, based on the CPU clock frequency *n*, expressed in megahertz.
–d[**is**]	Disassembles and annotates the analyzed object code with cycle times or number of pc samples.
–dso [*dso_name*]	Applies *prof* analysis to only the named DSO. If you don't specify *dso_name*, *prof* prints a list of applicable DSO names.
–e[**xclude**] *procedure_name*	Excludes information on the procedures specified by *procedure_name*. If you specify uppercase **–E**, *prof* also omits that procedure from the base upon which it calculates percentages.
–feedback	Produces a file with information that the compiler system can use to decide how to arrange procedures in the binary. The default is to use **–procedures** in positioning procedures in the binary. This option has to be used with **–pixie**. If **–gprof** is also specified, procedures are ordered using call graph counts, rather than invocation counts. The feedback file is named *a.out.fb* or *lib*so*.fb*. See Table 2-2 and "Rearranging Procedures With cord."
–gprof	A basic block counting option to be used with *prof*. Calculates cycles and propagates basic block counting to a procedure's callers proportionately. For more information, see "Inclusive Basic Block Counting."
–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**]	Lists statistics for each line of source code.

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

Name	Result
–m[**erge**] *filename*	Merges the input files into *filename* (the default is *mon.out*), allowing you to specify the name of the merged file (instead of several filenames) on subsequent profiler runs. This option is useful when using multiple input files of profile data.
–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.
–pcsample	Tells *prof* that the data to be analyzed is from pc sampling. This is the default. This option and **–pixie** are mutually exclusive.
–pixie	A basic-block-counting option. Indicates that information is to be generated on basic block counting, and that the *prog_name.Counts* files produced by *pixie* are to be used by default. This option and **–pcsample** are mutually exclusive.
–p[**rocedures**]	Lists the time spent in each procedure.
–q[**uit**] *n*	Condenses output listings by truncating unwanted lines. You can specify *n* in three 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; **ncum%**, 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 and memory access time
- Time spent in user 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 examples showing the use of various *prof* options. Specifically, this section covers:

- "Obtaining pc Sampling"
- "Creating Multiple Profile Data Files"
- "pc Sampling Frequency"
- "Examples of Using prof to Obtain pc Sampling Analysis Data"

Obtaining pc Sampling

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

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

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

   ```
   % cc –c myprog.c
   ```

2. Link the object file created in Step 1.

   ```
   % cc –p –o myprog myprog.o
   ```

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

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

   ```
   % myprog
   ```

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

4. Run the profile formatting program *prof*.

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

prof extracts information from *mon.out* and prints it in an easily readable format. If *mon.out* exists, it is overwritten. Therefore, rename each *mon.out* to save its output.

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

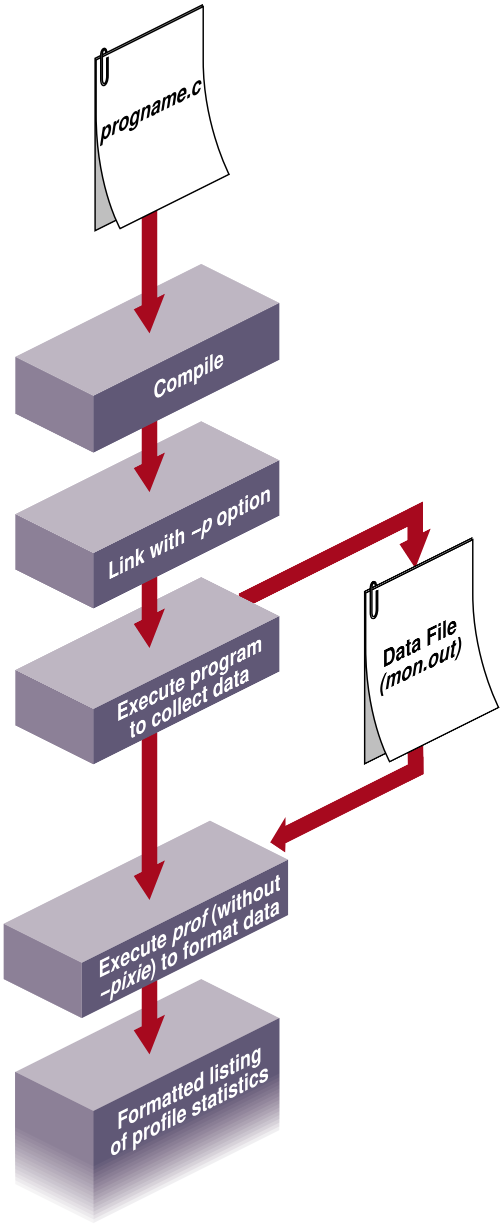

Figure 4-1 How pc Sampling Works

Creating Multiple Profile Data Files

When you run a program using pc sampling, raw data is collected and saved in the profile data file *mon.out*. To collect profile data in several files, or to specify a different name for the profile data file, set the environment variable PROFDIR, using the appropriate method from Table 4-2.

Table 4-2 Setting a PROFDIR Environment Variable

C shell and tcsh	**Bourne shell and Korn shell**
setenv PROFDIR *dirname*	PROFDIR=*dirname*; export PROFDIR

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

pc Sampling Frequency

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

```
PROF_SAMPLING=1
```

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

Examples of Using *prof* to Obtain pc Sampling Analysis Data

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

Example Using prof –pcsample

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

```
Profile listing generated Fri May 13 14:19:12 1994
    with:       prof -pcsample espresso
-------------------------------------------------------------------------------

samples   time    CPU    FPU   Clock   N-cpu  S-interval Countsize
   4160    42s  R8000  R8010  75.0MHz   1     10.0ms     4(bytes)

Each sample covers 4 bytes for every 10.0ms ( 0.02% of 41.6000sec)

-------------------------------------------------------------------------------
  -p[rocedures] using pc sampling.
  Sorted in descending order by the number of samples in each procedure.
  Unexecuted procedures are excluded.
-------------------------------------------------------------------------------

samples   time(%)      cum time(%)      procedure (file)

    847   8.5s( 20.4)  8.5s( 20.4)  massive_count
(espresso:/usr/people/guest/enjoy/008.espresso/cofactor.c)
    410   4.1s(  9.9)   13s( 30.2)       cofactor
(espresso:/usr/people/guest/enjoy/008.espresso/cofactor.c)
    341   3.4s(  8.2)   16s( 38.4)   setp_implies
(espresso:/usr/people/guest/enjoy/008.espresso/set.c)
    325   3.2s(  7.8)   19s( 46.2)  elim_lowering
(espresso:/usr/people/guest/enjoy/008.espresso/expand.c)
    303     3s(  7.3)   22s( 53.5)    essen_parts
(espresso:/usr/people/guest/enjoy/008.espresso/expand.c)
 ...
```

In the above listing:

- The `samples` column reports the number of samples in each procedure, sorted in descending order. For example, there were `847` samples for the procedure `massive_count`.
- The `time(%)` column lists the number of seconds and percentage of execution time spent in each procedure. For example, there were `8.5` seconds (`20.4`% of execution time) spent in `massive_count`.
- The `cum time(%)` column lists the percentage of the total execution time spent in each procedure. For example, there were `16` seconds (`38.4`% of total execution time) were spent cumulatively in the `massive_count`, `cofactor`, and `setp_implies` procedures. Note that this does not imply that these routines called each other; they may have executed sequentially.

- The `procedure (file)` column prints the procedure name and its source file. For example, the source file containing the `massive_count` procedure is `cofactor.c`.

Basic Block Counting

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

- "Using pixie"
- "Obtaining Basic Block Counts"
- "Summing Basic Block Count Results From Multiple Runs"
- "Profiling Multiprocessed Executables"

Using *pixie*

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

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

pixie Syntax

The syntax for *pixie* is:

```
pixie prog_name [options]
```

prog_name	Name of the input program.
options	One of the keywords listed in Table 4-3.

pixie Options

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

Table 4-3 Options for *pixie*

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

Obtaining Basic Block Counts

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

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

   ```
   % cc –o myprog myprog.c
   ```

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

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

   ```
   % pixie myprog
   ```

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

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

   ```
   % setenv LD_LIBRARY_PATH .
   ```

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

   ```
   % myprog.pixie
   ```

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

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

   ```
   % prof –pixie myprog myprog.Counts
   ```

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

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

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

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

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

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

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

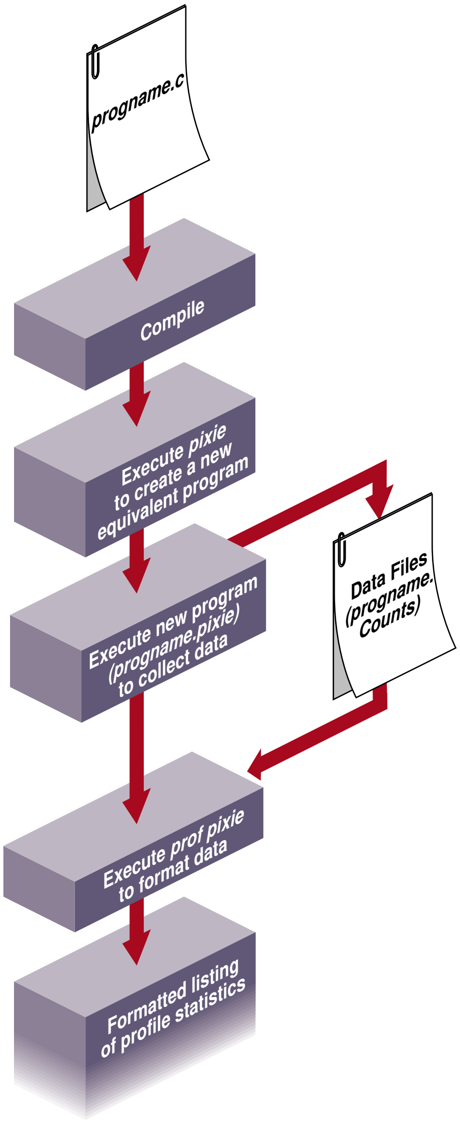

Figure 4-2 How Basic Block Counting Works

Examples of Basic Block Counting

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

Example Using prof –pixie –invocations

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

```
% prof –pixie –i espresso
```

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

         calls(%)    cum%      inst %coverage procedure (file)

   3055229(25.22)   25.22        26     25.00 full_row
(espresso:/usr/people/guest/enjoy/008.espresso/setc.c)
    966541( 7.98)   33.20        26     25.00 set_or
(espresso:/usr/people/guest/enjoy/008.espresso/set.c)
    772942( 6.38)   39.58        26     25.00 cleanfree
(espresso:/work/irix/lib/libc/libc_64/gen/malloc.c)
    611793( 5.05)   44.63        26     25.00 setp_implies
...
```

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

- The `calls(%)` column reports the number of calls (and the percentage of total calls) per procedure. For example, there were `3055229` calls (or `25.22`% of the total) spent in the procedure **full_row**.
- The `cum%` column shows the cumulative percentage of calls. For example, `25.22`% of all calls were spent in `full_row`.

- The `inst` column shows the number of instructions executed for a procedure. For example, there were `26` instructions in the procedure `full_row`.
- The `%coverage` column reports the percentage of instructions executed. For example, `25.00`% of instructions were executed in `full_row`.
- The `procedure (file)` column lists the procedure and its file. For example, the first line reports statistics for the procedure `full_row` in the file `setc.c`.

Example Using prof –pixie –heavy

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

```
% prof -pixie -heavy espresso
-------------------------------------------------------------------------------
Profile listing generated Fri May 13 14:28:56 1994
    with:      prof -pixie -heavy espresso
-------------------------------------------------------------------------------
...
* -h[eavy] using basic block counts.
*     Sorted in descending order by number of cycles per line.
*     Unexecuted lines are excluded.
*     The insts column contains distinct executed instructions for this line.
-------------------------------------------------------------------------------

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

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

- The `cycles(%)` column shows the total number of program cycles (and percentage of the total number). For example, there were `77948528` cycles (`4.95`% of the total number) for the procedure `cofactor`.
- The `cum%` column shows the cumulative percentage of the total program cycles. For example, `4.95`% of all program cycles were spent in `cofactor`. The first three procedures used `13.04%` of the total cycles.
- The `line` column lists the line number of the procedure: `57`.
- The `insts` column reports the number of distinct instructions that were executed at least once. For example, line `57` had `40` instructions.

Example Using prof –pixie –lines

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

```
% prof -pixie -lines espresso
```

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

In the previous listing:

- The `cycles(%)` column lists the number of program cycles (and the percentage of the total cycles) for each procedure. For example, there were `856768` program cycles (`0.05`% of the total) for `massive_count`.
- The `cum%` column shows the cumulative percentage of the total program cycles. For example, `0.05`% of all program cycles were spent in `massive_count`.
- The `line` and `insts` columns report the procedure's line number, and number of distinct instructions. For example, for the procedure `massive_count`, lines `121` and `128` each generated `12` instructions that were executed at least once, and line `134` generated `16` instructions that were executed at least once.

Example Using prof –pixie –quit

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

Consider the following sample listing:

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

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

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

```
prof –quit 4
prof –quit 5%
prof –quit 53cum%
```

Example Using prof –pixie –procedures

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

```
% prof –pixie –procedures espresso
```

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

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

In the above listing:

- The `cycles(%)` column lists the number of program cycles (and percentage of the total) used. For example, `massive_count` used `360331656` program cycles (`22.90%` of the total cycles).
- The `cum %` column reports the cumulative total of all cycles used. For example, `massive_count`, `cofactor`, and `elim_lowering` used `44.01%` of the cycles.

- The `seconds` column lists the time used by the procedure (the clock rate, 75 megahertz, is omitted in this example). For example, there were `4.80` seconds used by `massive_count`.
- The `cycles/call` and `bytes/line` columns report the average cycles per call, and the bytes of code per line of source text, respectively. For example, `massive_count` used an average of `4626` cycles per call, and had `93` bytes of generated code per line of source text.
- The `procedure (file)` column lists the procedure, `massive_count`, and its source file, `cofactor.c`.

Example Using prof –pixie –procedures –clock

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

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

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

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

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

Example Using prof –pixie –dis

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

```
-------------------------------------------------------------------------------
Profile listing generated Fri May 13 14:37:07 1994
    with:       prof –pixie –dis espresso
-------------------------------------------------------------------------------

Total cycles    Total Time  Instructions Cycles/inst    Clock
  1573247788        20.98s    2093722528       0.751  75.0MHz
...
/usr/people/guest/enjoy/008.espresso/main.c
main: <0x1001c90c–0x1001e238>
    219 total cycles(0.00%) invoked 1 times, average 219 cycles/invocation
  [20] 0x1001c90c       0x67bdfe40      daddiu  sp,sp,-448      # 0
                <1 cycle stall for following instruction>
  [20] 0x1001c910       0xffbf0170      sd      ra,368(sp)      # 2
  [20] 0x1001c914       0xffb00158      sd      s0,344(sp)      # 3
  ...
  [152] 0x1000b178 0x1540000d bne t2,zero,0x1000b1b0 # 23
  [152] 0x1000b17c 0x016c5823 subu t3,t3,t4 # 23
  Preceding branch executed 1000 times, taken 0 times.
  ^--- 3000 total cycles(0.01%) executed 1000 times, average 3 cycles.---^
  ...
```

The previous listing shows statistics about the procedure `main` in the file `main.c` and lists the beginning and ending addresses of `main`: `<0x1001c90c–0x1001e238>`

- The first column lists the line number of the instruction: `[20]`
- The second column lists the beginning address of the instruction: `0x1001c90c`.
- The third column shows the instruction in hexadecimal: `0x67bdfe40`.
- The next column reports the assembler form (mnemonic) of the instruction: `daddiu sp,sp,–448`.
- The last column reports the cycle in which the instruction executed: `# 2`

Other information includes:

- The number of times an above branch was executed and taken:
```
Preceding branch executed 1000 times, taken 0 times
```
- The total number of cycles in a basic block and the percentage of the total cycles for that basic block, the number of times the branch terminating that basic block was executed, and the number of cycles for one execution of that basic block:
```
300 total cycles(0.01%) executed 1000 times, average3 cycles.
```
- Any cycle stalls (cycles that were wasted)

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

Summing Basic Block Count Results From Multiple Runs

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

Use the following procedure to average *prof* results:

1. Compile and link the input file. Do not use the **–p** option. For example, consider the input file *myprog.c*:
```
% cc -o myprog myprog.c
```
   The *cc* compiler compiles *myprog.c* and saves the executable as *myprog*.
2. Run the profiling program *pixie*, adding the **–pids** option, so that each run produces a distinct file.
```
% pixie myprog -pids
```
   pixie generates the modified program *myprog.pixie*.
3. Run the profiled program as many times as desired. The instrumented program creates a *myprog.Counts.pid* file with each execution and appends the process ID.
```
% myprog.pixie < input1 > output1
% myprog.pixie < input2 > output2
% myprog.pixie < input3 > output3
```

4. Create the report.

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

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

Inclusive Basic Block Counting

The basic block counting explained in the previous section allows you to measure ideal time spent in each procedure, but doesn't propagate the time up to the caller of that procedure. For example, basic block counting may tell you that procedure **sin(x)** took the most time, but significant performance improvement can only be obtained by optimizing the callers of **sin(x)**. Inclusive basic block counting solves this problem.

Inclusive basic block counting calculates cycles just like regular basic block counting, and then propagates it proportionately to all its callers. The cycles of procedures obtained using regular basic block counting (called exclusive cycles), are divided up among its callers in proportion to the number of times they called this procedure. For example, if **sin(x)** takes 1000 cycles, and its callers, procedures **foo()** and **bar()**, call **sin(x)** 25 and 75 times respectively, 250 cycles are attributed to **foo()** and 750 to **bar()**. By propagating cycles this way, **__start** ends up with all the cycles counted in the program.

Note: The assumption made in propagating times from a callee to a caller is that all calls are equivalent, so that the time attributed is divided equally for all calls. For some functions (**sin()**, for example), this assumption is plausible. For others (matrix multiply, for example), the assumption can be *very misleading*. If **foo()** calls **matmult()** 99 times for 2X2 matrices, while **bar()** calls it once for 100X100 matrices, the inclusive time report will attribute 99% of **matmult()**'s time to **foo()**, but actually almost all the time derives from **bar()**'s one call.

If you are familiar with the *gprof* command, you will see the similarity. The difference is that inclusive basic block counting uses ideal cycles calculated by *prof*, rather than real sampled time. The output of *prof* is very similar to the traditional *gprof* output, except that it uses ideal cycles instead of real execution time.

The steps to obtain inclusive times are essentially the same as those explained in "Basic Block Counting." Additionally, you have to instrument (using *pixie*) your program with an additional **–gprof** option, and invoke *prof* with an additional **–gprof** option. For more information on *gprof*, see the *gprof*(1) reference page. To view the page, click the word gprof.

Syntax

The syntax is:

```
pixie -gprof [options] prog_name
```

This tells *pixie* to instrument the program to also provide call graph support.

To invoke *prof*, use the following syntax:

```
prof -gprof -pixie [options] prog_name
```

This produces an inclusive cycles report. The **–gprof** information is bundled in the *.Counts* file. Figure 4-3 illustrates how inclusive basic block counting works.

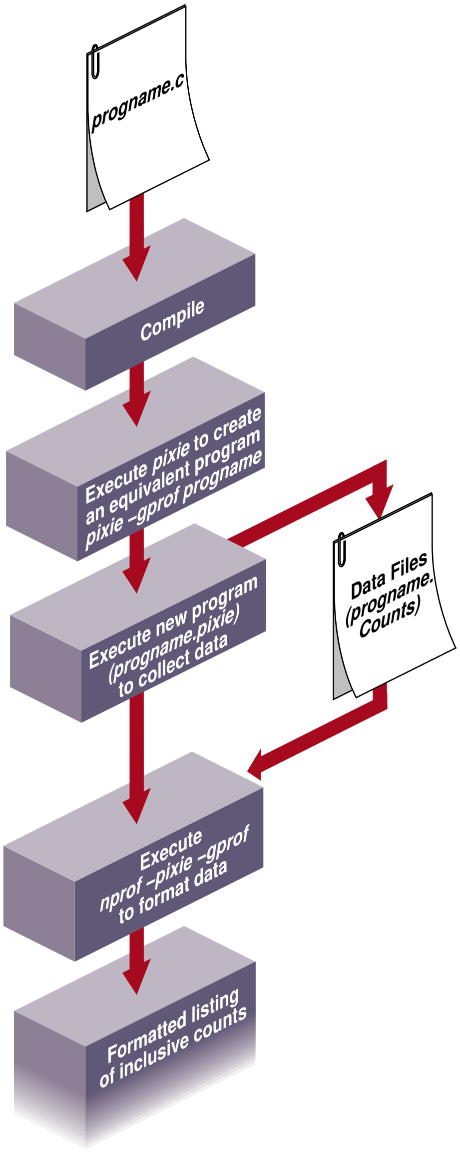

Figure 4-3 How Inclusive Basic Block Counting Works

Example of prof –gprof –pixie

Here's part of a sample output obtained by using the **–gprof** and **–pixie** options. The fields are explained in detail before the output is generated (the format of the output has been adjusted slightly).

```
call graph profile:
          The sum of self and descendents is the major sort
          for this listing.

          function entries:

index     the index of the function in the call graph
          listing, as an aid to locating it.

cycles(%cycles)
          the total cycles (percentage of total) of the program
          accounted for by this function and its descendents.

self(per)
          cycles (percent of total) spent in this function itself.

kids(per)
          cycles (percent of total) spent in the descendents of
          this function on behalf of this function.

called    the number of times this function is called (other
          than recursive calls).

self      the number of times this function calls itself recursively.

name      the name of the function, with an indication of
          its membership in a cycle, if any.

index     the index of the function in the call graph
          listing, as an aid to locating it.

          parent listings:

self*
          cycles of this function's self time which is due to calls from
          this parent.

kids*
          cycles of this function's  descendent time which is due to
```

```
          calls from this parent.

called**  the number of times this function is called by this parent.
          This is the numerator of the fraction which divides up the
          function's time to its parents.

total*    the number of times this function was called by all of its parents.
          This is the denominator of the propagation fraction.

parents   the name of this parent, with an indication of the
          parent's membership in a cycle, if any.

index     the index of this parent in the call graph
          listing, as an aid in locating it.

          children listings:

self*
          cycles of this child's self time
          which is due to being called by this function.

kids*
          cycles of this child's descendent's time which is due to
          being called by this function.

called**  the number of times this child is called by this function.  This
          is the numerator of the propagation fraction for this child.

total*    the number of times this child is called by all functions.
          This is the denominator of the propagation fraction.

children  the name of this child, and an indication of its
          membership in a cycle, if any.

index     the index of this child in the call graph listing,
          as an aid to locating it.

          * these fields are omitted for parents (or children) in
          the same cycle as the function.  If the function (or child)
          is a member of a cycle, the propagated times and propagation
          denominator represent the self time and descendent time of the
          cycle as a whole.

          ** static-only parents and children are indicated
          by a call count of 0.
```

```
                   cycle listings:
                   the cycle as a whole is listed with the same
                   fields as a function entry.  Below it are listed
                   the members of the cycle, and their contributions
                   to the time and call counts of the cycle.

                   All times are in milliseconds.
---------------------------------------------------------------------------

                       self            kids       called/total     parents
                     index  cycles(%)    self(per)       kids(per)  called+self     name
          index
                      self             kids       called/total    children

[1]   7825(99.47%)  36( 0.46%)       7789(99.54%)    0         __start [1]
                    33               7753            1/1          main [2]
                     3                  0            1/1  __readenv_sigfpe [76]

----------------------------------------------------------------------------

                     33              7753           1/1       __start [1]
[2]   7786(98.97%)   33( 0.42%)      7753(99.58%)   1            main [2]
                     18              4559           1/1         _MAIN [3]
                    183              2258           1/1        f_exit [8]
                    115               384           5/5       signal [25]
                     19               217           1/1          exit [33]

----------------------------------------------------------------------------

                     18             4559          1/1            main [2]
[3]   4577(58.18%)   18( 0.39%)     4559(99.61%)  1             _MAIN [3]
                      9             4082          1/1        s_wsle64 [4]
                      9              318          1/1       do_Lio64 [28]
                      8              133          1/1         e_wsle [43]

----------------------------------------------------------------------------

                      9             4082         1/1           _MAIN [3]
[4]   4091(52.00%)    9( 0.22%)     4082(99.78%) 1          s_wsle64 [4]
                     22             4060         1/1      s_wsle_com [5]

----------------------------------------------------------------------------

                     22             4060         1/1        s_wsle64 [4]
```

```
[5]  4082(51.89%)   22( 0.54%)     4060(99.46%) 1       s_wsle_com [5]
                    28              2016        1/1         c_le [10]
                   201              1815        1/2         f_init [7]

------------------------------------------------------------------------------

[6]  4033(51.26%)  402( 9.97%)     3631(90.03%) 2+8  <cycle 1 as a whole> [6]
                   142( 0.02)      3147( 0.42)  5     f_init <cycle 1> [7]
                   260( 0.03)       484( 0.06)  5   map_luno <cycle 1> [21]

------------------------------------------------------------------------------

                                                4   map_luno <cycle 1> [21]
                   201             1815         1/2       s_wsle_com [5]
[7]  3289(41.81%)  142( 4.32%)     3147(95.68%) 5   f_init <cycle 1> [7]
                    15             1896         1/1   init_test_and_set [13]
                    42              930         1/2         _calloc [12]
                    51              114         3/3      f77canseek [40]
                    45               27         3/7         _isatty [39]
                    15                0         5/5         _getpid [63]
                    12                0         4/9 _mips2_test_and_set [59]
                                                4    map_luno <cycle 1> [21]

------------------------------------------------------------------------------

                   183             2258         1/1             main [2]
[8]  2441(31.03%)  183( 7.50%)     2258(92.50%) 1             f_exit [8]
                   288             1967        32/32          f_clos [9]
                     3                0         1/1    __checktraps [79]

------------------------------------------------------------------------------
....
```

Profiling Multiprocessed Executables

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

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

Rearranging Procedures With *cord*

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

- "cord Syntax"
- "cord Options"
- "Example Using cord"

cord Syntax

The syntax for *cord* is:

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

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

The reorder file is produced by the **–feedback** option to *prof* (for information on *prof* and the **–feedback** option, see Table 4-1, Options for *prof*, or the *prof*(1) reference page). The default reorder file is named *prog.fb*, if you do not specify *reorder_file*. You can also produce a feedback file using the WorkShop performance tools. WorkShop includes a tool, **cvcord**, that can combine multiple working set files into an optimized feedback file.

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

cord Options

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

Table 4-4 Options for *cord*

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

Example Using *cord*

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

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

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

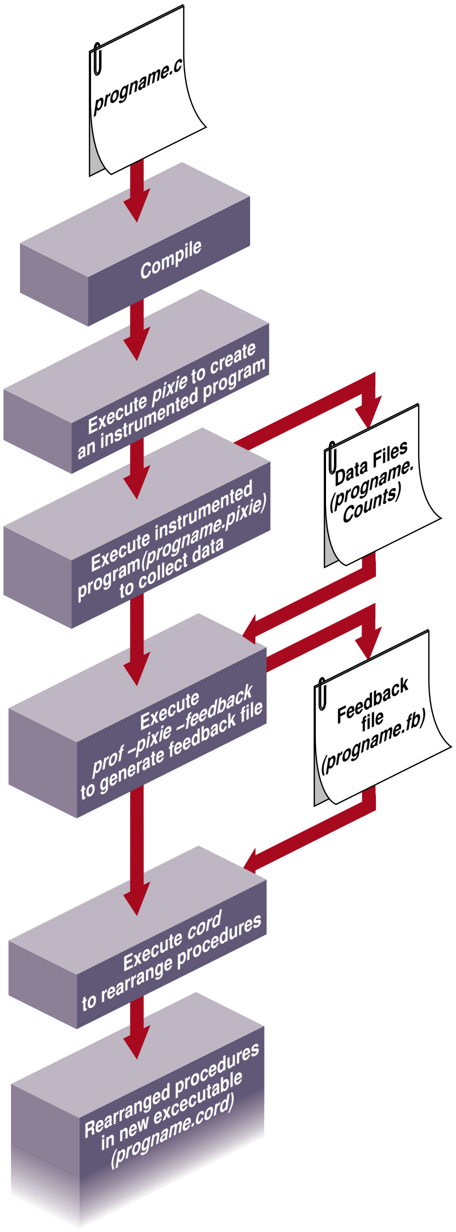

Figure 4-4 How *cord* Works

The procedure rearrangement depends on the data produced by the profiled runs of the executable. If these profiled runs approximate the actual use of the executable, the resultant binary is close to being optimally rearranged. Design your profiled runs accordingly.

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

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

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

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