Performance Profiling Tools

The tools discussed in this chapter have facilities for timing programs and obtaining performance analysis data. Some tools work only with the C programming language, while others work on modules written in any language.

This document is organized into the following sections:

Introduction

Performance analysis tools provide a variety of analysis levels. Analysis levels vary from simple timing of a command to a statement-by-statement analysis of a program. Select the level of granularity based on the amount of detail and optimization you wish to perform. Here are the performance analysis tools available from the simplest to the most detailed:

`prof`

Generates a profile for the modules in a program, showing which modules are most time-intensive. This tool is included with the Solaris operating system environment.

`gprof`

Generates not only a profile, but also a call graph showing which modules call other modules, and which modules are called by other modules. The *call graph* can sometimes point out areas where removing calls can speed up a program. This tool is included with the Solaris operating system environment.

`tcov`

Generates a detailed statement-by-statement analysis of program modules. There are two `tcov` implementations: the original `tcov`, and a new version referred to as `tcov` Enhanced, which supports all compiler platforms.

Profile Feedback

A mechanism to gather information about the runtime behavior of your program. The compiler uses this information to optimize your program.

`prof`--Generate the Profile of a Program

A *profile* of a program display assists in optimizing performance. Obtaining a profile is the next step after simple timing (which you can do with the `time`(1) utility). More detailed analysis is provided by the call-graph profile and the code coverage tools described later.

For example, a C source file called index.assist.c produces a program called `index.assist`. To compile a program for profiling, use either the -p or `-qp` option to the compiler:

```
% cc -p -o index.assist index.assist.c
```

Now run the `index.assist` program. Each time it runs, profiling data is sent to a file called mon.out at the end of the run. Every time you run the program a new mon.out file is created, overwriting the old version. You then use the prof command to interpret the results of the profile:

```
% index.assist

% ls mon.out

mon.out

% prof index.assist
```

Figure 1 on page 3 shows a sample `prof` output.

%Time	Seconds	Cumsecs	#Calls	msecs/call	Name
19.4	3.28	3.28	11962	0.27	compare_strings
15.6	2.64	5.92	32731	0.08	_strlen
12.6	2.14	8.06	4579	0.47	__doprnt
10.5	1.78	9.84			mcount
9.9	1.68	11.52	6849	0.25	_get_field
5.3	0.90	12.42	762	1.18	_fgets
4.7	0.80	13.22	19715	0.04	_strcmp
4.0	0.67	13.89	5329	0.13	_malloc
3.4	0.57	14.46	11152	0.05	_insert_index_entry
3.1	0.53	14.99	11152	0.05	_compare_entry
2.5	0.42	15.41	1289	0.33	lmodt
0.9	0.16	15.57	761	0.21	_get_index_terms
0.9	0.16	15.73	3805	0.04	_strcpy
0.8	0.14	15.87	6849	0.02	_skip_space
0.7	0.12	15.99	13	9.23	_read
0.7	0.12	16.11	1289	0.09	ldivt
0.6	0.10	16.21	1405	0.07	_print_index
. . (The rest of the output is insignificant)					

Figure 1 Sample `prof` Output

This display points out that most of the program running time is spent in the `compare_strings` routine; after that, most of the time is spent in the `_strlen` library routine. To make improvements to this program, concentrate on the `compare_strings` function.

Let's interpret the results of the profiling run-through. The results are listed under these column headings:

> `%Time`--The percentage of the total runtime of the program consumed by this routine.
>
> `Seconds`--The total number of seconds accounted for by this function.
>
> `Cumsecs`--A running sum of the number of seconds accounted for by this function and those listed above it.
>
> `#Calls`--The number of times this routine is called.
>
> msecs/call--The number of milliseconds this routine consumes each time it is called.
>
> `Name`--The name of the routine.

What results can be derived from the profile data? The `compare_strings` function consumes nearly 20% of the total time. To improve the runtime of `index.assist`, either improve the algorithm that `compare_strings` uses, or cut down the number of calls to `compare_strings`.

It is not obvious from the flat call graph that `compare_strings` is heavily recursive, but you can deduce this by using the call graph profile described in the next section. In this particular case, improving the algorithm also reduces the number of calls.

`gprof`--Generate a Call Graph Profile

While the flat profile can provide valuable data for performance improvements, sometimes the data obtained is not sufficient to point out exactly where improvements can be made. A more detailed analysis can be obtained by using the call graph profile to display a list identifying which modules are called by other modules, and which modules call other modules. Sometimes removing calls altogether can result in performance improvements.

Using the same `index.assist` program as an example, compile the program for call graph profiling. Use the `-xpg` option to the C compiler or the `-pg` option to other compilers:

```
% cc -xpg -o index.assist index.assist.c
```

Now run the `index.assist` program as before. When a program is compiled in this manner, each time it is run, call-graph profile data is sent to a file called `gmon.out` at the end of the run. This file is re-created each time you run the program. Use the `gprof` command to interpret the results of the profile. The output from `gprof` is voluminous--it's usually intended that you take the summaries away and read them later. For that reason, redirect the output to a file, `/tmp/g.output`:

```
% index.assist

% ls gmon.out

gmon.out

% gprof index.assist > /tmp/g.output
```

The output from `gprof` consists of two major items:

- The full call graph profile. Figure 2 on page 6 shows fragments of output from a profiling run.
- The "flat" profile, similar to the summary the `prof` command supplies.

The output from `gprof` contains an explanation of what the various parts of the summary mean. `gprof` also identifies the granularity of the sampling:

```
granularity: each sample hit covers 4 byte(s) for 0.14% of 14.74
seconds
```

This is part of the call graph profile:

				called/total parents		
index	%time	self	descendents	called+self	name	index
				called/total children		

		0.00	14.47	1/1	start	[1]
[2]	98.2	0.00	14.47	1	_main	[2]
		0.59	5.70	760/760	_insert_index_entry	[3]
		0.02	3.16	1/1	_print_index	[6]
		0.20	1.91	761/761	_get_index_terms	[11]
		0.94	0.06	762/762	_fgets	[13]
		0.06	0.62	761/761	_get_page_number	[18]
		0.10	0.46	761/761	_get_page_type	[22]
		0.09	0.23	761/761	_skip_start	[24]
		0.04	0.23	761/761	_get_index_type	[26]
		0.07	0.00	761/820	_insert_page_entry	[34]

				10392	_insert_index_entry	[3]
		0.59	5.70	760/760	_main	[2]
[3]	42.6	0.59	5.70	760+10392	_insert_index_entry	[3]
		0.53	5.13	11152/11152	_compare_entry	[4]
		0.02	0.01	59/112	_free	[38]
		0.00	0.00	59/820	_insert_page_entry	[34]
				10392	_insert_index_entry	[3]

Figure 2 Sample `gprof` Output

Assuming there are 761 lines of data in the input file to the `index.assist` program, the following conclusions can be drawn:

- `fgets` is called 762 times. The last call to `fgets` returns an end-of-file.
- The `insert_index_entry` function is called 760 times from main.
- In addition to the 760 times `insert_index_entry` is called from `main,` `insert_index_entry` also calls itself 10,392 times. `insert_index_entry` is heavily recursive.
- `compare_entry` (which is called from `insert_index_entry`) is called 11,152 times, which is equal to 760+10,392 times. There is one call to `compare_entry` for every time `insert_index_entry` is called. This is as it should be. If there were a discrepancy in the number of calls, you could suspect some problem in the program logic.
- `insert_page_entry` is called 820 times in total: 761 times from main while the program is building index nodes, and 59 times from `insert_index_entry`. This frequency indicates there are 59 duplicated index entries, so their page number entries are linked into a chain with the index nodes. The duplicate index entries are then freed; hence the 59 calls to `free`().

tcov--Statement-Level Analysis

tcov gives line-by-line information on how a program executes. It produces a copy of the source file, annotated to show which lines are used and how often. It also summarizes information about basic blocks.

tcov works with both C and C++ programs, but tcov does not support files that contain #line or #file directives. tcov does not enable test coverage analysis of the code in the #include header files. Applications compiled with -xa (C), -a (other compilers), and +d (C++) run slower than normal. The +d option inhibits expansion of C++ inline functions, and updating the .d file for each execution takes considerable time.

Using the index.assist Program for Use With tcov

Using the index.assist program, compile for use with tcov. To compile a program for code coverage, use the -xa option to the C compiler, or the -a option of other compilers:

```
% cc -xa -o index.assist index.assist.c
```

The C compiler generates an index.assist.d file, containing database entries for the basic blocks present in index.assist.c. When the program index.assist is run till completion, the compiler updates the index.assist.d file. The count of basic blocks cannot exceed the value represented by an unsigned int.

The index.assist.d file is created in the directory specified by the environment variable TCOVDIR, which is set as follows:

In a Bourne Shell:

```
$ TCOVDIR=directory

$ export TCOVDIR
```

In a C Shell:

```
% setenv TCOVDIR directory
```

If TCOVDIR is not set, index.assist.d is created in the current directory.

Having compiled index.assist.c, run index.assist.

```
% index.assist

% ls *.d

index.assist.d
```

Now, run tcov to produce a file containing the summaries of execution counts for each statement in the program. tcov uses the index.assist.d file to generate an index.assist.tcov file containing an annotated list of your code. The output shows the number of times each source statement is executed. At the end of the file, there is a short summary.

```
% tcov index.assist.c

% ls *.tcov

index.assist.tcov
```

Figure 3 shows a small fragment of the C code from one of the modules of index.assist--the module in question is the insert_index_entry function called so recursively.

```
          struct index_entry *

11152  -> insert_index_entry(node, entry)

          structindex_entry *node;
          struct index_entry *entry;
          {
            int result;
            int level;
              result = compare_entry(node, entry);
              if (result == 0) {          /* exact match */
                                          /* Place the page entry for the duplicate */
                                          /* into the list of pages for this node */
59     ->       insert_page_entry(node, entry->page_entry);
              free(entry);
              return(node);
              }
11093  ->     if (result > 0)             /* node greater than new entry -- */
                                          /* move to lesser nodes */
3956   ->       if (node->lesser != NULL)
3626   ->           insert_index_entry(node->lesser, entry);
                else {
330    ->         node->lesser = entry;
                  return (node->lesser);
                }
                else                      /* node less than new entry -- */
                                          /* move to greater nodes */
7137   ->         if (node->greater != NULL)
6766   ->             insert_index_entry(node->greater, entry);
                else {
371    ->         node->greater = entry;
                  return (node->greater);
                }
          }
```

Figure 3 Sample `tcov` Output

The insert_index_entry function is called 11,152 times, as determined in the output from `gprof`. The numbers to the side of the C code show how many times each statement was executed.

Following is the summary tcov placed at the end of `index.assist.tcov`:

Top 10 Blocks		
	Line	Count
	240	21563
	241	21563

	245	21563
	251	21563
	250	21400
	244	21299
	255	20612
	257	16805
	123	12021
	124	11962
77	Basic blocks in this file	
55	Basic blocks executed	
71.43	Percent of the file executed	
	439144	Total basic block executions
	5703.17	Average executions per basic block

Figure 4 `tcov` Basic Block Coverage

A program compiled for code coverage analysis can be run multiple times (with potentially varying input); tcov can be used on the program after each run to compare behavior.

Creating Profiled Shared Libraries

It is possible to create a profiled shareable library and use it in place of one where binaries have already been linked. Include the `-xa` (C) or
`-a` (other compilers) option when creating the shareable libraries.
For example:

```
%cc -G -xa -o foo.so.1 foo.o
```

This command includes a copy of the `tcov` profiling subroutines in the shareable libraries, so that clients of the library do not need to relink. If a client of the library is also linked for profiling, then the version of the `tcov` subroutines used by the client is used to profile the shareable library.

Locking Files

`tcov` uses a simple file-locking mechanism for updating the block coverage database in the `.d` files. It employs a single file, `/tmp/tcov.lock`, for this purpose. Consequently, only one executable compiled with `-xa` (C) or `-a` (other compilers) should be running on the system. If the execution of the program compiled with the `-xa` (or `-a`) option is manually terminated, then the `/tmp/tcov.lock` file has to be deleted manually.

Files compiled with the `-xa` or `-a` option call the profiling tools subroutines automatically when a program is linked for `tcov` profiling. At program exit, these subroutines combine the information collected at runtime for file `xyz.f` with the existing profiling information stored in file `xyz.d`. To ensure this information is not corrupted by several people simultaneously running a profiled binary, a `xyz.d.lock` lock file is created for `xyz.d` for the duration of the update. If there are any errors in opening or reading `xyz.d` or its lock file, or if there are inconsistencies between the runtime information and the stored information, then the information stored in `xyz.d` is not changed.

An edit and recompile of `xyz.d` may change the number of counters in `xyz.d`. This is detected if an old profiled binary is run.

If too many people are running a profiled binary, the lock cannot be obtained. An error message similar to the following is displayed after a delay of several seconds:

```
tcov_exit: Failed to create lock file

'/tmp_mnt/net/rbbb/export/home/src/newpattern/foo.d.lock'

for coverage data file

'/tmp_mnt/net/rbbb/export/home/src/newpattern/foo.d'

after 5 tries. Is somebody else running this binary?
```

The stored information is not updated. This locking is safe across a network. Since locking is performed on a file-by-file basis, other files may be correctly updated.

The profiling subroutines attempt to deal with automounted file systems that have become unaccessible. They still fail if the file system containing a coverage data file is mounted with different names on different machines, or if the user running the profiled binary does not have permission to write to either the coverage data file or the directory containing it. Be sure all the directories are uniformly named and writable by anyone expected to run the binary.

Reading Errors From `tcov` Subroutines

The following error messages occur from `tcov` subroutines:

```
tcov_exit: Could not open coverage data file

'coverage data file name' because

'system error message string'.
```

The user running the binary lacks permission to read or write to the coverage data file. The problem also occurs if the coverage data file has been deleted.

```
tcov_exit: Could not write coverage data file

'coverage data file name' because

'system error message string'.
```

The user running the binary lacks permission to write to the directory containing the coverage data file. The problem also occurs if the directory containing the coverage data file is not mounted on the machine where the binary is being run.

```
tcov_exit: Failed to create lock file 'lock file name' for
coverage data file 'coverage data file name' after 5 tries. Is
someone else running this executable?
```

Too many users are trying to update a coverage data file at the same time. The problem also occurs if a machine has crashed while a coverage data file is being updated, leaving behind a lock file. In the event of a crash, the longer of the two files should be used as the post-crash coverage data file. Manually remove the lock file.

```
tcov_exit: Stdio failure, probably no memory left.
```

No memory is available, and the standard I/O package will not work. You cannot update the coverage data file at this point.

```
tcov_exit: Coverage data file path name too long (length
characters) 'coverage data file name'.
```

The lock file name contains six more characters than the coverage data file name; therefore, the derived lock file name may not be legal.

```
tcov_exit: Coverage data file 'coverage data file name' is too
short. Is it out of date?
```

A library or binary with `tcov` profiling enabled is simultaneously being run, edited, and recompiled. The old binary expects a coverage data file of a certain size, but the editing often changes that. If the compiler creates a new coverage data file at the same time the old binary is trying to update the old coverage data file, the binary may see an apparently empty or corrupt coverage file.

tcov Enhanced--Statement-level Analysis

tcov Enhanced gives line-by-line information on how a program executes. It produces a copy of the source file, annotated to show which lines are used and how often. It also gives a summary of information about basic blocks. tcov Enhanced works with both C and C++ source files.

The original tcov does not support files with #line or #file directives. tcov does not enable test coverage analysis of the code in the #include header files. Applications compiled with -xa (C), -a (other compilers), and +d (C++) run slower than norma. The +d option inhibits expansion of C++ inline functions, and updating the .d file for each execution takes considerable time.

tcov Enhanced overcomes some of the shortcomings of the original tcov. In particular, functionality has been modified to provide more complete support for C++. tcov Enhanced supports code found in #include header files and corrects a flaw that obscured coverage numbers for template classes and functions. In addition, tcov Enhanced runtime is more efficient than the original tcov runtime. Finally, tcov Enhanced is supported for all the platforms the compilers support.

Using the index.assist Program With tcov Enhanced

For a description of how to use the index.assist program for the original tcov see "Using the index.assist Program for Use With tcov" on page 8.

tcov Enhanced has the same basic user model as the original tcov:

1. Compile a program for a tcov Enhanced experiment.
2. Run the experiment.
3. Analyze results using tcov(1).

The index.assist program can be used to illustrate the operation of tcov Enhanced. To compile a program for code coverage for tcov Enhanced, use the -xprofile=tcov option (for all compilers):

```
% cc -xprofile=tcov -o index.assist index.assist.c
```

tcov Enhanced, unlike tcov, does not produce a .d file. The coverage data file is not created until the program is run. Then one coverage data file is produced as opposed to one file for each module compiled for coverage analysis.

Having compiled index.assist.c, you can run index.assist

```
% index.assist
% ls -dF *.profile
index.assist.profile/
% ls *.profile
tcovd
```

By default, the name of the directory where the tcovd file is stored is derived from the name of the executable. Furthermore, that directory is created in the directory the executable was run in (the original tcov created the .d files in the directory where the modules were compiled).

The directory where the tcovd file is stored is also known as the "profile bucket." The profile bucket can be overridden by using the SUN_PROFDATA environment variable. This may be useful if the name of the executable is not the same as the value in argv[0] (for example, the invocation of the executable was through a symbolic link with a different name).

You can also override the directory where the profile bucket is created. To specify a location different from the run directory, specify the path using the SUN_PROFDATA_DIR environment variable.Absolute or relative pathnames can be specified in this variable. Relative pathnames are relative to the program's current working directory at program completion.

TCOVDIR is supported as a synonym for SUN_PROFDATA_DIR for backward compatibility. Any setting of SUN_PROFDATA_DIR causes TCOVDIR to be ignored. If both SUN_PROFDATA_DIR and TCOVDIR are set, a warning is displayed when the profile bucket is generated. SUN_PROFDATA_DIR takes precedence over TCOVDIR. The variables are used at runtime by a program compiled with -xprofile=tcov, and are used by the tcov command.

Note - This scheme is also used by the profile feedback mechanism.

Now that some coverage data has been produced, you can generate a report that relates the raw data back to the source files:

```
% tcov -x index.profile index.assist.c
% ls *.tcov
index.assist.c.tcov
```

The output of this report is identical to the one from the previous example (for the original tcov).

Creating Profiled Shared Libraries

Creating shared libraries for use with tcov Enhanced is accomplished by using the analogous compiler options:

```
% cc -G -xprofile=tcov -o foo.so.1 doo.o
```

Locking Files

tcov Enhanced uses a simple file-locking mechanism for updating the block coverage data file. It employs a single file created in the same directory as the tcovd file. The file name is tcovd.temp.lock. If execution of the program compiled for coverage analysis is manually terminated, then the lock file must be deleted manually.

The locking scheme does an exponential back-off if there is a contention for the lock. If, after five tries, the tcov runtime cannot acquire the lock, it gives up and the data is lost for that run. In this case, the following message is displayed:

```
tcov_exit: temp file exists, is someone else running this executable?
```

Profile Feedback

Profile Feedback and tcov share a common way of collecting and recording data, which includes placing their output in the same directory and using the same environment variables to control where the profile output goes and what it is called. The profile bucket specifies the directory where the profile output is generated (both tcov profile output and Profile Feedback output).

Compile the program with tcov or Profile Feedback collection turned on, and run the program. At exit, the running program generates a profile bucket. If a previous profile bucket exists, the program uses that profile bucket. If a profile bucket does not exist, it creates the profile bucket.

The default profile bucket the program creates is named after the executable with a ".profile" extension and is created in the place where the executable is run. Therefore, if you are in /home/joe, and run a program called /usr/bin/xyz, the default behavior is to create a profilebucket called xyz.profile in /home/joe.

There are two ways to override the default:

1. Specify the exact profile bucket you want generated on the compile line.

The running executable honors the specification even in the presence of environment variables. This is only done for Profile Feedback.

2. Use the environment variables to change the profile bucket.

There are two ways to change the profile bucket:

a. Change the name of the profile bucket using the environment variable SUN_PROFDATA.

b. Change the directory where the profile-bucket is placed, which can be controlled with `SUN_PROFDATA_DIR`.

The environment variables override the default location and name of the profile bucket. Both can be overridden independently.

For example, if you only choose to set `SUN_PROFDATA_DIR`, the profile bucket will go into the directory where you set `SUN_PROFDATA_DIR`. The default name (which is the executable name followed by a "`.profile`") will still be the name used for the profile bucket.

There are two forms of directories you can specify by using `SUN_PROFDATA_DIR` on the Profile Feedback compile line: absolute pathnames (which start with a `/'), and relative pathnames. If you use an absolute pathname, the profile bucket is dropped into that directory. If you specify a relative pathname, then it is relative to the current working directory where the executable is being run.

For example, if you are in `/home/joe` and run a program called `/usr/bin/xyz` with `SUN_PROFDATA_DIR` set to .. , then the profile bucket is called `/home/joe/../xyz.profile`. The value specified in the environment variable was relative, and therefore, it was relative to `/home/joe`. Also, the default profile bucket name is used, which is named after the executable.

The previous version of `tcov` (enabled by compiling with the `-xa` or `-a` flag) used an environment variable called `TCOVDIR`. `TCOVDIR` specified the directory where the `tcov` counter files go to instead of next to the source files. We have retained compatibility with this environment variable, in that the new `SUN_PROFDATA_DIR` environment variable behaves like the `TCOVDIR` environment variable. If both variables are set, a warning is output and `SUN_PROFDATA_DIR` takes precedence over `TCOVDIR`.

Environment Variables

`SUN_PROFDATA`

Can be used to specify the name of the profile bucket at runtime.The value of this variable is always appended to the value of `SUN_PROFDATA_DIR` if both variables are set.

`SUN_PROFDATA_DIR`

Can be used to specify the name of the directory containing the profile bucket. It is used at runtime and in the `tcov` command.

`TCOVDIR`

`TCOVDIR` is supported as a synonym for `SUN_PROFDATA_DIR` to maintain backward compatibility. Any setting of `SUN_PROFDATA_DIR` causes `TCOVDIR` to be ignored. If both `SUN_PROFDATA_DIR` and `TCOVDIR` are set, a warning is displayed when the profile bucket is generated.

`TCOVDIR` is used at runtime by a program compiled with `-xprofile=tcov` and it is used by the `tcov` command.

Compiler Options

These rules must be followed when using Profile Feedback compiler options:

1. The optimization level should 2 or greater.

2. For `-xprofile=collect` and `-xprofile=use`, you must use the same command-line options.

3. Source code changes are not allowed between the `collect` and `use` Profile Feedback phases.

-xprofile=collect[:nameopt]

This flag instructs the compiler to instrument the code for Profile Feedback data collection.

nameopt specifies a name for the generated profile bucket. It is not a path name to the profile bucket, but the actual name. *nameopt* can name an absolute or relative profile bucket.

If *nameopt* is not specified, the default output profile bucket is placed in the current working directory. It is given the same name as the executable with a `.profile` extension.

If *nameopt* is specified, the profile bucket is called nameopt. If nameopt is a relative pathname to the profile bucket, it is generated relative to the current working directory.

If *nameopt* is specified, it should be a path to the profile bucket. The compiler does not change the user specified name of the profile bucket. If *nameopt* is an absolute name, it is used as specified; otherwise *nameopt* is relative to the current working directory.

If *nameopt* is specified, the environment variables do not affect it. The compiler uses *nameopt* as the name of the profile bucket.

`-xprofile=use[:nameopt]`

This flag instructs the compiler to use Profile Feedback data that was collected from a program instrumented to collect this data.

If *nameopt* is not specified, the default profile bucket name the compiler uses is `a.out.profile` in the current working directory.

If *nameopt* is specified, it should be a path to the profile bucket. The compiler does not change the user specified name of the profile bucket. If *nameopt* is an absolute name, it is used as specified; otherwise *nameopt* is relative to the current working directory.

If *nameopt* is specified, the environment variables do not affect it. The compiler uses *nameopt* as the name of the profile-bucket.

`-xprofile=tcov`

This flag instructs the compiler to instrument the code for `tcov` data collection. It does not create any user-visible files at compile time, unlike the previous version of `tcov`, which created coverage files at compile time.

At the Runtime of an Instrumented Executable

All relative path names start at the current working directory of the program, which includes the environment variables and the directory names specified on the command line.

For `-xprofile=collect` and `-xprofile=tcov`

By default the profile bucket is called <`argv[0]`>`.profile` in the current directory.

If you set `SUN_PROFDATA`, the profile bucket is called `$SUN_PROFDATA`, wherever it is located.

If you set `SUN_PROFDATA_DIR`, the profile bucket is placed in the specified directory.

`SUN_PROFDATA` and `SUN_PROFDATA_DIR` are independent. If both are specified, the profile bucket name is generated by using `SUN_PROFDATA_DIR` to find the profile bucket and, `SUN_PROFDATA` is used to name the profile bucket in that directory.

A UNIX process can change its current working directory during the execution of a program. The current working directory used to generate the profile bucket is the current working directory of the program at exit. In the rare case where a program actually does change its current working directory during execution, the environment variables can easily control where the profile bucket is generated.

For a Program Compiled With `-xprofile=collect:nameopt`

If *nameopt* is specified by the user, then it is used for the name of the profile bucket. None of the environment variables have any effect if *nameopt* is used. This option is used to make sure the Profile Feedback data goes into a single profile bucket no matter what the environment of the program being used looks like. This is useful for groups trying to put an executable in a common area for many people to use and collect data on.

`tcov` *Program*

The `-xprofile-bucket` option specifies the name of the profil -bucket to use for the `tcov` analysis. `SUN_PROFDATA_DIR` or `TCOVDIR` are prepended to this argument, if they are set.

Sample User Scenarios

1. To profile different programs over a period of time, and have a private directory where the profile buckets are created:

- Set `SUN_PROFDATA_DIR` to your private directory and leave it set.
- Make sure this private directory is writable.

2. To collect data from two different sets of runs by using the same binary:

- Set `SUN_PROFDATA` to the name of a *specific* profile bucket that will be created, if necessary.
- Run your program repeatedly with different inputs.
- Set `SUN_PROFDATA` to a different name.
- Run your program repeatedly on a different set of inputs.
- Unset the `SUN_PROFDATA` variable.

3. To profile multiple executables in the same directory:

- Compile your program with only `-xprofile=collect`. By default, the profile data for each individual program goes into a separate profile bucket, each named after a different executable.

4. To make sure the output of your profile run always goes to a certain directory without using environment variables:

- Compile your program with `-xprofile=collect:`<*my profile-bucket*>.
- Run your program. Tthe output goes to the specified profile bucket.
- Recompile your program with `-xprofile=use:`<*my profile-bucket*> to use this data.

For example:

a. Compile your program with `-xprofile=collect`.

b. Specify where you would like the output to go.

```
% cc -xprofile=collect:/home/bar/run1.profile -o xyz xyz.c -x04
```

c. Run your program.

The profile bucket `/home/bar/run1.profile` is created.

```
% xyz
```

d. Recompile your program to use the profile information.

```
% cc -xprofile=use:/home/bar/run1.profile -o xyz xyz.c -x04
```

5. To examine the new version of tcov after you have TCOVDIR set in your environment:

- Compile your program (called `xyz`, for example) with `-xprofile=tcov`.
- Run xyz, the output goes to `$TCOVDIR/xyz.profile`.

- Run the tcov command:

```
tcov -x xyz.profile filename.c,
```

The output from tcov will be in filename.c.tcov.