PROFILING SUPPORT ON HPUX 11i VERSION 1.5

MOHIT SINHA
HEWLETT-PACKARD, ESDI,
29,CUNNIGHAM ROAD,
BANGALORE-560052
INDIA
(91)(80) 2251554 Ext 1047
mohits@india.hp.com
mohit_sinha@hp.com
Fax (91)(80) 220-0196

Abstract

Performance is a key differentiator for high-end applications. Profiling helps developers to analyze and improve the performance of their applications on UNIX platforms. Gprof style profiling generates the timing information and call graph for the user application. Single Shared Library Profiling (SSLP) Support has been introduced starting HP-UX 11i,which has been extended to Multiple Shared Library Profiling (MSLP) in subsequent versions. Profiling requires gprof tool and gprof library. To profile an application, the user needs to compile his sources using appropriate compiler options. These options will link gprof library implicitly with user application, resulting in runtime invocation of profiling routines. At termination of the application all the profiling information is dumped into a file named gmon.out. The gprof tool then can be used to generate the timing and call graph information using gmon.out. This paper explains how profiling works and how you can use it to measure and improve the performance of your applications/libraries. Especially it will be interesting to learn how to interpret call graphs and timing information for inter/intra-library calls. At the end of the paper, references of sample examples are given with a step-by-step explanation.

The MSLP support involved development of a new library libgprof and modifications in the existing gprof command to interpret the information dumped in the profile file, gmon.out.The format of gmon.out has to be changed in order to account for profiling of Multiple load modules.

The flow of the paper will be as follows:

- Introduction to profiling (what is profiling, what is profil syscall, _mcount routine, monitor routine and various techniques of profiling).
- Why Shared library Profiling.
- Tricks/Tips of gprof.
- Sample examples, with a step-by-step explanation, to educate users on how to interpret gprof output.
- Conclusion.
- Acknowledgements.

1. Introduction to profiling:

Profiling means providing the user with statistics of time taken by various routines along with the percentage of CPU time. The user can then use these statistics to either fine tune their applications or change them in an optimized manner. The HP-UX operating system provides two instrumentation techniques for profiling the execution of programs: program counter sampling, and procedure call counting. Program counter (PC) sampling implemented within the OS at the clock interrupt level, provides statistical information about how much time is spent at different points in the program, enabling a programmer to determine what routines are responsible for most of the execution time. Procedure call counting, implemented by code added to each procedure at compile time, provides an exact count of the number of times each procedure was called.

These two techniques are usually employed together in conjunction with the prof and gprof tools. The -p (prof) & -G (gprof) compiler options have two effects:

- When used at compile time, they cause the compiler to generate code in each procedure that counts the call;
- When used at link time, the compiler passes the options to the linker, which arranges for appropriate program startup code to enable PC sampling and to write the profiling results to a data file when the program terminates.

An instrumented program accumulates the profiling information in a buffer while it is executing, and then writes the data to a file when it terminates. The prof or gprof tool can then correlate this data to the symbol table of the appropriate load module and produce a profile report. These two tools differ in the amount of data collected and the details provided in the report; gprof collects call-graph information, and allocates time spent in each procedure proportionally to the callers of that procedure ("charge back"), in addition to giving flat profiling information about routines like prof.

Profiling involves the following components from runtime architecture:

- **The compilers**. With the -p and -G options, compilers instrument the compiled code by inserting calls to the _mcount routine.
- **The startup files** crt0.o, mcrt0.o, and gcrt0.o.Depending on link-time options an application is linked with one of these startup scripts. If neither -p nor -G is specified then application is linked with crt0.o, which contains a null _mcount routine (a dummy reference). The _mcount routine will be discussed in detail later in the paper. If -p is specified, the application is linked with mcrt0.o, which contains code to start profiling, a real version of _mcount and a code to write the accumulated profiling data to a file when program terminates. If -G is specified application links with gcrt0.o which contains similar code but accumulates more information to be used by gprof. These startup scripts are present only for PA32 runtime. For PA64 and IA64 runtime architectures, the profiling related codes have been moved from these startup files to equivalent libraries libprof and libgprof.
- **The C library**. It contains a prof compatible version of monitor API, which is a front end for profil syscall. For gprof, the compatible version of monitor is present in gcrt0.o.This is there only for PA32.For PA64 and IA64 the standard versions of monitor has been moved from libc or gcrt0.o to libprof and libgprof respectively.

- **Prof and gprof tools**. These analysis tools read the data files produced by an instrumented application, and generate the profile reports.

1.1. Program counter sampling: the profil/sprofil system call

The **profil/sprofil** syscall are used to request the OS to start sampling or to stop sampling. When sampling is on, the OS samples the program counter (PC) at certain intervals, maps the program counter to a "bucket" in the sample buffer, and increments the counter in that bucket. The parameters to profil control the location and size of the sample buffer, which is in user space, the starting address of the code region being profiled, and a scale factor, ranging from 0.0 to 1.0,that determines the mapping of PC values to the buckets. Larger scale factors provide greater granularity in the sampling data.

The **profil** system call has the following prototype:

```
void profil (
              unsigned short int *buff,
              size_t bufsiz,
              void *offset,
              unsigned int scale
             );
```

The first argument, buff, is a pointer to the base of the sample buffer, and the second argument, bufsiz, is the size of this buffer in bytes. The third argument, offset, is a pointer to the base of the text region to be profiled, and the fourth argument scale, specifies the scale factor to be used in mapping locations in the text segment to buckets, which effectively defines the size of the text region.

The sample buffer is an array of 16-bit buckets, to which instructions in the text region are mapped according to the scale factor. The scale factor is a fixed-point fraction between 0.0 and 1.0,with an implied radix point 16 bits from the right. If scale is 0 or 1, sampling is turned off. The third argument, offset, is expected to be the actual address of the beginning of the region of the code to be profiled.

The current profil system call dates back to the first implementation of UNIX, and suffers from this in two respects. First, it allocates the sample buffer as an array of 16-bit counters, as a result of which the bucket will overflow if it receives more than 65,535 hits (almost 11 minutes of CPU time at 100 Hz). The OS provides no overflow indication. Second, the interface allows for only a single text region to be profiled. This has been a serious limitation for doing performance analysis on shared libraries. So a new syscall **sprofil** has been implemented in order to allow profiling of Multiple load modules (text segments). The sprofil system call has the following prototype:

```
int sprofil (struct prof *profp,
             int profcnt,
             struct timeval *tvp,
             unsigned int flags
            );
```

The first argument, profp, is a pointer to an array of prof structures; the second argument, profcnt, is the number of elements in the array. Each element of the array specifies one text region that should be profiled. When called with a non-zero value of profcnt, profiling is enabled for each text region specified in the array; if profiling was already enabled for some regions, it is disabled for those regions before being enabled for the new set of regions. When called with a zero value of profcnt, profiling is disabled. If non-NULL, tvp

points to a struct timeval, which on return will contain the time value corresponding to one clock tick. The flags argument can be used to choose 16-bit or 32-bit buckets. If set to PROF_USHORT, profiling will treat the sample buffer as an array of 16-bit buckets; if set to PROF_UINT, profiling will treat it as an array of 32-bit buckets. (There should be no need for 64-bit buckets, since 32-bits allow counts to reach many CPU-years.) The **prof** structure has four fields, corresponding to the four arguments to the original profil system call, and is defined as follows:

```
struct prof (void *pr_base; /* buffer base */
             unsigned int pr_size; /* buffer size */
             unsigned int pr_off; /* pc offset */
             unsigned int pr_scale; /* pc scaling */
            );
```

The field pr_base is a pointer to the base of the sample buffer for this region, and pr_size is the size of this buffer, in bytes. The field pr_off is a pointer to the base of the text region to be profiled, and pr_scale specifies the scale factor to be used in mapping locations in the text segment to buckets, which effectively defines the size of the text region.

The extension of Multiple Shared Library Profiling (MSLP) on HPUX 11i version 1.5 was mainly possible as a result of development of this new syscall. The details of MSLP are covered later in the paper.

1.2. Procedure call counting: the _mcount routine

When compiling code with the -p or -G option, the compiler instruments each procedure by inserting a call to _mcount at the beginning of each procedure (following the procedure prologue). This procedure has the following interface:

```
void _mcount (unsigned long rp,
              unsigned long pc,
              unsigned int **counter_ptr);
```

The first argument, rp, is a copy of the return pointer that is, address of the call site. This gives gprof the call graph information it needs for charge back.

The second argument, pc, is a pointer to an arbitrary instruction in the instrumented procedure itself (rp and pc parameters of _mcount allow it to be implemented in high level language instead of assembly.).

The third argument, counter_ptr, is a pointer to a statically allocated double word in the .sbss section. The compiler must allocate one of these double words for each instrumented procedure. When _mcount is called from a procedure for the first time, it will find *counter_ptr initialized to zero. Counters should be 32 bit unsigned integers.

There will be three versions of the _mcount routine:

- The C library will contain an empty version, to satisfy references from code instrumented at compile time, but linked into a non-instrumented program.
- The prof library will have a version that accumulates basic call counting information. For this version, the rp argument is not used, since prof does not do charge back. When compiling with -p instead of -G, the compiler may choose to set this parameter to zero.
- The gprof library will have a version that accumulates full call counting information. Current gprof implementation of _mcount does not use this third parameter.

1.3. The monitor library routine

The **monitor** routine is a higher-level interface to profil syscall. It sets up program counter sampling, allocates data structures for both sampling and call counting, turns off profiling, and writes the profiling data to a disk file. It has the following interface:

```
void monitor (void (*lowpc)(),
              void (*highpc)(),
              long *buffer,
              int bufsize,
              int nfunc);
```

The first two arguments define the range of addresses that will be profiled. These two arguments must be valid function pointers. For convenience in profiling the whole text segment in a statically bound program, the linker defines two symbols, __text_start_f and __etext_f, which are the beginning and ending addresses, respectively, of the text segment. These symbols are declared in the header file <crt0.h>, and may be used as parameters to monitor. The third and fourth arguments define the starting address and length of a buffer for the sample buckets, and the final argument is the total number of procedure call counters that should be allocated.

The monitor routine will automatically calculate the proper scale factor and call the profil system call to enable sampling. It will also allocate an array of procedure call counters and the data structures necessary for _mcount to allocate counters from this array dynamically.

There will be two versions of the monitor routine:

- The prof library will have a version that allocates data structures for basic call counting information and dumps the data in a prof-compatible format.
- The gprof library will have a version that allocates data structures for full call counting information and dumps the data in a gprof-compatible format. New libgprof does not have this routine. Instead it defines a new interface smonitor, compatible with sprofil. This is discussed later in the paper.

1.4. Linking for use with prof and gprof

When linking a program for use with prof, the compiler will pass the "-lprof" (lower-case L) option to the linker. The -l option causes the linker to load the prof library before the C library, where it will find prof versions of the monitor and _mcount routines. When linking a program for use with gprof, the compiler will pass the -lgprof option to the linker. The -l option causes the linker to load the gprof library before the C library, where it will find gprof versions of the monitor and _mcount routines.

The prof and gprof libraries will be delivered in shared forms only starting 11.20. The shared forms are normal DLLs (Dynamically loadable libraries), and will be loaded automatically when the program is executed. When linking a program for either profiler tool, the compiler will also pass the -L /usr/ccs/lib/hpux32/libp option to the linker. This option adds an extra directory to the library search path, where instrumented versions of selected system libraries may be found. These libraries will have been compiled with the -G option, so that they are instrumented for procedure call counting.

Note that linker does not support the -p or -G options for profiling. These options are meaningful only to compiler drivers, which in turn pass the appropriate options to the linker. To invoke the linker directly, the linker options mentioned above must be used instead. Note also that no separate crt0.o files are required.

1.5. Running the instrumented program

When an instrumented program starts execution, the initializer in the .init section of the profiling library (libprof or libgprof) is called before control reaches main. The initializer will call monitor to start profiling on the program's text segment; allocate the data structures used for sampling and call counting; and arrange, via atexit, to call monitor again to stop profiling at the end of the program.

As the program is running, the OS in the buffer allocated for it will accumulate program counter sampling data. Any code compiled with instrumentation will call the _mcount routine each time it is called to accumulate the call counting data.

When an instrumented program terminates, one of its atexit actions will be to call monitor to disable profiling and to write the profiling data to a disk file.

2. Why Shared library profiling

Shared library offers many advantages as compared to archive libraries. As there was continuous stress on discontinuing shipping of archive versions of system libraries to keep HPUX alive in this competitive world and improve performance, the need to support profiling of shared libraries was also felt. If HPUX continued profiling of applications alone call-graph and timing information will be broken and users would not have the liberty to identify potential bottlenecks in their application. HPUX 11i saw the introduction of Single Shared library Profiling (SSLP) and after that Multiple shared library profiling was introduced for HPUX 11i version 1.5.As HPUX 11i version 1.5 is not shipping archive versions of system libraries so shared library profiling becomes all the more important and offers user tremendous flexibility to fine tune his application completely.

2.1 History of profiling support on HPUX

The table below shows the history of profiling on HPUX:

	Application profiling	**SSLP**	**MSLP**	**C**	**C++**	**FORTRAN**	**PASCAL**
11.0	✓			✓	✓	✓	✓
11.i	✓	✓ (For 32 bits only)		✓	✓	✓	
11.i version 1.5	✓	✓	✓	✓	✓	✓ (Only F90)	Compiler not supported

2.2 Technical Overview of Shared Library profiling Support

HP-UX supported profiling of fully archive bound applications up till 11.11.Starting 11.11, the support for profiling of single shared library was introduced for 32 bits. The rest of the paper will talk about gprof style of profiling only. All the support of profiling Shared libraries has been introduced only for gprof style profiling. Prof based profiling exists still for application profiling alone. Users should not get confused on this aspect.

For charge back, SSLP used runtime instrumentation of procedures with _callcount. The basic design involved patching of PLT's & dynamically generated OPD's. Whenever the routines of shared library, that was to be profiled, were called then instead of the routine, "_gprof_stub" should be called. The reason was "_gprof_stub" would first call the "_call_count" routines, which would do the instrumentation, after that control would come back to "_gprof_stub", now it would call the original routine. So this was the functionality of "_gprof_stub", which was written in IA assembly. Now we had to feed the address of "_gprof_stub" in both PLT and OPD entries because if we call a function normally, call would be routed through PLT entries and if we call the function using function pointers it would be routed through OPDs.

The catch was we couldn’t simply replace the address of routine that has to be profiled with the address of "_gprof_stub". Since "_gprof_stub" needs the address of routines that had to be profiled (for later restoration & subsequent branching to that address) and address would be available only at run time, we had to put some placeholders (0x00000000) for the following things:

- Address of routines that had to be called,
- Global data pointer (gp) of library that had to be profiled (which is known as "libltptr")
- Address of "_call_count"
- Gp value of libgprof (this is required for "_call_count" & is popularly called as prof_ltptr).

Now at run time we would be having these addresses. So we had to patch "_gprof_stub" with these addresses. To patch it we would count how many instruction bundles "gprof_stub" would have. Now we would figure out which particular bits in a particular instruction bundle would have our placeholders. After getting this we will put the run time addresses in those bits. There were specific macros written for this purpose to feed the runtime bits at exact position in the correct instruction bundle. Explicit bundling was used to simplify the things a little. Now of course shared library that has to profile may have more than one routine. So if we want to profile more than one routine, "_gprof_stub" will have to be different for all the routines because the address of the routine that has to be profiled will be unique, popularly called in our implementation as "to_pc". So for every routine in the library we were profiling, we need a local copy of "_gprof_stub". In this local copy of gprof_stub we feed in the runtime values of "to_pc", "call_count_addr", "prof_ltptr" & "lib_ltptr". Finally we replace the address of routine with address of stub .So now whenever the routine, of the library we are profiling, is called control used to go to "gprofstub", then "call count routine” back to gprof stub where actual routine address was restored & finally to routine & then back.

SSLP suffered from following drawbacks:

- Since profil syscall was used, it could profile only one text segment at a time.

- It was not possible to patch BOR events. So the library had to be built with -Wl, -B immediate option.
- It uses runtime instrumentation, which was a big overhead.

To overcome these drawbacks, profiling of Multiple shared libraries was taken up. MSLP required a lot of changes in profile file (gmon.out) format, gprof command, libgprof, linker & compiler. The remaining sections discuss these changes one by one.

2.2.1. Changed profile file format

The gmon.out file format has to be changed completely in order to account for Multiple Shared Library Profiling. The new file is much more structured as compared to the old profile file, as it has to contain information about multiple load modules now. It contains various headers having information like how many load modules are getting profiled, what are the profil and call count dump sizes for each load modules and where exactly are these dumps starting from in the profile file. The new file will have an overall structure like this:

Gmon.out

<File Header>
<Loadable Module Key table>
<Array of Loadable module headers>(one for each load modules)
<Data Dump> one for each loadable module.

Explanation:

<File Header>

This is the first structure that will be stored at the beginning of the file. This will have information about how many loadable modules are there, whether it's the new format or old format gmon.out file, version strings etc.

<Loadable module Key table>

```
struct ldm_key {int ldm_key;/* Unique key for each load module */

    int ldm_pathsize;/* Contains the size of the path name */

    char *ldm_name;/* Contains the path name of the load module */
};
```

<Loadable module header entries>

Every time any loadable module is loaded and then unloaded the call count information and the profil information maintained for that module has to be dumped to the gmon.out file. This header is used to give details about the buffer and its location in the file and to which file it belongs to. It also stores in the profile file the offset from where the profil and call count dump will start for that load module, their respective sizes etc.

<Data portion>

The data portion will be a byte stream. The byte stream can be looked as:
<The call count dump for load module 1><The profil dump for load module 1><The call count dump for load module 2><The profil dump for load module 2> ...
<The call count dump for load module n><The profil dump for load module n>

In order to account for this new format both libgprof & gprof command has to be changed. Libgprof should dump information at application termination into the profile file gmon.out in the specified format, which gprof command should be able to parse and correlate with the symbol table of the appropriate load module.

2.2.2. Development of libgprof:

A new library, libgprof, has to be developed to account for Multiple Shared library profiling. The new libgprof uses sprofil syscall to account for profiling of Multiple load modules. With this "gprof" library user always has the options to profile as many loadable modules as desired. Loadable module might be "a.out" or shared library. Load modules to be profiled should be specified in "LD_PROFILE" environment variable. "LD_PROFILE" is a colon-separated list of the load modules names, not the path names. It should be noted that if dld is loading "libc.so" and user wants to profile "libc.so", name of shared library specified in "LD_PROFILE" should be "libc.so" only. It should not be "libc.so.1" etc. The names of the load modules can be found out either by running ldd or chatr on the executable.

The basic design of new libgprof requires registering certain user routines with DLD events. DLD events, which we are interested in, are:

- **POST_INIT** The routine registered with this event will do all the initialization.
- **LOAD_COMPLETE** The routine registered with this event will call "sprofil". All the parameters to be passed to "sprofil" are already in place.
- **TERMINATE_START** This is a new event registered with dld specifically for the purpose of Multiple Shared library profiling. This event marks the beginning of the process when loader starts unloading the libraries. The routine registered with this event will dump all the information collected, both profiling and call count, up till now in the profile file (gmon.out) in the format specified above.
- **UNLOAD_POST_FINI** The routine registered with this event is called whenever any of the libraries is unloaded. The routine checks whether the actual termination of application has started or it is just some module getting unloaded (due to shl_unload etc). In the latter case the routine will have to stop "sprofil" syscall, condense the profp array & then start the "sprofil" syscall again.

The new libgprof also provides user with a new API, **smonitor**, a higher-level interface to sprofil. The smonitor is an extension of monitor for Multiple text segments, as sprofil is of profil syscall. Using smonitor, user can allocate all the profiling buffers himself, start sprofil & get all the profiling information in his specified buffers. In effect using smonitor user can take up

profiling control under his charge as against libgprof in normal program. User can then use gprof tool to interpret & get a listing of profile information.

smonitor has the following interface:

```
void smonitor (struct text_region *regions,
               int nregions,
               void *buffer,
               size_t bufsize,
               int nfunc,
               unsigned flags
               );
```

The first argument regions is an array of structures of type text_regions defined in <mon.h> header file. The structure has the following fields:

```
struct text_region (unsigned long text_start;
                    unsigned long text_end;
                    char *name;
                    );
```

nregions is the number of elements in the array regions.

buffer is the starting address of a buffer to collect sampling information.

bufsize is the length of buffer. buffer is the only memory region used by smonitor () to collect profiling information, so it should be big enough for all specified regions. smonitor () does not initialize buffer. With more than one call to smonitor () in the same process, smonitor () dumps the sampling information collected with the last call. Smonitor () does not discard the information collected in previous calls if it is present in the buffer passed to last call of smonitor ().

nfunc is unused and kept for future usage.

flags are used to choose 16-bit or 32-bit buckets to collect sampling information. If flags is set to PROF_USHORT, smonitor () treats the buffer as an array of 16-bit buckets; if set to PROF_UINT, smonitor () treats the buffer as an array of 32-bit buckets.

smonitor is largely influenced by environment variable LD_PROFILE settings, which is discussed later in the paper.

2.2.3. Changes done to gprof command:

Gprof command also required substantial modifications in order to account for Multiple shared library profiling. The old gprof command was capable of parsing a single file & reading & correlating that with the symbol table of either the application or the single library. Gprof command was now modified to read the symbol tables of all load modules, populate a global structure module_info for each loadable modules, make nl structures for the functions of the load modules that are getting profiled & finally correlate the profiling & call count information with symbol table of the appropriate load module. Finally it produces a listing of all the function with the complete call graph. In the flat profile report a new column on module index was added specifying that the said function is present in which of the loadable module. In the end a list of all loadable modules & their corresponding module indices is reported. In the

beginning of the report itself, a listing is given as to which load modules are not getting profiled.

Gprof has a limitation. If symbol table of any of the load modules is chopped (i.e. compiled with **-x linker** option) then gprof won't be able to find the parent of this call & will report this as a warning.

2.2.4. Changes done in linker:

The following changes were done in linker:

- A new event was defined in the loader to mark the beginning of the termination of the application. This event was required because with this we can start dumping all the profiling information into the profile file.

- A new option +profilebucketsize [16|32] was introduced to specify the counter size as required by sprofil syscall, from the link line. User can specify the values of 16 or 32 here. Default value is 16.This value will be overridden if any valid value is specified at runtime through environment variable LD_PROFILEBUCKET_SIZE. This variable will be discussed in detail later in the paper.

2.2.5. Changes done in compiler:

The compiler saw the introduction of one new command line option. A new option +profilebucketsize= [16|32] was introduced. This option will be passed to the linker, which will set the appropriate counter size. The valid values for this option are 16 & 32, with 16 as default. The value can be overridden with runtime selection.

2.2.6. External Influences:

MSLP is largely influenced by the following environment variables:
LD_PROFILE determines the modules to be profiled as follows.

LD_PROFILE=ALL

Profile all load modules. That is, report timing and call count information for all loadable modules, including a.out.

LD_PROFILE=ldm1: ldm2

Profile only loadable modules ldm1 and ldm2.ldm1 and ldm2 are not full pathnames; they are the names recorded in the executables, which can be displayed using chatr (1).

If **LD_PROFILE is not set**, gprof behaves as though LD_PROFILE=ALL.
LD_PROFILEBUCKET_SIZE controls the size of profiling counters. The acceptable value for this variable is 16 or 32. Counter size can also be specified at compile time using the +profilebucketsize option. The runtime value overrides the compile time value. A warning is issued if the counter size is set to a value other than 16 or 32; in this case the value specified at compile time is used. The default value of the counter is 16, which is used if a valid value is not specified. See the description of the cc (1) +profilebucketsize option for more details.

In addition the behavior of following environment variable has been retained as it is:
GPROFDIR controls the name of the file created by a profiled program. If GPROFDIR is not set, gmon.out is produced in the current directory when the program

terminates. If GPROFDIR=string, string/pid.progname is produced, where progname is argv [0] with any path prefix removed, and pid is the program's process ID. If GPROFDIR is set to a null string, no profiling output is produced.

3. Tricks/Tips OF GPROF:

3.1. Reducing Statistical Sampling Error

The run-time figures that gprof gives you are based on a sampling process, so they are subject to statistical inaccuracy. If a function runs only a small amount of time, so that on the average the sampling process ought to catch that function in the act only once, there is a pretty good chance that user will actually find that function zero times, or twice in the report. By contrast, the number-of-calls and basic-block figures are derived by counting, not sampling. They are completely accurate and will not vary from run to run if your program is deterministic.

The sampling period that is printed at the beginning of the flat profile says how often samples are taken. The rule of thumb is that a run-time figure is accurate if it is considerably bigger than the sampling period. The actual amount of error can be predicted. For n samples, the expected error is the square root of n. For example, if the sampling period is 0.01 seconds and foo's run-time is 1 second, n is 100 samples (1 second/0.01 seconds), sqrt (n) is 10 samples, so the expected error in foo's run-time is 0.1 seconds (10*0.01 seconds), or ten percent of the observed value. Again, if the sampling period is 0.01 seconds and bar's run-time is 100 seconds, n is 10000 samples, sqrt (n) is 100 samples, so the expected error in bar's run-time is 1 second, or one percent of the observed value. It is likely to vary this much on the average from one profiling run to the next. (Sometimes it will vary more). This does not mean that a small run-time figure is devoid of information. If the program's total run-time is large, a small run-time for one function does tell you that that function used an insignificant fraction of the whole program's time. Usually this means it is not worth optimizing.

One way to get more accuracy is to give your program more (but similar) input data so it will take longer. Another way is to combine the data from several runs, using the `-s' option of gprof. Here is how:

1. Run your program once.
2. Issue the command `mv gmon.out gmon.sum'.
3. Run your program again, the same as step 1.
4. Merge the new data in `gmon.out' into `gmon.sum' with this command:
   gprof -s executable-file gmon.out gmon.sum
5. Repeat steps 3,4 as often as you wish.
6. Analyze the cumulative data using this command:
   gprof executable-file gmon.sum > output-file

3.2. Estimating children Times

Some of the figures in the call graph are estimates--for example, the children time values and all the time figures in caller and subroutine lines. There is no direct information about these measurements in the profile data itself. Instead, gprof estimates them by making an assumption about your program that might or might not be true.

The assumption made is that the average time spent in each call to any function foo is not correlated with who called foo. If foo used 5 seconds in all,

and 2/5 of the calls to foo came from a, then foo contributes 2 seconds to a's children time, by assumption. This assumption is usually true enough, but for some programs it is far from true. Suppose that foo returns very quickly when its argument is zero; suppose that a always passes zero as an argument, while other callers of foo pass other arguments. In this program, all the time spent in foo is in the calls from callers other than a. But gprof has no way of knowing this; it will blindly and incorrectly charge 2 seconds of time in foo to the children of a.

We hope some day to put more complete data into `gmon.out', so that this assumption is no longer needed, if we can figure out how. For the nonce, the estimated figures are usually more useful than misleading.

4. EXAMPLES AND INTERPRETATION OF GPROF OUTPUT:

Gprof reports two types of listings, the flat profile followed by call graph listing. The flat profile shows the total amount of time your program spent executing each function. Unless the `-z' option is given, functions with no apparent time spent in them, and no apparent calls to them, are not mentioned. Note that if a function was not compiled for profiling, and didn't run long enough to show up on the program counter histogram, it will be indistinguishable from a function that was never called.

This is part of a flat profile for a small program:

4.1. Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	Module Index	name
33.34	0.02	0.02	7208	0.00	0.00	2	open
16.67	0.03	0.01	244	0.04	0.12	2	offtime
16.67	0.04	0.01	8	1.25	1.25	2	memccpy
16.67	0.05	0.01	7	1.43	1.43	2	write
16.67	0.06	0.01					mcount
0.00	0.06	0.00	236	0.00	0.00	2	tzset
0.00	0.06	0.00	192	0.00	0.00	2	tolower
0.00	0.06	0.00	47	0.00	0.00	2	strlen
0.00	0.06	0.00	45	0.00	0.00	2	strchr
0.00	0.06	0.00	1	0.00	50.00	0	main
0.00	0.06	0.00	1	0.00	0.00	2	memcpy
0.00	0.06	0.00	1	0.00	10.11	0	print
0.00	0.06	0.00	1	0.00	0.00	1	profil
0.00	0.06	0.00	1	0.00	50.00	1	report

...

The functions are sorted by decreasing run-time spent in them. The functions `mcount' and `profil' are part of the profiling apparatus and appear in every flat profile; their time gives a measure of the amount of overhead due to profiling.

The sampling period estimates the margin of error in each of the time figures. A time figure that is not much larger than this is not reliable. In this example, the `self seconds' field for `mcount' might well be `0' or `0.04' in another run.

Here is what the fields in each line mean:

% time

This is the percentage of the total execution time your program spent in this function. These should all add up to 100%.

cumulative seconds
This is the cumulative total number of seconds the computer spent executing this functions, plus the time spent in all the functions above this one in this table.

self seconds
This is the number of seconds accounted for by this function alone. The flat profile listing is sorted first by this number.

calls
This is the total number of times the function was called. If the function was never called, or the number of times it was called cannot be determined (probably because the function was not compiled with profiling enabled), the calls field is blank.

self ms/call
This represents the average number of milliseconds spent in this function per call, if this function is profiled. Otherwise, this field is blank for this function.

total ms/call
This represents the average number of milliseconds spent in this function and its descendants per call, if this function is profiled. Otherwise, this field is blank for this function.

module index
This gives the corresponding number of the load module where the particular function resides.

name
This is the name of the function. This field sorts the flat profile alphabetically after the self seconds field is sorted.

4.2. Call-graph

The second part of gprof output contains a call graph listing. The call graph shows how much time was spent in each function and its children. From this information, you can find functions that, while they themselves may not have used much time, called other functions that did use unusual amounts of time.
Here is a sample call graph from a small program. This call came from the same gprof run as the flat profile example.
granularity: each sample hit covers 2 byte(s) for 20.00% of 0.05 seconds

```
index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.00    0.05                 start [1]
                0.00    0.05       1/1           main [2]
                0.00    0.00       1/2           on_exit [28]
                0.00    0.00       1/1           exit [59]
-----------------------------------------------
                0.00    0.05       1/1           start [1]
[2]    100.0    0.00    0.05       1         main [2]
                0.00    0.05       1/1           report [3]
```

```
-----------------------------------------------
                0.00    0.05       1/1           main [2]
[3]    100.0    0.00    0.05       1         report [3]
                0.00    0.03       8/8           timelocal [6]
                0.00    0.01       1/1           print [9]
                0.00    0.01       9/9           fgets [12]
                0.00    0.00      12/34          strncmp <cycle 1> [40]
                0.00    0.00       8/8           lookup [20]
                0.00    0.00       1/1           fopen [21]
                0.00    0.00       8/8           chewtime [24]
                0.00    0.00       8/16          skipspace [44]
-----------------------------------------------
[4]     59.8    0.01        0.02       8+472     <cycle 2 as a whole>   [4]
                0.01        0.02     244+260         offtime <cycle 2> [7]
                0.00        0.00     236+1           tzset <cycle 2> [26]
-----------------------------------------------
```

The lines full of dashes divide this table into entries, one for each function. Each entry has one or more lines. In each entry, the primary line is the one that starts with an index number in square brackets. The end of this line says which function the entry is for. The preceding lines in the entry describe the callers of this function and the following lines describe its subroutines (also called children when we speak of the call graph). The entries are sorted by time spent in the function and its subroutines. The internal profiling function _mcount is never mentioned in the call graph.

4.2.1. The Primary Line

The primary line in a call graph entry is the line that describes the function, which the entry is about and gives the overall statistics for this function. For reference, we repeat the primary line from the entry for function report in our main example, together with the heading line that shows the names of the fields:

```
index  % time    self  children called     name
...
[3]    100.0    0.00    0.05       1         report [3]
```

Here is what the fields in the primary line mean:

index

Entries are numbered with consecutive integers. Each function therefore has an index number, which appears at the beginning of its primary line.

Each cross-reference to a function, as a caller or subroutine of another, gives its index number as well as its name. The index number guides you if you wish to look for the entry for that function.

% time

This is the percentage of the total time that was spent in this function, including time spent in subroutines called from this function.

The time spent in this function is counted again for the callers of this function. Therefore, adding up these percentages is meaningless.

self

This is the total amount of time spent in this function. This should be identical to the number printed in the seconds field for this function in the flat profile.

children
This is the total amount of time spent in the subroutine calls made by this function. This should be equal to the sum of all the self and children entries of the children listed directly below this function.

called
This is the number of times the function was called.
If the function called itself recursively, there are two numbers, separated by a `+'. The first number counts non-recursive calls, and the second counts recursive calls.
In the example above, the function report was called once from main.

name
This is the name of the current function. The index number is repeated after it.
If the function is part of a cycle of recursion, the cycle number is printed between the function's name and the index number. For example, if function gnurr is part of cycle number one, and has index number twelve, its primary line would be end like this:
gnurr <cycle 1> [12]

4.2.2. Lines for a Function's Callers

A function's entry has a line for each function it was called by. These lines fields correspond to the fields of the primary line, but their meanings are different because of the difference in context.
For reference, we repeat two lines from the entry for the function report, the primary line and one caller-line preceding it, together with the heading line that shows the names of the fields:

```
index  % time    self  children called     name
...
                0.00    0.05       1/1           main [2]
[3]    100.0    0.00    0.05       1         report [3]
```

Here are the meanings of the fields in the caller-line for report called from main:

self
An estimate of the amount of time spent in report itself when it was called from main.

children
An estimate of the amount of time spent in subroutines of report when report was called from main.
The sum of the self and children fields is an estimate of the amount of time spent within calls to report from main.

called
Two numbers separated by a slash (/): the number of times report was called from main, followed by slash and the total number of non-recursive calls to report from all its callers.

name and index number
The name of the caller of report to which this line applies, followed by the caller's index number.
Not all functions have entries in the call graph; some options to gprof request the omission of certain functions. When a caller has no entry of its own, it still has caller-lines in the entries of the functions it calls.
If the caller is part of a recursion cycle, the cycle number is printed between the name and the index number.
If the identity of the callers of a function cannot be determined, a dummy caller-line is printed which has `<spontaneous>' as the "caller's name" and all other fields blank. This can happen for signal handlers.

4.2.3. Lines for a Function's Subroutines

A function's entry has a line for each of its subroutines--in other words, a line for each other function that it called. These lines' fields correspond to the fields of the primary line, but their meanings are different because of the difference in context.
For reference, we repeat two lines from the entry for the function main, the primary line and a line for a subroutine, together with the heading line that shows the names of the fields:

```
index  % time    self  children called     name
...
[2]    100.0    0.00    0.05       1         main [2]
                0.00    0.05       1/1           report [3]
```

Here are the meanings of the fields in the subroutine-line for main calling report:

self
An estimate of the amount of time spent directly within report when report was called from main.

children
An estimate of the amount of time spent in subroutines of report when report was called from main.
The sum of the self and children fields is an estimate of the total time spent in calls to report from main.

called
Two numbers, the number of calls to report from main followed by the total number of nonrecursive calls to report.

name
The name of the subroutine of main, to which this line applies, followed by the subroutine's index number.
If the caller is part of a recursion cycle, the cycle number is printed between the name and the index number.

4.3. EXAMPLES

4.3.1. Profiling of application alone

```
 $ cat a.c
       void func ()
             {
                   printf ("I am in func\n");
             }

       void main ()
             {
                   printf ("I am in main\n");
                   func ();
             }
$ cc -G a.c
$ ldd a.out
 libsin.so.1 => /usr/lib/hpux32/libsin.so.1
 libgprof.so => /usr/lib/hpux32/libgprof.so
 libc.so.1 => /usr/lib/hpux32/libp/libc.so.1
 libelf.so.1 => /usr/lib/hpux32/libelf.so.1
 libdl.so.1 => /usr/lib/hpux32/libdl.so.1
 libdl.so.1 => /usr/lib/hpux32/libdl.so.1
$ export LD_PROFILE=a.out
$ a.out
I am in main
I am in func
$ unset LD_PROFILE
$ ll a.out gmon.out a.c
-rw-rw-rw-   1 vts        ssgrp           99 May 24 12:27 a.c
-rwxrwxrwx   1 vts        ssgrp        16272 May 24 12:27 a.out
-rw-r--r--   1 vts        ssgrp       356278 May 24 12:44 gmon.out
$ gprof >aa
```

4.3.2. Profiling of application and a shared library

```
$ cat  test.c
           void a()
           {
                  printf("I am in a\n");
           }
$ cc -c +Z -G test.c
$ ld -b -o libtest.so.1 test.o
$ ln -s ./libtest.so.1 libtest.so
$ cat main.c
           extern void a();
           main()
           {
                  printf("Hello world\n");
                  a();
           }
$ cc -G main.c  -L. -ltest
$ export LD_PROFILE=a.out:libtest.so
```

```
$ export LD_PROFILEBUCKET_SIZE=16
$ ./a.out
          hello world
          I in a
$ unset LD_PROFILE
$ unset LD_PROFILEBUCKET_SIZE
$ ls gmon.out
           gmon.out
$ gprof  >aa1
```

5. CONCLUSION

The profiling support available starting HPUX 11i version 1.5 promises users tremendous flexibility to fine tune their applications. Full call graph can be generated for inter/intra library calls, including system libraries. However in order to generate full call graph of a system library it is necessary that symbol table of the library should not be chopped. This new feature promises user some great performance improvement in their applications. This information can show you which pieces of your program are slower than you expected, and these pieces might be candidates for rewriting to make your program execute faster. It can also tell you which functions are being called more or less often than you expected. This may help you spot bugs that had otherwise been unnoticed.

6. ACKNOWLEDGEMENTS

I would gratefully like to acknowledge and thank the following people for all the technical guidance and support extended during the enhancement of Shared Library Profiling Support on HPUX 11i version 1.5 and preparation of this paper.

- Cary Coutant
- Kumar Rangarajan
- David Gross
- Marcel Moolenar
- Loreena Wong
- Mike Liaw
- Shreekanth Prabhu
- Malay Shah
- Krishna Chythanya V.
- Shruthi K
- Sudhanshu Gupta