
Dynamic Memory Solutions **Dynamic Profile™**

User Guide

Version 1.1 for Solaris™

Document Number DP20-S-011-0

Copyright © 2004-2005 Dynamic Memory Solutions LLC

www.dynamic-memory.com

Notices

Information in this document is subject to change without notice.

Trademarks

Dynamic Memory Solutions and Dynamic Profile are trademarks of Dynamic Memory Solutions LLC.

Sun, Sun Microsystems, and Solaris are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

All other trademarks are the property of their respective owners.

Comments

Comments about this document may be sent to:

Dynamic Memory Solutions LLC
Publications Department
30703 Round Lake Road
MT DORA FL 32757

Or sent to sales@dynamic-memory.com

Copyright

Copyright © 2004-2005 Dynamic Memory Solutions LLC All Rights Reserved

Note to U.S. Government Users – Documentation related to restricted rights.

Contents

<i>Notices</i>	2
<i>Trademarks</i>	2
<i>Comments</i>	2
<i>Copyright</i>	2
Contents	3
Figures	5
Preface	6
<i>About Dynamic Memory Solutions</i>	6
<i>Support</i>	6
Overview	8
<i>Major features of Dynamic Profile</i>	8
<i>Benefits of using Dynamic Profile</i>	9
Feature Descriptions	10
<i>CPU Time Profiling</i>	10
<i>Elapsed Time Coverage</i>	10
<i>Variable Sampling Frequency</i>	10
<i>Real Time Reporting</i>	10
<i>Stop and Start Profiling</i>	10
<i>Focused Output</i>	11
<i>Ease of Use</i>	12
Installing Dynamic Profile	13
<i>Installation Requirements</i>	13
<i>Obtaining the Dynamic Profile software</i>	13
<i>Demonstration version</i>	13
<i>Full version</i>	13
<i>Installation procedure</i>	14
<i>Before you install</i>	14
<i>Demonstration version installed</i>	14
<i>Demonstration version NOT installed</i>	15
<i>First install</i>	15
<i>Additional product install</i>	16
<i>User configuration</i>	16

Quick Start	18
Using Dynamic Profile	19
<i>Setup</i>	19
<i>Invoking Dynamic Profile</i>	19
<i>Using Options</i>	20
<i>Supported Options</i>	20
DYNROOT = [directory].....	20
DYNLICENSEFILE = [path and file name].....	20
DYNOUTPUT = [directory].....	20
DYNQUIET	20
DYNPROFUPDATE = [seconds].....	21
DYNPROFSAMPLERATE = [milliseconds].....	21
DYNPROFNOSTARTUP.....	21
DYNPROFMAXSTACKDEPTH = [integer].....	22
DYNVERBOSE.....	22
DYNPROFSHOWALL.....	22
The Dynamic Profile Report	23
<i>Profile Report Layout</i>	24
Banner.....	24
Total by Function Call.....	24
Total by Function.....	26
Examples	28
<i>Example Source Code</i>	28
<i>Example Scenario 1</i>	32
<i>Example Scenario 2</i>	35
<i>Example Scenario 3</i>	38
<i>Example Scenario 4</i>	41
Limitations of Dynamic Profile	43
<i>Known Limitations</i>	43
FAQ	44

Figures

Figure 1 - Sending a signal to toggle profiling.....	11
Figure 2 - Using showHostDetails to obtain key information.....	15
Figure 3 - Using hostname and hostid to obtain key information.....	15
Figure 4 - Installing the product tar file.....	16
Figure 5 - User configuration command example.....	17
Figure 6 - Example Report File – Single Report Update.....	23
Figure 7 - Example Source Code (Part 1 of 3).....	29
Figure 8 - Example Source Code (Part 2 of 3).....	30
Figure 9 - Example Source Code (Part 3 of 3).....	31
Figure 10 - Example Scenario 1 Report (Part 1 of 2).....	33
Figure 11 - Example Scenario 1 Report (Part 2 of 2).....	34
Figure 12 - Example Scenario 2 Report (Part 1 of 2).....	36
Figure 13 - Example Scenario 2 Report (Part 2 of 2).....	37
Figure 14 - Example Scenario 3 Report (Part 1 of 2).....	39
Figure 15 - Example Scenario 3 Report (Part 2 of 2).....	40
Figure 16 - Example Scenario 4 Report	42

Preface

About Dynamic Memory Solutions

From Day One, we at Dynamic Memory Solutions have focused our efforts to provide you with the highest quality software development tools that enable you to maximize your productivity and the quality and stability of your code. We backup our products with the level of technical support that you expect from experienced software professionals that have been developers themselves. Our goal is to help you use our products in the most effective way possible and we welcome your opinions and suggestions.

You can contact DMS via:

❖ Our Website -

www.dynamic-memory.com

❖ Email -

Sales -- sales@dynamic-memory.com

Support – support@dynamic-memory.com

❖ Telephone -

Voice – 1-877-293-4144

Fax – 1-877-293-4144

Support

DMS is fully committed to supporting our products. With each sale, DMS establishes a support agreement where technical support is provided via one or more of the following:

- **Website** - Please visit our website at www.dynamic-memory.com for answers to the most frequently asked questions regarding installation and use.
- **Email** - support@dynamic-memory.com -- We attempt to answer email questions within three business days though most will be answered by the next business day.
- **Telephone** - Telephone support is obtained by calling 1-877-293-4144 during our regular business hours. Your support agreement may include telephone support, otherwise, it is provided at additional cost.
- **On-site support** - DMS will travel to your site or connect to your server via an externally accessible network. Our consultants will help diagnose and resolve difficult problems found while using our tools. On-site support is negotiated on a case-by-case basis. Please call our telephone support line for more information on this service.

- **Training** – DMS will arrange training seminars via electronic delivery or in person at your facility. Please contact us for additional information.

Overview

Dynamic Memory Solutions' Dynamic Profile is an easy-to-use tool that provides a run-time execution profile of your software. It identifies the functions and methods where your program spends its time while it is running. As with all products in our Dynamic Suite, Dynamic Profile does not require modification of your application to insert compile-time or link-time instrumentation.

Profile information includes five major pieces of information: stack depth, elapsed time, user time, system time, and function name. Together, these show you where your CPU cycles are being used. This information is invaluable during software performance analysis since it indicates which code to improve to get the greatest run-time performance benefit.

Major features of Dynamic Profile

- Accurate profiling of function's elapsed, user and system times. Learn what code uses the most CPU cycles.
- Variable sampling frequency – user configurable. Adjust sampling rate to minimize overhead or maximize level of detail.
- Variable reporting intervals – user configurable. Output report is updated and can be examined while program is running.
- Start and stop profiling – user trigger turns profiling on and off during program execution. Application profiling can be delayed until after program has initialized.
- Report filtering – ordinarily unimportant profile information is suppressed. Options allow user to see the extra information only when required.
- Non-intrusive - minimal performance impact. No need to recompile or relink your program.
- C and C++ support.
- 32 bit and 64 bit support.
- Multi thread support.

Benefits of using Dynamic Profile

Dynamic Profile saves your company time and money by enabling your organization to quickly identify performance bottlenecks in the development, test and production environments. Dynamic Profile clearly identifies the software areas where you need to apply development resources to improve software performance. The ease of use and clarity of results reduces the software development labor required. By fixing the performance problems Dynamic Profile identifies, you are able to process more data and serve more users with fewer hardware resources.

Feature Descriptions

CPU Time Profiling

Dynamic Profile reports the amount of CPU time used by each function and method. Both system and user time is reported. The report also indicates the percentage of system and user time used by each function and method.

Elapsed Time Coverage

Similar to CPU time profiling, this feature reports the time elapsed during each function. A function with an elapsed time that is significantly larger than CPU time usually indicates it is waiting on a shared resource. Possibilities include the CPU, an I/O device, network latency, or a database.

Variable Sampling Frequency

Dynamic Profile allows the user to balance output accuracy against performance impact. The default sample rate of one millisecond provides the most accurate profile. A less frequent sample rate may be specified to reduce the resources used by Dynamic Profile, but may also reduce the reporting accuracy.

You may override the default sample rate with the `DYNPROFSAMPLERATE` option.

Real Time Reporting

Dynamic Profile writes intermediate results to the output file in order to provide the user with profiling information while the program is still running. Unlike some code profilers that wait until your program has terminated to provide results, Dynamic Profile makes results available at a user definable update rate so you can monitor your application under changing input and loading conditions.

Profile updates are written at a default interval of 60 seconds and include a time stamp. Each report update writes the running or cumulative profile function times to the output file. You may override the default reporting interval with the `DYNPROFUPDATE` option.

Stop and Start Profiling

Dynamic Profile provides you with a triggering mechanism that turns profiling on and off while your program is running. Many user programs have a one-time initialization prior to beginning steady-state operation. Usually with long running processes, this initialization is not of importance. However, there may be cases when it will skew the results if included in the profile of the application. Dynamic Profile allows you to delay profiling until the point you decide to start it. This may be immediately after the program initialization is complete or at another time of your choosing. Additionally, you may also trigger profiling to stop and resume as often as you desire.

When you send a trigger to stop profiling, the report is updated and the Dynamic Profile internal timers are reset. This report update shows the cumulative function times since profiling started or resumed. If profiling is restarted, only function times since the restart are included in the subsequent report updates.

Dynamic Profile uses either user signal one (USR1) or user signal two (USR2) to toggle profiling. If the user program includes a signal handler for the signal that is sent, Dynamic Profile will not receive the signal and profiling will not start or stop. Most processes do not have handlers for the USR1 and USR2 signals.

One way to send a signal to your program is to use the Solaris **kill** command. The **kill** command has a number of format options but the most common is:

kill <signal> <PID>

where <signal> is **-USR1** for user signal one or **-USR2** for user signal two and PID is the program identifier of your executable program. For example, to send user signal one to your program with a PID of 12345, send the command shown in Figure 1.

```
kill -USR1 12345
```

Figure 1 - Sending a signal to toggle profiling

The option DYNPROFNOSTART is used when you want to start with profiling disabled. If set, Dynamic Profile waits until a trigger is received before starting.

Focused Output

Profiling a large application can produce a large quantity of output. Dynamic Profile filters out normally insignificant information to allow you to focus more quickly on important, time consuming areas.

Functions that use an insignificant amount of elapsed, user, and system time are not reported by default. To qualify for this omission, all three times must be less than one percent (1%) of the elapsed, user, and system time used by the application. The option DYNSHOWALL can be used to display all functions measured including those with elapsed, user, and system times of less than 1%.

Dynamic Profile also filters information for functions whose names cannot be identified. When using certain built-in C functions such as “%” (modulo division), C may call a library function even though your source code does not explicitly show the call. Reporting of these “No symbol information” functions is usually an unnecessary level of information. Instead, the time spent in such functions is included at the caller level (function that made the implicit call). The option DYNVERBOSE can be used to include “No symbol information” functions in the report.

Ease of Use

As with other Dynamic Memory Solutions products, there is no need to recompile or relink your program. You are able to start using Dynamic Profile as soon as it is installed. Additionally, there is very little performance impact on your application or server allowing you to use Dynamic Profile in the development, test, and production environments.

Installing Dynamic Profile

Installation Requirements

Minimum system requirements for Dynamic Profile:

- ❖ A Sun Solaris distribution version 2.8 or 2.9. Please contact us for current availability on other versions and platforms.
- ❖ SUNWlibC is required by the Dynamic Profile product. Please ensure it is installed on the target machine.
- ❖ At least 50 MB of available disk space

Obtaining the Dynamic Profile software

There are two versions of Dynamic Profile. A limited function demonstration version that is available for download from our website and the full function version available for purchase. For either version, the first step is to read and accept the Dynamic Profile software license. If you do not accept our license, you should not download our software and should remove any existing copies immediately.

Demonstration version

The demonstration version of Dynamic Profile may be downloaded from our website. This version does not require a license key. The installation procedure for this version is the same as for the full version except that the steps related to license key installation are skipped.

Full version

Please contact us to obtain a full version copy of Dynamic Profile. We prefer electronic distribution via our website or email but special arrangements can be made for delivery on Compact Disc.

When you contact us, we will request information from you regarding the system you will use to host our software. This information permits us to generate a license key to permit our software to run on that server or workstation. Your license key options are:

- ❖ Evaluation license – An evaluation license key enables full program operation for a short time to allow evaluation of Dynamic Profile. Normally there is no charge for this type of license.
- ❖ Full license – A full license key enables you to use Dynamic Profile on the designated server or workstation and entitles you to minor program updates. Normally this entitles you to use the software for an unlimited period of time as long as you abide by the license.

Installation procedure

Dynamic Memory Solutions offers a number of products to maximize your productivity and software quality. Normally, you will want to install these products in the same directory. Installation of the Dynamic Profile software does not modify any system files and while installation as root is suggested, it is not required but facilitates use by multiple users. Our products can be installed on a shared file system but you will still require a license key for each client machine.

Dynamic Profile is distributed as a compressed tar file (tar.gz) with a top-level directory named dms-1.1/. The complete directory structure is:

dms-1.1/bin	executable binaries
/lib	supporting libraries
/env	environment setup scripts
/license	product license files
/docs	product documentation
/examples	example program and results
/extras	optional or supporting files

Before you install

In preparation for installation, please follow the following steps:

- ❖ Obtain the software from Dynamic Memory Solutions. This is either the limited feature, demonstration version or the full version.
- ❖ If installing the full version, obtain either an evaluation or full license key. To provide this key, we require information about the system on which our software will be running. The information we require is purely hardware related and requires no disclosure of software or data on your server.

There are two ways to determine the required information about your hardware system:

Demonstration version installed

If you have installed the demonstration version of any DMS product on the target system, you can execute the command **showHostDetails** located in the DMS product bin/ directory. Before issuing the command, be sure to set DYNROOT and update your PATH variable as show in Figure 2.

```
export DYNROOT=<directory>
. $DYNROOT/env/dyn.env
showHostDetails

host name Solaris1
host id 1234ABCD
```

Figure 2 - Using **showHostDetails** to obtain key information

Demonstration version NOT installed

To obtain the required information, issue the commands **hostname** and **hostid** as either root or user and record the values as shown in Figure 3. In the example, the commands capture the information to a file that can be sent to us for key generation.

After obtaining the information, email the file or print it and fax it to us. Please use a descriptive name that identifies the system to you and we will reference this name when we send you the license key. When you request multiple license keys, this allows you to match the key to the corresponding system.

```
hostname > /tmp/<descriptive name>.dmshost
hostid >> /tmp/<descriptive name>.dmshost
```

Figure 3 - Using **hostname** and **hostid** to obtain key information

First install

If you are installing your first DMS product follow these steps to install the Dynamic Profile software.

- ❖ Decide where you will install Dynamic Profile and create the directory if it does not exist. For example, directory /apps. Ensure that users of the product will have read access to this directory. You may also install the product to a shared file system (e.g. using NFS).
- ❖ Change to the directory chosen for the install (e.g. /apps). Uncompress and install the product as shown in Figure 4.

This will result in the creation of the dms-1.1/ directory and subdirectories containing the product code. For example, if you installed the software in the /apps directory,

```
cd /< install directory >
cp <full path / product filename > .
gunzip < product filename >
tar -xvf < product filename>
```

Figure 4 - Installing the product tar file

directory /apps/dms-1.1 will be created. This is your product root directory and you set the environment variable DYNROOT to it.

- ❖ Copy the Dynamic Profile license file (sent separately) to a directory where users have read permission. The default location and name for the license file is \$DYNROOT/license/license.dat. License keys may be kept on a shared file system if that is where you installed the software.

If you choose a license key file directory and name other than the default \$DYNROOT/license/license.dat, the \$DYNLICENSEFILE environment variable must be set to the full path and name of the file before you run the software.

Additional product install

If you have other Dynamic Memory Solutions products installed, it is highly recommended that you install Dynamic Profile in the same location. This allows users to access any of the products at the same release level without needing to modify environment variables including \$DYNROOT, \$PATH and \$LD_LIBRARY_PATH when switching between products.

When installing a new release of a DMS product, the name of the top-level directory in the archive is different in order to support multiple installed releases. For example, if you have already installed Dynamic Profile release 1.0 in the /apps directory, the product is in directory /apps/dms-1.0. When you install Dynamic Profile release 1.1, it will be installed in directory /apps/dms-1.1. You can use either version by setting your \$DYNROOT environment variable to corresponding directory.

User configuration

Before running Dynamic Profile, you must configure the environment by setting a few variables. The DYNROOT environment variable must be set to the directory containing the release (e.g. /apps/dms-1.1). Once this is set, you run a script we provide (dms-1.1/ env/dyn.env) to set the other variables PATH and LD_LIBRARY_PATH.

The commands in Figure 5 may be added to a user's shell profile. For example, add these lines to the .kshrc file to configure the variables and Dynamic Profile will always be available by simply typing its name.

```
export DYNROOT=<directory>  
. $DYNROOT/env/dyn.env
```

Figure 5 - User configuration command example

Quick Start

Dynamic Profile is easy to use and comes with default settings appropriate for most C and C++ applications. This section describes how to use the basic features of Dynamic Profile to profile a program.

To use Dynamic Profile:

1. Compile and link your program as you normally do. It is not necessary to compile with the debug flag, but Dynamic Profile can provide more information if the application is compiled with the debug flag enabled.
2. Ensure that the \$DYNROOT environment variable is set and that the \$DYNROOT/bin directory is in your \$PATH. Set the \$DYNOUTPUT environment variable to the directory where you want Dynamic Profile to write the results.
3. Execute

```
dynprof <programname> [<program options>]
```

where <programname> is the name of your executable program, and <program options> are the command line parameters you use for your program.

4. Dynamic Profile writes its results to \$DYNOUTPUT/<programname>.<pid>.profile . By default, the results are reported once per minute. This output file continues to grow as long as the program is running.

If you do not set \$DYNOUTPUT or you do not have write permissions to that directory, the output file is created in the /tmp directory.

Using Dynamic Profile

Setup

Before using Dynamic Profile, it must be installed on your server with a valid license key. Please see the product installation section, starting on page 13 of this document for more information.

In order to use Dynamic Profile, the `$DYNROOT` environment variable must be set to the directory chosen for the product at installation. Additionally, you will need to add the `$DYNROOT/bin` directory to your `$PATH` environment variable. The setting of these variables is independent of the location of your application program and may be included in your profile for increased ease of use. See User configuration on page 16 for detailed instructions.

Output from Dynamic Profile is directed to the directory specified by the `$DYNOUTPUT` environment variable. In a development environment, it may be most convenient to set this to the directory used to compile and link the executable. In a test environment where the executable is already built and resides in a shared, read-only directory, `$DYNOUTPUT` may be set to a directory owned by the tester with write access. If `$DYNOUTPUT` has not been set or is set to a directory to which the user does not have write access, output will be directed to the `/tmp` directory.

Invoking Dynamic Profile

The application program should be compiled and linked with your normal procedures. It is not necessary to compile the application with the debug flag, but Dynamic Profile can provide more information with the debug flag enabled.

Set any additional Dynamic Profile options you require for your test. Options are identified in the following section and are set using environment variables.

After performing these steps, invoke Dynamic Profile as follows:

```
dynprof <programname> [<program options>]
```

where `<programname>` is the name of your executable program, and `<program options>` are the command line parameters you use with your program.

Using Options

Dynamic Profile supports a number of options. These options are enabled by setting the environment variable indicated by the option name to the desired value. They are disabled by unsetting the environment variable. For example, to suppress output banner information, execute this statement:

```
export DYNQUIET=1
```

To disable garbage collection banner suppression after it has been enabled, execute this statement:

```
unset DYNQUIET
```

Supported Options

DYNROOT = [directory]

The DYNROOT variable must be set in the user's environment. It defines the installation directory for Dynamic Profile.

DYNLICENSEFILE = [path and file name]

This variable overrides the default path to the license file. When this variable is not set, Dynamic Profile looks for the license file \$DYNROOT/license/license.dat. If the license file is not located in the default directory or does not have the default name, then this option designates its location including full path and file name.

DYNOUTPUT = [directory]

The DYNOUTPUT variable specifies the directory where Dynamic Profile writes its output files. If the DYNOUTPUT option is not specified, or designates a directory where the user does not have write permissions, then the output files will be written to the /tmp directory.

DYNQUIET

Dynamic Profile writes banner information and results summary to standard output. Setting the DYNQUIET environment variable prevents this writing to standard output (stdout). This

is useful in a variety of situations including program I/O from the stdin/stdout, piping program output to another program or running the target program in the background.

Set this variable to prevent banner information from being written to standard output. If this variable is not set, banner information is written.

DYNPROFUPDATE = [seconds]

The DYNPROFUPDATE variable specifies the number of seconds between report updates during a Dynamic Profile run. When this variable is not set, a value of 60 seconds is used. Depending on the user program and the problem you are analyzing, a longer or shorter update rate may be desired and DYNPROFUPDATE can be set to a positive integer value greater than one. Dynamic Profile always writes a least one report update when the user program terminates.

DYNPROFSAMPLERATE = [milliseconds]

This variable indicates how frequently, in milliseconds, Dynamic Profile samples the target program. When this variable is not set, a value of one millisecond is used. One millisecond is the smallest acceptable value and results in the fastest sampling rate and greatest accuracy of the results. However, on slow servers or for cases where less accuracy is acceptable, DYNPROFSAMPLERATE can be set to an integer value greater than one.

DYNPROFNOSTARTUP

This variable indicates whether Dynamic Profile begins profiling at the beginning of program execution or waits for a trigger. If DYNPROFNOSTARTUP is not set, profiling begins immediately. The trigger is used to stop and resume profiling at any time. This option simply defines the initial state.

If DYNPROFNOSTARTUP is set, profiling does not begin until the user sends either a user signal one (USR1) or user signal two (USR2) to the user program. Either signal acts as a toggle and when a second signal is sent, profiling is stopped. By using the signal, profiling may be enabled and disabled as desired.

If the user program includes a signal handler for the signal that is sent, Dynamic Profile will not receive the signal and profiling will not begin. Most processes do not have handlers for the user signals.

DYNPROFMAXSTACKDEPTH = [integer]

This option allows you to set the maximum depth of functions to report in a profiled stack. It can be useful if the Dynamic Profile report is cluttered by deep stacks that are not interesting.

If used, DYNPROFMAXSTACKDEPTH should be set to a positive integer.

DYNVERBOSE

This variable indicates whether the report output should include information for functions whose names cannot be identified. When using certain built-in C functions such as “%” (modulo division), C may call a library function even though your source code does not explicitly show the call. Reporting of these “No symbol information” functions is usually an unnecessary level of information. Instead, the time spent in such functions is included at the caller level (function that made the implicit call).

If DYNVERBOSE is set, then Dynamic Profile includes the profile statistics for these unknown symbols. In conjunction with the debugger, you can use the information to determine what your program is doing when the call is made. By default, unknown symbols are not reported.

DYNPROFSHOWALL

This variable controls whether the report includes functions that use an insignificant amount of elapsed, user, and system time. By default, if a function's elapsed, user, and system times are each less than one percent (1%), the function is not included in the report unless it calls a function that exceeds the 1% threshold.

If DYNPROFSHOWALL is set, all functions sampled, including those with elapsed, user, and system times of less than 1%, are included in the report.

The Dynamic Profile Report

Dynamic Profile produces a report containing the execution profile of your program. When your program terminates and at the default or specified update interval, the report is written to \$DYNOUTPUT/<programname>.<pid>.profile . If you do not set \$DYNOUTPUT or you do not have write permissions to that directory, the output file is created in the /tmp directory. The stack depth, elapsed time, user CPU time and system CPU time are written for each function in the program.

```
Dynamic Profile
Dynamic Memory Solutions
www.dynamic-memory.com
Copyright 2004,2005 All rights reserved

Dynamic Profile Report #1 January 10 10:05

> Total by Function Call

Call      Elapsed      User      System
Depth  Seconds  %   Seconds  %   Seconds  %   Function Name and Address
0       9.94 100    9.80 100    0.01 100    _start 0X10668
1       9.94 100    9.80 100    0.01 100    main 0X10A30
2       1.87  18     1.85  18     0.00  0     testFunction1 0X1082C
2       5.57  56     5.48  55     0.01  0     testFunction2 0X108DC
3       5.29  53     5.21  53     0.01  0     testFunction3 0X10980
4       5.29  53     5.21  53     0.01  0     fopen 0XF9F12170
5       5.29  53     5.21  53     0.01  0     _endopen 0XF9F0EAEC
6       5.29  53     5.21  53     0.01  0     _libc_open 0XF9F164AC
7       5.29  53     5.21  53     0.01  0     __open 0XF9F1D500
3       0.28  2      0.28  2      0.00  0     strcat 0XF9ECF470
2       2.50  25     2.47  25     0.00  0     testFunction4 0X109BC
3       2.50  25     2.47  25     0.00  0     testFunction1 0X1082C

> Total by Function

      Elapsed      User      System
Seconds %   Seconds %   Seconds %   Function Name and Address
      5.29  53     5.21  53     0.01  0     __open 0XF9F1D500
      1.76  17     1.74  17     0.00  0     testFunction1 0X1082C
      0.28  2      0.28  2      0.00  0     strcat 0XF9ECF470

End of Dynamic Profile report #1
```

Figure 6 - Example Report File – Single Report Update

Profile Report Layout

Figure 6 on page 23, shows a report generated by running Dynamic Profile. For this report, only DYNROOT and DYNOUTPUT were set. The report consists of a *Banner* section and *Total by Function Call* and *Total by Function* sections containing the profiling data. The *Total by Function Call* and *Total by Function* sections repeat if there is more than one report update.

Banner

The banner identifies the program generating the output, Dynamic Profile.

Total by Function Call

This section of the report provides the performance profile for each function. A section is generated for each report update. The report is updated at a default rate of every 60 seconds but can be configured via the DYNPROFUPDATE option (see page 21). There is always at least one report update. Each report update provides a running cumulative time and percentage for each function.

Only functions that are actually sampled and their calling functions appear in this section. If a function executes so quickly that Dynamic Profile does not sample it, it is not listed. A missing function does not mean that it is not called and does not execute. Use the default sampling rate (see DYNPROFSAMPLERATE on page 21), if it is important to see a function that executes very quickly.

Short programs that execute quickly or call many functions just once or a few times each before terminating may generate erratic profiling results from run to run. Since Dynamic Profile is a statistical profiler, results are based on the number of samples taken and fewer samples decrease reporting accuracy.

For each report update, there are five major pieces of information: stack depth (*Call Depth*), elapsed time (*Elapsed Seconds*), user time (*User Seconds*), system time (*System Seconds*), and function (*Function Name and Address*).

Call Depth

The Call Depth column provides the stack frame level of the corresponding function and helps indicate its calling sequence. When a function is called in different places in a program, it may be called at different stack frame levels. In Figure 6, **testFunction1** is called at a depth of 2

```
  _start -> main -> testFunction1
```

and **testFunction1** is called at a depth of 3

```
  _start -> main -> testFunction4 -> testFunction1.
```

The stack depth is visually reflected in the indentation of the Function Name and Address.

Elapsed Seconds

The Elapsed Seconds column provides the wall clock time, in seconds, spent in and the overall percent of wall clock time used by the corresponding function. The value reported includes the elapsed time while in the function and in all the functions it calls.

Elapsed time is comprised of the time executing in user and system mode and the time spent waiting for a resource. In the multiprocessing environment provided by an operating system such as Solaris, each program is vying for a slice of the processor's time. The more programs that are ready to run, the fewer time slices available for each program. While other programs are receiving their time slice, time continues to elapse for your program. Additionally, when a process requires a resource other than the processor, it may have to wait for the resource to become available. An example of a resource your program may have to wait for is file input/output (I/O). When your program reads from a file, a system call is made to access a hard drive – an over simplification but sufficient for this discussion. While the hard drive is physically locating the data on the drive, your program is suspended, waiting for the operation to complete. Once the operating system has the data ready, your program is allowed to run again. As with the wait for a time slice, time continues to elapse for your function.

User Seconds

User Seconds is the time, in seconds, spent executing your function. This value includes only the processor time your function actually receives and not the time spent waiting for a resource. This column provides the both the actual time and the percentage of overall user time used by the corresponding function. The value reported includes the user time while in the function and in all the functions it calls. It does not include the system time used by the corresponding function.

This percentage of user time used by the function is usually more important to know than its elapsed time since it is not affected by the load (number of other programs running) on the system.

System Seconds

System Seconds is the time, in seconds, used by Solaris, in support of your function. This value includes only the system time your function actually receives and not the time spent waiting for a resource. This column provides the both the actual time and the percentage of overall user time used in support of the corresponding function. The value reported includes the system time while in the function and in all the functions it calls.

Function Name and Address

This column provides the function name and the address of the function. The function names are indented according to its stack depth when called. The address of the function is in hexadecimal and is the same address you would see if you were using a debugger.

If a function name is not available, which may happen when library functions are implicitly called, the function name will be reported as “No symbol information” followed by the

address sampled. In this case, the address can be used to determine the actual function called and to locate the summary information for the function.

The DYNVERBOSE option must be set in order to see “No symbol information” entries.

You may notice that the first function listed is **_start**. When you execute a C program, Solaris passes control to initialization code that executes prior to calling the program's **main** function and does things such as setting up command line and environment options. Usually, this **_start** code executes very quickly and the time for **_start** and **main** will be the same.

Total by Function

This section of the report provides the profile summary information for each function (see Figure 6 on page 23). A section is generated for each report update. The report is updated at a default rate of every 60 seconds but can be configured via the DYNPROFUPDATE option (see page 21). There will always be at least one reporting window. Each report update provides a running cumulative time and percentage for each function.

Only functions that are actually sampled appear in this section. Calling functions shown in the *Total by Function Call* section will not appear if they were not themselves sampled. If a function executes so quickly that Dynamic Profile does not see it, it will not be listed. A missing function does not mean that it is not called and does not execute. Use the default sampling rate (see DYNPROFSAMPLERATE on page 21), if it is important to see a function that executes very quickly.

For each report update, there are four major pieces of information: elapsed time (*Elapsed Seconds*), user time (*User Seconds*), system time (*System Seconds*), and function (*Function Name and Address*).

Elapsed Seconds

The Elapsed Seconds column provides the wall clock time, in seconds, spent in and the overall percent of wall clock time used by the corresponding function. The value reported includes the total elapsed time while in the function, independent of the call stack.

User Seconds

User Seconds is the time, in seconds, spent executing your function. This value includes only the processor time your function actually receives and not the time spent waiting for a resource. This column provides both the actual time and the percentage of overall user time used by the corresponding function everywhere it is called. It does not include the system time used by the corresponding function.

System Seconds

System Seconds is the time, in seconds, used by Solaris, in support of your function. This value includes only the system time your function actually receives and not the time spent waiting for a resource. This column provides both the actual time and the percentage of

overall user time used in support of the corresponding function. The value reported includes the system time while in the function everywhere it is called.

Function Name and Address

This column provides the function name and the address of the function. The address of the function is in hexadecimal and is the same address you would see if you were using a debugger.

If a function name is not available, which may happen when library functions are implicitly called, the function name will be reported as “No symbol information” followed by the address of the sample. In this case, the address can be used to determine the actual function called and to locate the calling information for the function in the *Total by Function Call* section of the report.

The DYNVERBOSE option must be set in order to see “No symbol information” entries.

Examples

Example Source Code

Figures 7 through 9 contain the source code for the example. This simple program has 4 functions plus a **main**. As you will observe in the source code, the functions use varying amounts of CPU time as they are each executed a different number of times and perform different tasks.

The functions are:

- **testFunction1** - performs mathematical operations
- **testFunction2** - performs string operations and calls **testFunction3**
- **testFunction3** - performs file I/O
- **testFunction4** - performs memory allocations and calls **testFunction1**

The function calling sequence is:

```
main  -> testFunction1
      -> testFunction2      -> testFunction3
      -> testFunction3
      -> testFunction4      -> testFunction1
```

testFunction1, shown in Figure 7 on page 29 performs math operations. Math operations are performed in user space and incur mostly user time.

testFunction2, shown in Figure 8 on page 30 performs string operations. String operations are performed in user space and incur mostly user time. Additionally, this function calls **testFunction3**.

testFunction3, also shown in Figure 8, performs file operations with calls to C library functions **fopen**, **fprintf**, and **fclose**. These file functions perform part of their operation in user time and the remainder in system time.

testFunction4, shown in Figure 9 on page 31, performs a large number of memory allocation and deallocations. Since memory allocations are made from the user heap, much of this operation takes place in user time though some underlying system calls are required to manage access to the heap. Additionally, this function calls **testFunction1**.

main, also shown in Figure 9, calls each function within a loop. The loop is used to insure that the example program uses enough time to be accurately profiled.

```
#include <stdlib.h>
#include <stdio.h>
#include <strings.h>

void testFunction1();
void testFunction2();
void testFunction3();
void testFunction4();

// Function: testFunction1()
// Purpose: perform a number of mathematical operations

void testFunction1()
{
    double number = 1.2345;
    long bigNumber = 98765432;
    double answer;
    long result;
    int i;
    int loopCnt1 = 100000;

    for (i = 0; i < loopCnt1; i++)
    {
        if ( i != 0)
        {
            answer = i/number;
            result = bigNumber%i;
        }
    }
}
```

Figure 7 - Example Source Code (Part 1 of 3)

```

// Function: testFunction2()
// Purpose: perform a number of string operations and
//          call testFunction3()

void testFunction2()
{
    char *string1 = "This is string 1";
    char *string2 = "This is string 2";
    char result[500];
    long compareValue;
    int i;
    int loopCnt2 = 10000;

    for ( i = 0; i < loopCnt2; i++)
    {
        compareValue = strcmp(string1, string2);

        strcpy(result, string1);
        strcat(result, string2);
    }

    testFunction3();
}

// Function: testFunction3()
// Purpose: perform file operations

void testFunction3()
{
    char *string1 = "This is string 1";
    FILE *filePtr;
    int i;
    int loopCnt3 = 100;

    for ( i = 0; i < loopCnt3; i++)
    {
        filePtr = fopen("/tmp/example_file", "w+");
        if ( filePtr != NULL )
        {
            fprintf(filePtr, "%s\n", string1);
            fclose(filePtr);
        }
    }
}

```

Figure 8 - Example Source Code (Part 2 of 3)

```

void testFunction4()
{
    char *string1;
    int i;
    int loopCnt4 = 10000;

    for (i = 0; i < loopCnt4; i++)
    {
        string1 = (char *) malloc (16000);
        string1[0] = '\0';
        free (string1);
    }

    testFunction1();
}

// Function: main()
// Purpose: call functions to demonstrate various types of
//           processing. Loop enough times that profiling
//           information is collected.

int main()
{
    int i;
    int loopCnt = 100;

    for (i = 0; i < loopCnt; i++)
    {
        testFunction1();
        testFunction2();
        testFunction3();
        testFunction4();
    }

    return 0;
}

```

Figure 9 - Example Source Code (Part 3 of 3)

Example Scenario 1

We compiled the example source code shown in Figures 7 through 9 with the command

```
gcc -g -o example example.c
```

and ran Dynamic Profile with the command

```
dynprof ./example
```

The only options used were DYNROOT and DYNOUTPUT.

The output file *example.11939.profile* was created and is shown in Figures 10 and 11.

The **example** program took 15.09 seconds of wall clock time to run as indicated by the elapsed seconds at call depth 0. During those 15.09 seconds, 11.23 were user seconds and 3.40 were system seconds. (See Figure 10, Highlight “A”) The program was executed on a lightly loaded server with virtually no competition for system resources.

In the program, **testFunction1** is called twice, once from **main** and again from **testFunction4**. This is evident in the report by examining the call depth. Highlight “B” in Figure 10 shows the call to **testFunction1** from **main**, indicated by a call depth of 1 for **main** and 2 for **testFunction1**. Figure 10, highlight “C”, shows the call to **testFunction1** from **testFunction4**. In this case, **testFunction4** has the call depth of 2 and **testFunction1** has a call depth of 3.

Highlight “D” points to the call of **testFunction2** from **main**. In addition to the call depth, the indentation of the function name also indicates the caller.

testFunction3 is also called twice but with very different profiling results. The first time it is called in the sample program (see “E”), its elapsed and user times are much greater than when it is called later (see “F”). **testFunction3** performs file I/O and suffers greater overhead during its first access of the file system.

Highlight “G” in Figure 10 points out **_fflush_u**, one of the many functions that are called as a result of using library functions.

testFunction4 (“H”) is not listed in the *Total by Function* part of the report (shown in Figure 11). This is an indication that **testFunction4** ran too quickly to be sampled at the current sample rate. We know that **testFunction4** was called because **testFunction1** was sampled.

Dynamic Profile Report #1 January 10 12:48

> Total by Function Call

Call Depth	Elapsed Seconds	Elapsed %	User Seconds	User %	System Seconds	System %	Function Name and Address
0	15.09	100	11.23	100	3.40	100	_start 0X10690
1	15.09	100	11.23	100	3.40	100	main 0X10AD8
2	0.89	5	0.77	6	0.10	2	testFunction1 0X10854
2	11.63	77	9.79	87	1.59	46	testFunction2 0X10900
3	11.41	75	9.60	85	1.57	46	testFunction3 0X109AC
4	0.77	5	0.62	5	0.14	4	_fprintf 0XF9F07F4C
5	0.77	5	0.62	5	0.14	4	_doprnt 0XF9F04AB4
6	0.77	5	0.62	5	0.14	4	_findbuf 0XF9F0EDD8
7	0.32	2	0.26	2	0.05	1	_isatty 0XF9EC11BC
8	0.32	2	0.26	2	0.05	1	ioctl 0XF9F1D038
7	0.45	3	0.36	3	0.09	2	_fstat64 0XF9F1B920
4	4.88	32	3.99	35	0.79	23	fclose 0XF9F11D9C
5	4.44	29	3.63	32	0.73	21	_fflush_u 0XF9F11CF4
6	4.44	29	3.63	32	0.73	21	_xflsbuf 0XF9F11AB8
7	4.44	29	3.63	32	0.73	21	_libc_write 0XF9F1FA70
5	0.44	2	0.36	3	0.07	2	_private_close 0XF9F1C770
4	5.76	38	4.99	44	0.64	18	fopen 0XF9F12170
5	5.76	38	4.99	44	0.64	18	_endopen 0XF9F0EAEAC
6	5.76	38	4.99	44	0.64	18	_libc_open 0XF9F164AC
7	5.76	38	4.99	44	0.64	18	__open 0XF9F1D500
3	0.21	1	0.19	1	0.02	0	strcat 0XF9ECF470
2	2.22	14	0.49	4	1.57	46	testFunction3 0X109AC
3	0.31	2	0.07	0	0.22	6	_fprintf 0XF9F07F4C
4	0.31	2	0.07	0	0.22	6	_doprnt 0XF9F04AB4
5	0.31	2	0.07	0	0.22	6	_findbuf 0XF9F0EDD8
6	0.10	0	0.02	0	0.07	2	_isatty 0XF9EC11BC
7	0.10	0	0.02	0	0.07	2	ioctl 0XF9F1D038
6	0.21	1	0.05	0	0.15	4	_fstat64 0XF9F1B920
3	1.27	8	0.28	2	0.90	26	fclose 0XF9F11D9C
4	1.15	7	0.25	2	0.82	23	_fflush_u 0XF9F11CF4
5	1.15	7	0.25	2	0.82	23	_xflsbuf 0XF9F11AB8
6	1.15	7	0.25	2	0.82	23	_libc_write 0XF9F1FA70
4	0.12	0	0.03	0	0.09	2	_private_close 0XF9F1C770
3	0.64	4	0.14	1	0.45	13	fopen 0XF9F12170
4	0.64	4	0.14	1	0.45	13	_endopen 0XF9F0EAEAC
5	0.64	4	0.14	1	0.45	13	_libc_open 0XF9F164AC
6	0.64	4	0.14	1	0.45	13	__open 0XF9F1D500
2	0.35	2	0.18	1	0.14	4	testFunction4 0X10A54
3	0.33	2	0.17	1	0.13	3	testFunction1 0X10854

~~~~~ cut ~~~~~

Figure 10 - Example Scenario 1 Report (Part 1 of 2)

```

~~~~~ cut ~~~~~

> Total by Function

 Elapsed User System
Seconds % Seconds % Seconds % Function Name and Address

 6.40 42 5.13 45 1.09 32 __open 0XF9F1D500
 5.59 37 3.88 34 1.54 45 _libc_write 0XF9F1FA70
 0.66 4 0.40 3 0.23 6 _fstat64 0XF9F1B920
 0.56 3 0.39 3 0.15 4 _private_close 0XF9F1C770
 0.47 3 0.40 3 0.06 1 testFunction1 0X10854
 0.42 2 0.28 2 0.12 3 ioctl 0XF9F1D038
 0.21 1 0.19 1 0.02 0 strcat 0XF9ECF470

End of Dynamic Profile Report #1

```

Figure 11 - Example Scenario 1 Report (Part 2 of 2)

Figure 11 contains the *Total by Function* portion of the report. As described on page 26, this section of the report indicates which functions were sampled while profiling. All of the entries here appear in the *Total by Function Call* portion of the report, however, not all of the functions shown in the *Total by Function Call* portion of the report appear here.

Leaf-level functions such as **strcat** (Figure 11, “I”) that are only called from one location in the source code have time values matching those reported in the *Total by Function Call* portion of the report. Leaf-level functions such as **ioctl** and **\_private\_close** (“J” and “K”) that are called from multiple locations or in multiple code paths, appear multiple times in the *Total by Function Call* portion of the report but only once in this section which provides the time totals.

Non leaf-level functions such as **testFunction1** do not easily add up to or reflect the times shown in the *Total by Function Call* section (see Example Scenario 4 on page 41). *Note, functions called by testFunction1 are not shown in this example because they fall below the one percent reporting threshold.* That is because non leaf-level function shown in the *Total by Function* section do not include the times for the leaf-level functions they call. However, these leaf-level function times are still reflected in the *Total by Function Call* section. Non-reported functions and rounding errors may also cause the percentages in this section to not total 100%.

## Example Scenario 2

For this example, we use the same example source code but change the *loopCnt* variable in **main** from 100 to 1000. This causes the program to run longer and to exceed the default report update time (60 seconds). Accordingly, multiple report sections are generated. As with Example Scenario 1, only the DYNROOT and DYNOUTPUT variables are used.

In this scenario, Dynamic Profile generated the report *example.11997.profile*. A portion of this report is shown in Figures 12 and 13 on pages 36 and 37 respectively.

There are three Dynamic Profile report updates in the output file. They are numbered sequentially and the times in each are cumulative over the profile. The first report shows **\_start** with an elapsed time of 60.17 seconds (“A”). The second report shows **\_start** with an elapsed time of 121.05 seconds (“B”). The third and final report indicates **\_start** with a total elapsed time of 179.95 seconds (“C”). The first two reports are generated at approximately 60 second intervals. The third report is generated at program termination.

Because the program runs for a much longer period of time, many more samples are taken and some functions appear that are not shown in Example Scenario 1 on page 32. In the section of the report shown, **strcat** is shown as being called by **testFunction2** (see Figure 13 “D”). Additionally, **strcat** is shown in the *Total by Function* section of the report (“E”). Generally, functions that don't appear unless the program is run for a long period of time are not very important since they run for very little time. At the completion of this scenario, **strcat** barely registers time wise.

It is also possible for functions to appear in some report updates because of changing conditions on the server. If the load on the server increases due to other programs, it may increase the elapsed time for some functions being profiled enough to include them in the report. In subsequent report updates, they may once again disappear when their elapsed, user, and system time percentages drop below one percent.

```

~~~~~ cut ~~~~~
Dynamic Profile Report #1 January 10 13:04
> Total by Function Call A

```

| Call Depth | Elapsed Seconds | User Seconds | User % | System Seconds | System % | Function Name and Address |                       |
|------------|-----------------|--------------|--------|----------------|----------|---------------------------|-----------------------|
| 0          | 60.17           | 100          | 44.80  | 100            | 13.56    | 100                       | _start 0X10690        |
| 1          | 60.17           | 100          | 44.80  | 100            | 13.56    | 100                       | _main 0X10AD8         |
| 2          | 1.91            | 3            | 1.50   | 3              | 0.34     | 2                         | testFunction1 0X10854 |
| 2          | 47.91           | 79           | 40.61  | 90             | 6.33     | 46                        | testFunction2 0X10900 |
| 3          | 47.30           | 78           | 40.11  | 89             | 6.24     | 46                        | testFunction3 0X109AC |
| 4          | 5.20            | 8            | 4.36   | 9              | 0.74     | 5                         | _fprintf 0XF9F07F4C   |
| 5          | 5.20            | 8            | 4.36   | 9              | 0.74     | 5                         | _doprnt 0XF9F04AB4    |
| 6          | 5.20            | 8            | 4.36   | 9              | 0.74     | 5                         | _findbuf 0XF9F0EDD8   |
| 7          | 2.42            | 4            | 2.06   | 4              | 0.32     | 2                         | _isatty 0XF9EC11BC    |
| 8          | 2.42            | 4            | 2.06   | 4              | 0.32     | 2                         | _ioctl 0XF9F1D038     |
| 7          | 2.78            | 4            | 2.31   | 5              | 0.42     | 3                         | _fstat64 0XF9F1B920   |
| 4          | 18.58           | 30           | 15.18  | 33             | 3.03     | 22                        | fclose 0XF9F11D9C     |

```

~~~~~ cut ~~~~~
5 3.08 5 0.66 1 2.20 16 _libc_open 0XF9F164AC
6 3.08 5 0.66 1 2.20 16 __open 0XF9F1D500
2 1.52 2 0.82 1 0.60 4 testFunction4 0X10A54
3 1.44 2 0.79 1 0.55 4 testFunction1 0X10854
> Total by Function

```

| Elapsed Seconds | %  | User Seconds | %  | System Seconds | %  | Function Name and Address |
|-----------------|----|--------------|----|----------------|----|---------------------------|
| 26.60           | 44 | 21.22        | 47 | 4.67           | 34 | __open 0XF9F1D500         |
| 20.65           | 34 | 14.32        | 31 | 5.69           | 41 | _libc_write 0XF9F1FA70    |
| 3.45            | 5  | 2.45         | 5  | 0.89           | 6  | _fstat64 0XF9F1B920       |
| 2.71            | 4  | 2.12         | 4  | 0.53           | 3  | ioctl 0XF9F1D038          |
| 2.71            | 4  | 1.87         | 4  | 0.75           | 5  | _private_close 0XF9F1C770 |
| 1.53            | 2  | 1.11         | 2  | 0.34           | 2  | testFunction1 0X10854     |

```

End of Dynamic Profile Report #1

Dynamic Profile Report #2 January 10 13:05
> Total by Function Call B

```

| Call Depth | Elapsed Seconds | User Seconds | User % | System Seconds | System % | Function Name and Address |                       |
|------------|-----------------|--------------|--------|----------------|----------|---------------------------|-----------------------|
| 0          | 121.05          | 100          | 90.23  | 100            | 27.16    | 100                       | _start 0X10690        |
| 1          | 121.05          | 100          | 90.23  | 100            | 27.16    | 100                       | _main 0X10AD8         |
| 2          | 3.67            | 3            | 2.80   | 3              | 0.70     | 2                         | testFunction1 0X10854 |
| 2          | 96.59           | 79           | 82.02  | 90             | 12.60    | 46                        | testFunction2 0X10900 |
| 3          | 95.08           | 78           | 80.74  | 89             | 12.42    | 45                        | testFunction3 0X109AC |
| 4          | 9.63            | 7            | 8.06   | 8              | 1.38     | 5                         | _fprintf 0XF9F07F4C   |
| 5          | 9.63            | 7            | 8.06   | 8              | 1.38     | 5                         | _doprnt 0XF9F04AB4    |
| 6          | 9.63            | 7            | 8.06   | 8              | 1.38     | 5                         | _findbuf 0XF9F0EDD8   |
| 7          | 3.75            | 3            | 3.12   | 3              | 0.55     | 2                         | _isatty 0XF9EC11BC    |
| 8          | 3.75            | 3            | 3.12   | 3              | 0.55     | 2                         | _ioctl 0XF9F1D038     |
| 7          | 5.87            | 4            | 4.94   | 5              | 0.83     | 3                         | _fstat64 0XF9F1B920   |
| 4          | 36.79           | 30           | 30.14  | 33             | 5.92     | 21                        | fclose 0XF9F11D9C     |

```

~~~~~ cut ~~~~~

```

Figure 12 - Example Scenario 2 Report (Part 1 of 2)

```

~~~~~ cut ~~~~~
 3 6.07 5 1.32 1 4.31 15 fopen 0XF9F12170
 4 6.07 5 1.32 1 4.31 15 _endopen 0XF9F0EAEC
 5 6.07 5 1.32 1 4.31 15 _libc_open 0XF9F164AC
 6 6.07 5 1.32 1 4.31 15 __open 0XF9F1D500
 2 2.97 2 1.57 1 1.19 4 testFunction4 0X10A54
 3 2.76 2 1.50 1 1.07 3 testFunction1 0X10854

> Total by Function

 Elapsed User System
Seconds % Seconds % Seconds % Function Name and Address
54.73 45 43.85 48 9.43 34 __open 0XF9F1D500
41.66 34 28.98 32 11.41 42 _libc_write 0XF9F1FA70
7.19 5 5.23 5 1.76 6 _fstat64 0XF9F1B920
4.79 3 3.23 3 1.40 5 _private_close 0XF9F1C770
4.52 3 3.29 3 1.10 4 ioctl 0XF9F1D038
3.00 2 2.03 2 0.80 2 testFunction1 0X10854

End of Dynamic Profile Report #2

Dynamic Profile Report #3 January 10 13:06
> Total by Function Call
Call Elapsed User System
Depth Seconds % Seconds % Seconds % Function Name and Address
0 179.95 100 112.29 100 33.73 100 _start 0X10690
1 179.95 100 112.29 100 33.73 100 main 0X10AD8
2 11.36 6 4.60 4 0.85 2 testFunction1 0X10854

~~~~~ cut ~~~~~

 4    64.28 35   51.29 45   6.14 18    fopen 0XF9F12170
 5    64.28 35   51.29 45   6.14 18    _endopen 0XF9F0EAEC
 6    64.28 35   51.29 45   6.14 18    _libc_open 0XF9F164AC
 7    64.28 35   51.29 45   6.14 18    __open 0XF9F1D500
 3    4.39 2     1.34 1     0.13 0     strcat 0XF9ECF470
 2    23.67 13   4.89 4     15.71 46   testFunction3 0X109AC

~~~~~ cut ~~~~~

 3 8.55 4 1.63 1 5.28 15 fopen 0XF9F12170
 4 8.55 4 1.63 1 5.28 15 _endopen 0XF9F0EAEC
 5 8.55 4 1.63 1 5.28 15 _libc_open 0XF9F164AC
 6 8.55 4 1.63 1 5.28 15 __open 0XF9F1D500
 2 5.83 3 2.45 2 1.59 4 testFunction4 0X10A54
 3 5.48 3 2.31 2 1.41 4 testFunction1 0X10854

> Total by Function

 Elapsed User System
Seconds % Seconds % Seconds % Function Name and Address
72.83 40 52.92 47 11.42 33 __open 0XF9F1D500
53.67 29 35.75 31 13.99 41 _libc_write 0XF9F1FA70
13.19 7 6.27 5 2.30 6 _fstat64 0XF9F1B920
9.07 5 4.07 3 1.98 5 _private_close 0XF9F1C770
7.82 4 3.37 3 0.98 2 testFunction1 0X10854
6.20 3 3.80 3 1.33 3 ioctl 0XF9F1D038
4.39 2 1.34 1 0.13 0 strcat 0XF9ECF470

End of Dynamic Profile Report #3

```

**C**

**D**

**E**

Figure 13 - Example Scenario 2 Report (Part 2 of 2)

### Example Scenario 3

Using the same executable from Example Scenario 1 but adding the DYNPROFSHOWALL option to the DYNROOT and DYNOUTPUT variables, generates the Dynamic Profile report example.12749.profile. A portion of this report is shown in Figures 14 and 15 on pages 39 and 40.

By using the DYNPROFSHOWALL option, the report includes entries for functions whose elapsed, user, and system times were each below one percent. In the *Total by Function Call* section (see Figure 14) the output includes functions **strcmp** and **strcat**, both called from **testFunction2**, with all zero percentages. Additionally, **testFunction4** is shown to cause calls to **\_ti\_mutex\_unlock**, **malloc**, and **\_malloc\_unlocked** that didn't appear before adding the option.

These entries, with the exception of **malloc**, are also reported in the *Total by Function* section (see Figure 15). The absence of **malloc** is an indication that no profile samples were taken while in the function and that it is included in the *Total by Function Call* due to its call to **\_malloc\_unlocked**.

```

~~~~~ cut ~~~~~

> Total by Function Call

Call      Elapsed      User      System
Depth  Seconds %   Seconds %   Seconds %   Function Name and Address
0       15.05 100   11.15 100   3.40 100   _start 0X10690
1       15.05 100   11.15 100   3.40 100   main 0X10AD8
2        0.60  3     0.47  4     0.08  2     testFunction1 0X10854
2       11.77 78     9.96 89     1.56 45     testFunction2 0X10900
3       11.66 77     9.88 88     1.54 45     testFunction3 0X109AC
4        1.12  7     0.94  8     0.16  4     _fprintf 0XF9F07F4C
5        1.12  7     0.94  8     0.16  4     _doprnt 0XF9F04AB4
6        1.12  7     0.94  8     0.16  4     _findbuf 0XF9F0EDD8
7        0.67  4     0.59  5     0.07  2     _isatty 0XF9EC11BC
8        0.67  4     0.59  5     0.07  2     ioctl 0XF9F1D038
7        0.45  2     0.35  3     0.09  2     _fstat64 0XF9F1B920
4        4.47 29     3.68 32     0.70 20     fclose 0XF9F11D9C
5        4.09 27     3.36 30     0.64 18     _fflush_u 0XF9F11CF4
6        4.09 27     3.36 30     0.64 18     _xflsbuf 0XF9F11AB8
7        4.09 27     3.36 30     0.64 18     _libc_write 0XF9F1FA70
5        0.38  2     0.31  2     0.06  1     _private_close 0XF9F1C770
4        6.08 40     5.26 47     0.68 20     fopen 0XF9F12170
5        6.08 40     5.26 47     0.68 20     _endopen 0XF9F0EAEC
6        6.08 40     5.26 47     0.68 20     _libc_open 0XF9F164AC
7        6.08 40     5.26 47     0.68 20     __open 0XF9F1D500
3        0.02  0     0.02  0     0.00  0     strcmp 0XF9EB2B24
3        0.04  0     0.03  0     0.01  0     strcat 0XF9ECF470
2        2.28 15     0.50  4     1.61 47     testFunction3 0X109AC
3        0.23  1     0.05  0     0.16  4     _fprintf 0XF9F07F4C
4        0.23  1     0.05  0     0.16  4     _doprnt 0XF9F04AB4
5        0.23  1     0.05  0     0.16  4     _findbuf 0XF9F0EDD8
6        0.11  0     0.02  0     0.08  2     _isatty 0XF9EC11BC
7        0.11  0     0.02  0     0.08  2     ioctl 0XF9F1D038
6        0.12  0     0.03  0     0.08  2     _fstat64 0XF9F1B920
3        1.27  8     0.28  2     0.90 26     fclose 0XF9F11D9C
4        1.10  7     0.24  2     0.78 22     _fflush_u 0XF9F11CF4
5        1.10  7     0.24  2     0.78 22     _xflsbuf 0XF9F11AB8
6        1.10  7     0.24  2     0.78 22     _libc_write 0XF9F1FA70
4        0.17  1     0.04  0     0.12  3     _private_close 0XF9F1C770
3        0.78  5     0.17  1     0.55 16     fopen 0XF9F12170
4        0.78  5     0.17  1     0.55 16     _endopen 0XF9F0EAEC
5        0.78  5     0.17  1     0.55 16     _libc_open 0XF9F164AC
6        0.78  5     0.17  1     0.55 16     __open 0XF9F1D500
2        0.40  2     0.22  1     0.15  4     testFunction4 0X10A54
3        0.37  2     0.21  1     0.14  4     testFunction1 0X10854
3        0.01  0     0.01  0     0.00  0     _ti_mutex_unlock 0XF9CC18B0
3        0.01  0     0.01  0     0.00  0     malloc 0XF9EC1CB8
4        0.01  0     0.01  0     0.00  0     _malloc_unlocked 0XF9EC1CF4

~~~~~ cut ~~~~~

```

Figure 14 - Example Scenario 3 Report (Part 1 of 2)

```
~~~~~ cut ~~~~~  
  
> Total by Function  
  
    Elapsed      User      System  
Seconds %   Seconds %   Seconds %   Function Name and Address  
    6.86 45     5.43 48     1.24 36   __open 0XF9F1D500  
    5.19 34     3.60 32     1.42 41   _libc_write 0XF9F1FA70  
    0.78  5     0.61  5     0.15  4   ioctl 0XF9F1D038  
    0.57  3     0.37  3     0.18  5   _fstat64 0XF9F1B920  
    0.55  3     0.35  3     0.18  5   _private_close 0XF9F1C770  
    0.53  3     0.39  3     0.12  3   testFunction1 0X10854  
    0.05  0     0.04  0     0.01  0   testFunction2 0X10900  
    0.04  0     0.03  0     0.01  0   strcat 0XF9ECF470  
    0.02  0     0.02  0     0.00  0   strcmp 0XF9EB2B24  
    0.01  0     0.01  0     0.00  0   _malloc_unlocked 0XF9EC1CF4  
    0.01  0     0.01  0     0.00  0   _ti_mutex_unlock 0XF9CC18B0  
  
End of Dynamic Profile Report #1  
  
~~~~~ cut ~~~~~
```

Figure 15 - Example Scenario 3 Report (Part 2 of 2)

## Example Scenario 4

Using the same executable from Example Scenario 3 but adding the DYNVERBOSE option to the DYNROOT, DYNOUTPUT, and DYNPROFSHOWALL variables, generates the Dynamic Profile report example.12806.profile. A portion of this report is shown in Figure 16 on page 42.

By adding the DYNVERBOSE option, the report includes entries for profiling samples where no symbol information is available. In this particular example, we used **gdb** to determine that the “No symbol information” entries under **testFunction1** are from the use of modulo division in the function. These entries are also reported in the *Total by Function* section. The function's hexadecimal address is used to match the corresponding entries.

The presence of multiple, adjacent “No symbol information” entries should not be read as multiple function calls. Since symbol information is not available, Dynamic Profile cannot determine the start and end of a function so the address that is sampled is provided.

You may notice that in this example we demonstrate the DYNVERBOSE option with the aid of the DYNPROFSHOWALL option. In many cases, “No symbol information” entries are associated with very little time and will not be displayed without DYNPROFSHOWALL.

```

~~~~~ cut ~~~~~

> Total by Function Call

Call      Elapsed      User      System
Depth Seconds %    Seconds %    Seconds %    Function Name and Address
0         15.07 100      11.19 100      3.42 100      _start 0X10690
1         15.07 100      11.19 100      3.42 100      main 0X10AD8
2         0.36  2       0.26  2       0.09  2       testFunction1 0X10854
3         0.01  0       0.00  0       0.01  0       No symbol information 0XF9F40E8C
3         0.01  0       0.01  0       0.00  0       No symbol information 0XF9F40E98
3         0.01  0       0.01  0       0.00  0       No symbol information 0XF9F40E9C
3         0.05  0       0.05  0       0.00  0       No symbol information 0XF9F40EA0
3         0.15  1       0.10  0       0.04  1       No symbol information 0XF9F40EAC
3         0.03  0       0.03  0       0.00  0       No symbol information 0XF9F40EB8
2         12.03 79      10.26 91      1.52 44      testFunction2 0X10900
3         11.86 78      10.11 90      1.50 43      testFunction3 0X109AC
4         0.95  6       0.77  6       0.16  4       _fprintf 0XF9F07F4C
5         0.95  6       0.77  6       0.16  4       _doprnt 0XF9F04AB4
6         0.95  6       0.77  6       0.16  4       _findbuf 0XF9F0EDD8
7         0.14  0       0.09  0       0.04  1       _isatty 0XF9EC11BC
8         0.14  0       0.09  0       0.04  1       ioctl 0XF9F1D038
7         0.81  5       0.68  6       0.11  3       _fstat64 0XF9F1B920
4         5.00  33      4.12  36      0.78  22      fclose 0XF9F11D9C

~~~~~ cut ~~~~~

> Total by Function

 Elapsed User System
Seconds % Seconds % Seconds % Function Name and Address
6.73 44 5.40 48 1.15 33 __open 0XF9F1D500
5.34 35 3.67 32 1.50 43 _libc_write 0XF9F1FA70
1.00 6 0.72 6 0.25 7 _fstat64 0XF9F1B920
0.93 6 0.72 6 0.19 5 _private_close 0XF9F1C770
0.28 1 0.14 1 0.11 3 testFunction1 0X10854
0.22 1 0.14 1 0.07 1 No symbol information 0XF9F40EAC
0.22 1 0.11 0 0.10 2 ioctl 0XF9F1D038
0.14 0 0.13 1 0.00 0 strcat 0XF9ECF470
0.06 0 0.04 0 0.02 0 No symbol information 0XF9F40EB8
0.06 0 0.06 0 0.00 0 No symbol information 0XF9F40EA0
0.01 0 0.01 0 0.00 0 No symbol information 0XF9F40E9C
0.01 0 0.00 0 0.01 0 No symbol information 0XF9F40E8C
0.01 0 0.01 0 0.00 0 strcpy 0XF9EB2E30
0.01 0 0.00 0 0.00 0 testFunction2 0X10900
0.01 0 0.01 0 0.00 0 No symbol information 0XF9F40E98
0.01 0 0.01 0 0.00 0 No symbol information 0XF9F40EBC
0.01 0 0.00 0 0.01 0 lwp_yield 0XF9CC4D84
0.01 0 0.00 0 0.01 0 strcmp 0XF9EB2B24
0.01 0 0.01 0 0.00 0 No symbol information 0XF9F40EA8

~~~~~ cut ~~~~~

```

Figure 16 - Example Scenario 4 Report

# Limitations of Dynamic Profile

---

## Known Limitations

1. Applications that statically link libthread.a or libpthread.a cannot be profiled by Dynamic Profile. These applications should be relinked with libthread.so or libpthread.so to allow profiling.
2. Short programs that execute quickly or call many functions just once or a few times each before terminating may generate erratic profiling results from run to run. Since Dynamic Profile is a statistical profiler, results are based on the number of samples taken and fewer samples decrease reporting accuracy.

## FAQ

---

This section answers many commonly asked questions about Dynamic Profile

- My function xyz() does not appear in the report but I know it is being called. Why?

There are a number of reasons why one or more of your functions may not appear in the Dynamic Profile report. They include:

1. The function is small and the compiler has *inlined* it. Inlining is a performance enhancement compilers make that moves the code for small functions into the calling function. Inlining can also be forced by the software developer via compiler directives.
2. The function executes very quickly with respect to the Dynamic Profile sampling interval. For example, the default Dynamic Profile sampling rate is 1 millisecond. If the user function executes in less than 1 millisecond, statistically there is only a chance that Dynamic Profile will see it. Using the smallest sample rate via the DYNPROFSAMPLERATE option increases the odds you may see it in the report.

- Do I have to recompile or re-link my application to use Dynamic Profile?

No, recompilation or re-link is not required. The application must be compiled with the debug flag, `-g`, to get full information. If the executable is fully stripped, little information will be available. If a shared library is stripped, function level information will still be available. If an application is a mix of stripped and not stripped libraries, Dynamic Profile will provide as much information as is available.

- I am having problems with my 64-bit program compiled with gcc. It works fine when I do not use the Dynamic Profile, but I get an ELF class error when running with the DLC. What can I do?

Put `/usr/local/lib/sparcv9` in your `LD_LIBRARY_PATH` before `/usr/local/lib`.

- When I run my application with Dynamic Profile, I am told that `libdemangle.so.1` is missing. Why?

Dynamic Profile has a requirement that the SUN SUNWlibC package is installed. Please have this package installed and try the tool again.

- Is Dynamic Profile thread-safe?

Yes, it is thread-safe.

---

Document Number DP20-S-011-0

Copyright © 2004-2005 Dynamic Memory Solutions LLC

[www.dynamic-memory.com](http://www.dynamic-memory.com)

---