Statistical Profiling

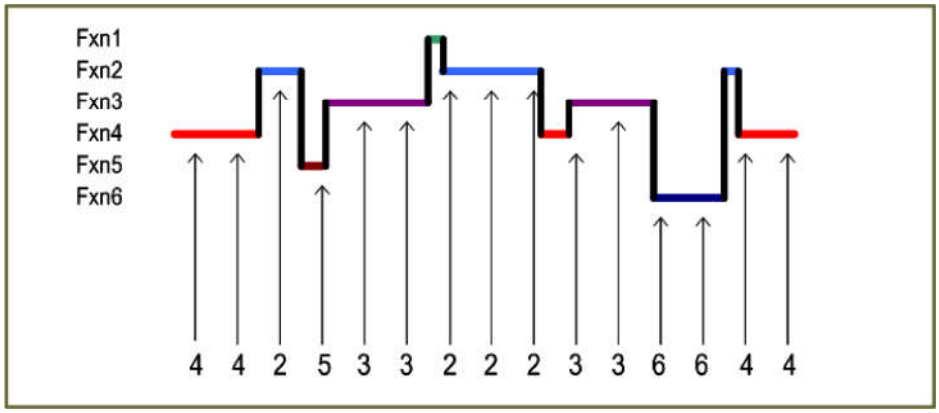
{{#switchcategory:MSP430=<McuHitboxHeader/>|C2000=<McuHitboxHeader/>|Stellaris=<McuHitboxHeader/>|TMS570=<McuHitboxHeader/>|MCU=<McuHitboxHeader/>|MAVRK=<MAVRKHitboxHeader/>|<HitboxHeader/>}}

Contents

Statistical Profiling

Overview

Statistical Profiling is a method of profiling an application by taking samples of the Program Address at regular intervals while the application is executing. These samples are then associated with either a specific function or a specific memory range, and then a simple statistical analysis is performed to determine the areas where an applications spends the largest portions of its cycles.



Statistical Profiling samples function execution periodically

Statistical Profiling focuses on the relative number of cycles spent in each function, rather than on the exact number of cycles spent in each function. Statistical Profiling is a very quick and handy way to get a first look at which functions are consuming the largest proportions of cycles so that a closer look can be taken with other, cycle accurate profiling methods.

The largest advantage of statistical profiling over traditional profiling is the amount of data that is required to be captured. Traditional profiling will capture every executed program address. Statistical profiling only captures a fraction of the entire execution path.

Statistical Profiling can be implemented on 64x+ devices that support XDS560 Trace.

Advantages of Statistical Profiling over Traditional Profiling

There are a number of huge advantages in using Statistical Profiling as an alternative to traditional profiling with XDS560 Trace.

- **Amount of Data Stored**

In traditional profiling with XDS560 Trace, we would capture every instruction in the execution path. While this provides a very detailed look at the path of execution, it also requires much additional effort before results can be analyzed. The decoding of huge amounts of trace data can be very time and resource intensive. In a case where a 64Mb buffer is used, the amount of decoded data provided will potentially expand into terabytes of decoded data. In the Statistical Profiling case, we capture only periodic samples of the PC.

- **Portion of Application Profiled**

Depending on the size of the trace buffer used, traditional profiling may fill the buffer before some of the application gets executed. With Statistical Profiling, this problem can be alleviated by choosing a larger period for sampling.

Useful Tip

When choosing the sampling interval for statistical profiling, you likely want to start with a large interval, and then, if necessary reduce it on subsequent profiles. Ideally, if you're profiling an application that processes many frames of data you would like the application to process multiple frames before the trace buffer is full.

- **Processing Effort**

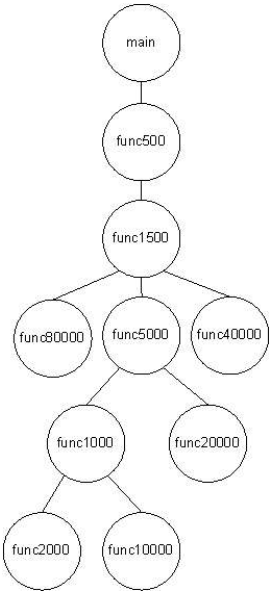
Because Statistical Profiling captures on the order of tens of thousands samples as compared to traditional profiling that can potentially capture tens of millions of samples, there is much less processing of the data to be done with Statistical Profiling. A statistical profiling analysis can be done in a matter of minutes from start to finish. Traditional profiling can require many hours for scripts to analyze all of the data that has been captured.

Disadvantages of Statistical Profiling

- The largest disadvantage of statistical profiling is the fact that it does not provide cycle accurate data. Cycle counts can potentially be estimated if the actual cycle count of a reference function is known.

Statistical Profiling Theory and Example

Statistical Profiling data can be captured through the UBM plugin in Code Composer Studio. A very simple demo is supplied to give a quick look at the type of data that is captured with Code Composer Studio. In the example, we have a number of functions with names such as func500, func1000, etc. Each function essentially consists of an empty for loop that spins for the corresponding number of cycles (40000 for func40000, etc). Each function may also call one or more of the other functions. The main function just calls func500 in an infinite loop. The call tree of the demo is shown in the diagram on the right.



One thing to keep in mind when dealing with cycle values in terms of statistical profiling is that we typically deal with proportional values. When we assume that func500 consumes ~500 cycles, and that func40000 consumes 40000 cycles, we're really assuming that their cycle values are in that proportion. It's possible that func500 could consume 1000 cycles per execution, but then we would expect func40000 to consume 80000 cycles per execution. In adding up all of the functions, we get an expected total of 160000 cycles (500+1500+80000+5000+1000+2000+10000+20000+40000). We expect func80000 to consume ~50% of the cycles (80000/160000), func40000 to consume ~25% (40000/160000), and so on, down to func500 consuming .3125% of the total cycles (500/160000).

Each function should be called nearly the same number of times. Depending on when the trace buffer completely fills, some of the functions will have been called an additional time. For example, if the Trace buffer gets full while func80000 is processing, the data will reflect func500, func1500, and func80000 being called one more time than the rest of the functions. This is why it is important to choose a sampling period that will allow multiple *frames* to be processed before the trace buffer becomes full. This will reduce truncation error as detailed in the example below.

Example

Assume that a small enough sampling period is chosen such that the trace buffer gets full at the completion of executing func80000 for the second time. So, the trace data will contain 2 executions of func500, func1500, and func80000, and 1 execution of all of the rest. The total cycles executed in this case would be 260000 (160000+500+1500+80000). But the total number of cycles consumed by func80000 would be two complete executions worth, 160000. So the result would indicate that func80000 consumed 160000/260000 (61.5%) of the total cycles. This shows an error of 11.5% from what we expected. Now take the case where a sampling period is chosen that allows the trace buffer to not get full until the end of the 11th execution of func80000. In this case, the three aforementioned functions will have execute 11 times, while the rest have executed 10 times. The total cycles for 10 executions is 1600000. So, the total cycles captured will be 1682000 (1600000+500+1500+80000). The 11 executions of func80000 consume 880000 cycles, which represent 88000/1682000 (52.3%) of the total cycles. The error has been reduced to 2.3%. Increasing the sampling period to get through 20 full buffers will further reduce the error to 1.2%.

This demo application is designed to show they type of accuracy that we can get from using statistical profiling and the effects that the choice of sampling interval can have on the results. Statistical Profiling is designed to identify the handful of top cycle intensive functions in an application. It is not designed to give an accurate cycle count for every function in the application.

Useful Tip

Be careful in selecting a sampling interval to avoid values that correspond to periodic events in the application. For example, if a timer interrupt is serviced every N cycles, you probably want to avoid using factors and multiples of N as the Statistical Profiling interval. We want the trace sampling to be done in a pseudo random fashion. If the samples are aligned with interrupts, the data can potentially be tainted because more often than not we might be sampling in the interrupt service routine.

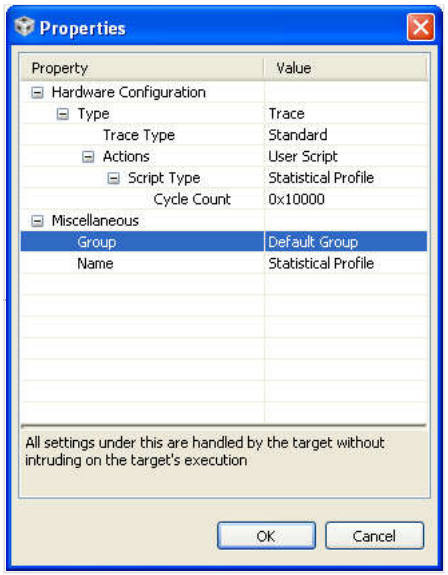
Capturing the Statistical Profiling Data

Configure trace in the Trace Control menu as desired. Typically, "Stop on Buffer Full" mode is used with Statistical Profiling, but it can also be used with "Circular Buffer" mode. If using an XDS560T, configure the desired trace buffer size. Typically, larger is better because more of the application will be captured. Because statistical profiling data captures PC addresses *randomly*, the trace data does not compress as well. Consequently, the decode of the data is faster and using a larger trace buffer is not as problematic.

The Unified Breakpoint Manager (UBM) Plugin in Code Composer Studio can be easily configured to capture statistical profiling.

- From the Breakpoint window, create a new Trace job.
- Edit the properties of the job, and select Trace Type->Standard, Actions->User Script, and Script Type->Statistical Profiling as shown in the image.

- In the cycle count field, enter a value to be used as the sampling interval.
- Click OK to save the configuration and ensure that the job is enabled in the breakpoint window.
- Run the application. You should see trace data being captured in the Trace Display Window.



Processing Statistical Profiling Data

We will use the Stand Alone trace decoder to fet the trace data for our script. The Stand Alone decoder is shipped with CCSv4 and is located at the following location <CCS4_INSTALL_DIR>\ccsv4\emulation\analysis\bin\td.exe. It can be called from the command line. Use the command td.exe -help to get details on the switches to be passed to the stand alone decoder. CCSv4 will create a file called XDS560_RecTraceData.bin in the <CCS4_Install_Dir>\ccsv4\emulation\analysis\bin\logs directory automatically whenever trace is captured. This is a copy of the raw data trace file. We will pass this file to the trace decoder to be processed.

Generating the func_info file

The first step in processing the trace data is to create a func_info.csv file from your .out file using the utility ofd6x.exe. This is a utility that is shipped with the code generation tools. The file that we need to generate will contain a list of all of the functions and filenames, along with the start, end, and length of each function. The utility ofd.exe is also shipped with the trace script package. It is located in the utils directory. Create the func_info file with the following command line

```
ofd6x --func_info <.out file name> > <.csv file name>
```

The <.out file name> parameter specifies the location and name of the .out file. The <csv file name> specifies the name of the file to be generated. You can choose whatever name you wish, but it is recommended to use the .csv extension. This file only needs to be regenerated when the .out file changes. One option is to have CCS execute this command as a post-build step every time you build your application so you will always have an up to date func_info file.

Processing the Statistical Profiling Data

As noted earlier, we will use the stand alone decoder to decode the data in the .bin file. We will then pipe this data into the trace processing application via stdin, and the results will be output via stdout. The application we will use in this case is called trace_stat_profile.exe and can be found in the trace scripts package in the .bin directory. (Note that there is a perl script that can perform the same operations called trace_stat_profile.pl, but it requires perl 5.8.8 or higher. The .exe does not require perl). The command that we will use to process the data is:

Decoder Prerequisites

In order for the Statistical Profiling application to be able to process the data, the following is required of the output of td.exe.

- The following fields must be included in the output (Default is All fields, which is fine)
  - Program Address
  - Cycles
  - Trace Status
- The timestamp must be in **Absolute** mode (-timestamp abs ***NOT THE DEFAULT***)
- The format must be in CSV_NO_TPOS_QUOTE form (-format CSV_NO_TPOS_QUOTE **This is the default**)

Command Line

One of the following commands can be used to process the trace data. Note that full or relative paths to each file supplied must be provided if all files are not in the current directory.

td.exe -bin XDS560_RecTraceData.bin -app <out file> <trace_decoder_options> | trace_stat_profile.exe –func_input <.csv file name> [-delta <sampling period>]

or

td.exe -bin XDS560_RecTraceData.bin -app <out file> <trace_decoder_options> | perl trace_stat_profile.pl --func_input <.csv file name> [–delta <sampling period>] [–noheader]

Specifying the sampling period is optional. However, it can lead to more accurate counts. There are certain occasions where trace sample captures can be delayed. Specifying the sampling period that was used when capturing the data will allow the script to determine where these samples should be allocated.

Sample Results Data

The results data is output through stdout in comma separated value form. It's essentially a histogram of the function samples captured in the trace data. A header line is prepended to the output, detailing the contents of each field. It can be supressed by using the --no_header option with the script.

```
Function name,File name,Times Encountered,Percentage
"func80000","statistical_profiling.c",30760,48.96%
"func40000","statistical_profiling.c",17515,27.88%
"func20000","static_functions.c",7141,11.37%
"func10000","static_functions.c",3572,5.69%
"func5000","static_functions.c",1787,2.84%
"func2000","static_functions.c",768,1.22%
"func1500","statistical_profiling.c",701,1.12%
"func1000","static_functions.c",385,0.61%
"func500","statistical_profiling.c",194,0.31%
```

The results can be piped to a file or to another script for further processing.

Trace Script Download

The script package can be downloaded at the following location https://www-a.ti.com/downloads/sds_support/applications_packages/trace_csv_scripts/index.htm

Note that in order to use the stand alone trace decoder, you must have version 3.0.0 of the scripting package or later.

Downloads

- Statistical Profiling Demo Source Files

Trace Script Frequently Asked Questions

Generic Scripting FAQS

- Q: Why do I sometimes see "UNKNOWN", "UNKNOWN" in the output for functions/filenames
  - A: The function/filename symbols are determined from the output of the ofd6x.exe (Object File Dump utility), which generates a list of functions and filenames from the .out file, along with their staring and ending addresses. If "UNKNOWN" values are showing in the output, it's because trace captured program execution that had a program address outside the ranges specified by the OFD utility. Common causes might be code that has been dynamically loaded/allocated which wouldn't have associated information in the .out file. You can usually determine the exact cause by looking in the file generated by ofd6x.exe and the .map file and determining why the offending program address is not defined.

Statistical Profiling Specific FAQS

- What if I want my Statistical Profiling Results to be Thread Aware?
  - The Statistical Profiling processing script doesn't natively handle thread aware applications, but if you're ambitious enough, you can add that functionality. What needs to happen is a hook function needs to be execute in the code when the operating system switches threads. That hook function should write the ID of the next thread to a global variable. The existing statistical profiling script bins everything together. It needs to be modified to create a number of bins based on the various thread IDs. The trace configuration scenario remains the same, except another job needs to be added to capture the write data to the global variable that holds the thread ids. The results should then be presented on a thread by thread basis.

{{ 1. switchcategory:MultiCore= • For technical support on MultiCore devices, please post your questions in the C6000 MultiCore Forum • For questions related to the BIOS MultiCore SDK (MCSDK), please use the BIOS Forum Please post only comments related to the article **Statistical Profiling** here.	Keystone= • For technical support on MultiCore devices, please post your questions in the C6000 MultiCore Forum • For questions related to the BIOS MultiCore SDK (MCSDK), please use the BIOS Forum Please post only comments related to the article **Statistical Profiling** here.	C2000=*For technical support on the C2000 please post your questions on The C2000 Forum. Please post only comments about the article **Statistical Profiling** here.*	DaVinci=*For technical support on DaVincoplease post your questions on The DaVinci Forum. Please post only comments about the article **Statistical Profiling** here.*	MSP430=*For technical support on MSP430 please post your questions on The MSP430 Forum. Please post only comments about the article **Statistical Profiling** here.*	OMAP35x=*For technical support on OMAP please post your questions on The OMAP Forum. Please post only comments about the article **Statistical Profiling** here.*	OMAPL1=*For technical support on OMAP please post your questions on The OMAP Forum. Please post only comments about the article **Statistical Profiling** here.*	MAVRK=*For technical support on MAVRK please post your questions on The MAVRK Toolbox Forum. Please post only comments about the article **Statistical Profiling** here.*	*For technical s please post yo questions at http://e2e.ti.cor Please post on comments abo article **Statistic Profiling** here.* }}

Links

Amplifiers & Linear	DLP & MEMS	Processors	Switches & Multiplexers
Audio	High-Reliability	• ARM Processors	Temperature Sensors & Control ICs
Broadband RF/IF & Digital Radio	Interface	• Digital Signal Processors (DSP)	Wireless Connectivity
Clocks & Timers	Logic	• Microcontrollers (MCU)	
Data Converters	Power Management	• OMAP Applications Processors	

{{#switchcategory:MSP430=<McuHitboxFooter/>|C2000=<McuHitboxFooter/>|Stellaris=<McuHitboxFooter/>|TMS570=<McuHitboxFooter/>|MCU=<McuHitboxFooter/>|MAVRK=<MAVRKHitboxFooter/>|<HitboxFooter/>}}