Technical notes on using Analog Devices DSPs, processors and development tools
Visit our Web resources http://www.analog.com/ee-notes and http://www.analog.com/processors or e-mail processor.support@analog.com or processor.tools.support@analog.com for technical support.

Cycle Counting and Profiling

Contributed by C. Gilchrist

Rev 2 – March 27, 2008

Introduction

VisualDSP++® development environment provides utilities that can be used to profile an application's performance. These utilities provide a means to analyze your application, allowing you to identify and eliminate performance bottlenecks and to optimize code. By using execution profiles, application developers can hand-tune code memory layout to achieve better system performance.

The provided tools include:

- Linear Profiler, a simulator-only utility
- Statistical Profiler, a utility for use with emulator and EZ-KIT Lite® targets
- Embedded profiling via cycle count facilities

The Linear and Statistical Profiling tools, which are non-intrusive, provide a profile of the instruction execution time, the number of clock cycles spent executing instructions, number of instructions executed, and the number of memory reads and writes an application takes to execute.

The Linear Profiler samples every PC executed, which provides a complete and accurate picture of the performance of the code. The cost of this accuracy, however, is a slow method of profiling the application, which would not be feasible on a hardware target.

The Statistical Profiler is faster, allowing you to profile without affecting the real-time characteristics of the application. The Statistical Profiler achieves this by sampling the PC periodically. It is not as accurate or consistent as the Linear Profiler, but provides a means to profile code execution on a hardware target.

The VisualDSP++ libraries also offer the ability to obtain accurate cycle counts and measure execution time by embedding code in your application to calculate and return cycle count information. This is facilitated by the cycle count registers and macros provided in the libraries.

This document describes the functional behavior of these tools, as well as their use, performance benefits, features, and limitations.

Additionally, this document is accompanied by example code for Blackfin, SHARC, and TigerSHARC processors to demonstrate the differences between profiling results returned from the simulator and on hardware.

Profiler and Cycle Count User Guide

The following instructions apply to all architectures (Blackfin®, SHARC®, and TigerSHARC®).

Statistical and Linear Profilers

Enabling and Configuring the Profiler

The Linear Profiler, which is available for simulator sessions only, is enabled via `Tools -> Linear Profiling`.

The Statistical Profiler, which is available for emulator/EZ-KIT Lite sessions only, is enabled via `Tools -> Statistical Profiling`.

You can configure a number of options within the profilers to fine-tune the profile information you require. These options are accessible by right-clicking the Profiler window and choosing `Properties`.

- The `Display` tab (Figure 1) allows you to specify the memory type, and the associated metrics you wish to profile. The available memory types are specific to the target, and the available metrics are specific to the memory type. The optional metrics allow you to include data such as the number of reads/writes, cache hits/misses, execution counts, and so on, within your profile data.

The optional metrics are available within Blackfin and TigerSHARC simulator sessions only. They are not available on hardware or SHARC simulator sessions.

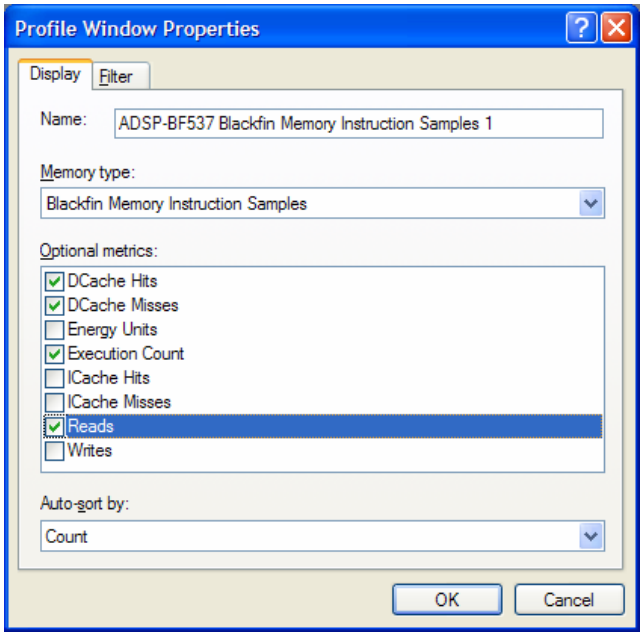

Figure 1. Display tab

- The `Filter` tab (Figure 2) allows you to specify what you wish to profile. You can choose to profile the entire memory space, selected C/C++ functions, or specific memory ranges. Specifying these options allows you to generate a more concise profile, containing only the information you require.

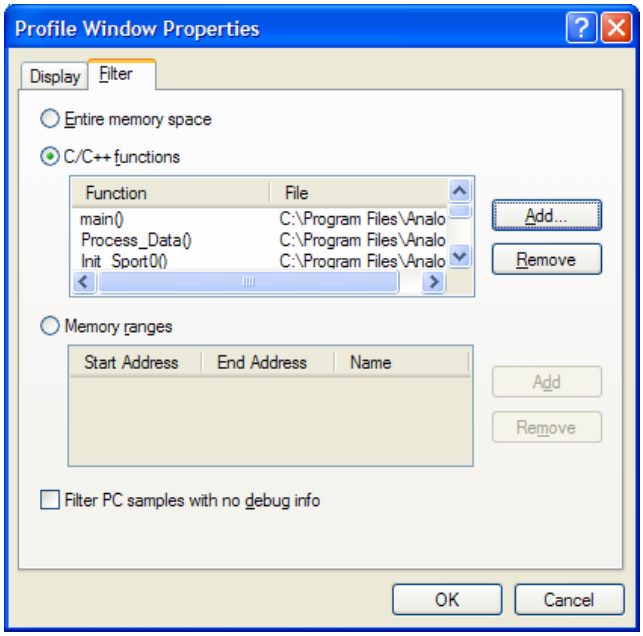

Figure 2. Filter tab

Upon specifying these options, you simply need to run your project so that the profiler can collect execution data samples.

Interpreting Results

Once the execution is complete, the profiler will display the profile information for your application. You can then view the profile in terms of either sample count, or execution percentage.

Clicking on an execution unit within the left pane (Figure 3) of the Profiler window displays the source for that execution unit in the right pane (Figure 4) (or the Disassembly window in the case of a library function). This allows you to identify the specific lines of code within your application that use the most processor time.

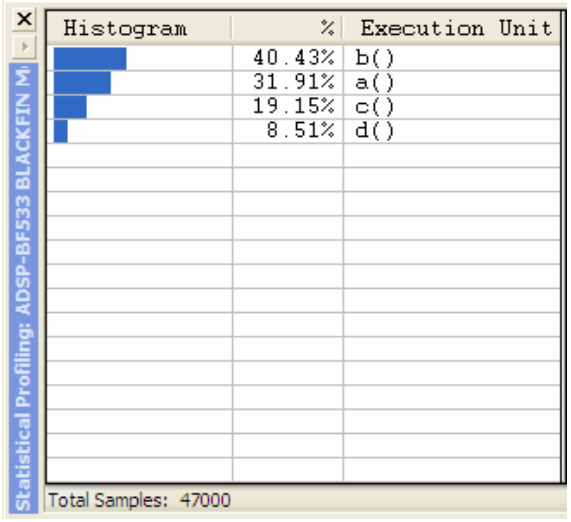

Figure 3 Left Pane of the Statistical Profiler showing the execution percentage of several functions

Profile results can be saved to a `.txt`, `.xml`, or `.prf` file, allowing them (`.prf` and `.xml` only) to be loaded later to review results, or concatenated (`.prf` and `.xml` only) with an open profile to produce merged data, providing a clearer picture of the performance of your application over multiple runs. For concatenation, the profile must have the same target and memory type as the open profile, and it must use the same optional profile metrics. All of these options are available via the `Tools -> Statistical Profiling` or `Tools -> Linear Profiling` menu.

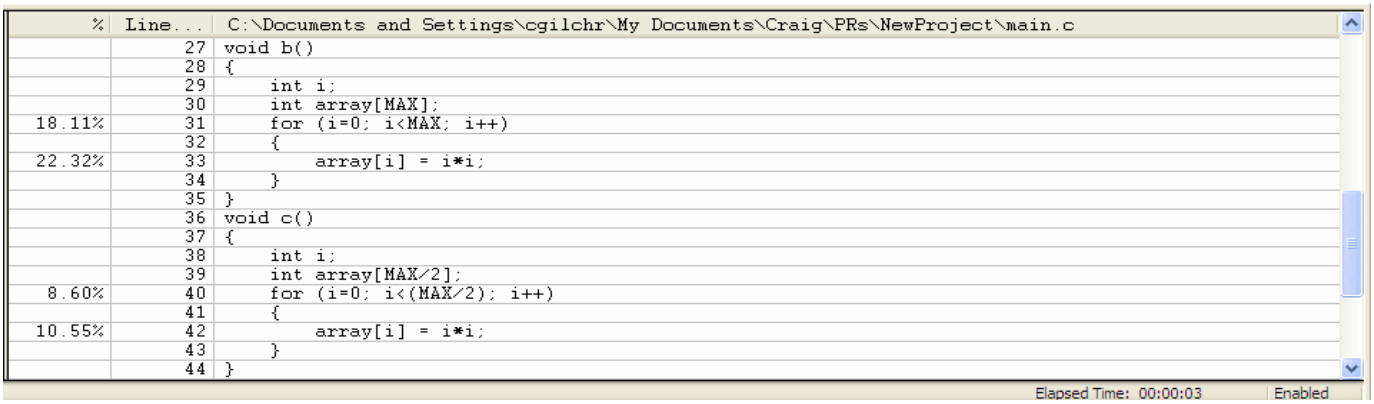

Figure 4. Right Pane of the Statistical Profiler showing the source-level execution percentage

Cycle Counting Library Functions

The cycle count macros require the `-DDO_CYCLE_COUNTS` compiler switch to be asserted.

The cycle count macros offer three different cycle count methods:

- Basic cycle count - Returns the number of clock cycles taken to execute a section of code.
- Cycle count with statistics - Returns the average, minimum, and maximum number of clock cycles taken to execute a repeated body of code.
- Cycle count using `time.h` - Returns the execution time of a section of code in seconds.

Within VisualDSP++ releases prior to release 5.0, the parameterized macros that are defined in the `cycle_count.h` and `cycles.h` header files expanded into multiple statements. This implementation led to unexpected side-effects, as it was possible that the statements would not result in the intended instructions.

Take the following as an example:

```
if (condition)
    STOP_CYCLE_COUNT(cnt2, cnt2);
```

As the STOP_CYCLE_COUNT macro expands into multiple statements, only the first statement of the expanded macro is executed conditionally. The remaining statements are always executed and therefore produce incorrect cycle counts.

To address this issue, the macros are now implemented using a statement block. With the previous cycle count macros being a sequence of instructions, it was valid to use the macros without a trailing semicolon; this is no longer the case in VisualDSP++ 5.0.

To assist with porting existing code, the compile-time __USE_CYCLE_MACRO_REL45__ macro can be defined to enable legacy support and avoid any modifications to the existing use of the cycle count macros.

Cycle counting on all architectures is facilitated by two 32-bit registers. On Blackfin processors, these are CYCLES and CYCLES2; on TigerSHARC processors, these are CCNT0 and CCNT1; on SHARC processors, these are EMUCLK and EMUCLK2. The first register increments for every cycle and wraps around to zero after 0xffffffff. When the first register wraps around, the second register increments by one, as though they are one 64-bit register. For Blackfin and TigerSHARC processors, both registers are used in the cycle counting macros. To save memory and execution time, however, the SHARC macros do not use the EMUCLK2 register. If the code being measured executes for a long period of time, EMUCLK may wrap around, and EMUCLK2 would need to be taken into account for an accurate measurement.

Basic Cycle Count

This is done using three simple macros defined within cycle_count.h. The functions are:

- PRINT_CYCLES(T)
- START_CYCLE_COUNT(S)
- STOP_CYCLE_COUNT(T,S)

The S parameter is initialized to the current value of the cycle count register in START_CYCLE_COUNT(S), and T is the total cycles calculated by subtracting S from the current value of the cycle count register when STOP_CYCLE_COUNT(T,S) is performed. The small number of cycles overhead from calling the STOP_CYCLE_COUNT() macro is also removed to give an accurate reading.

An example is given in Listing 1.

```
#include <cycle_count.h>
#include <stdio.h>
extern int main(void)
{
    cycle_t start_count;
    cycle_t final_count;
    START_CYCLE_COUNT(start_count);

Some_Function_Or_Code_To_Measure();

    STOP CYCLE COUNT(final count,start
_count);
    PRINT CYCLES("Number of cycles:
",final_count);
}
```

Listing 1. Basic cycle count example code

Cycle Count with Statistics

This cycle count technique offers five macros for the cycle count:

- CYCLES_INIT(S)
- CYCLES_START(S)
- CYCLES_STOP(S)
- CYCLES_PRINT(S)
- CYCLES_RESET(S)

The S parameter is initialized to 0 by CYCLES_INIT(S) and set to the current cycle count register value by CYCLES_START(S). CYCLES_STOP(S) extracts the current cycle count register value and accumulates statistics in S, based on the most recent call to CYCLES_START(S). CYCLES_PRINT(S) prints out the statistics for that run, providing details of the average, minimum, and maximum cycles for the measured code, and provides details about the number of times the code section was executed.

An example is given in Listing 2.

```
#include <cycles.h>
#include <stdio.h>
extern int main(void)
{
    cycle_stats_t stats;
    int i;
    CYCLES_INIT(stats);
    for (i = 0; i < LIMIT; i++)
    {
         CYCLES_START(stats);
Some_Function_Or_Code_To_Measure();
         CYCLES_STOP(stats);
    }
    CYCLES_PRINT(stats);
    CYCLES_RESET(stats);
}
```

Listing 2. Cycle counting with statistics example code

Cycle Count Using time.h

This facility uses a data-type (`clock_t`) together with the `CLOCKS_PER_SEC` macro to calculate the amount of time spent in a function or block of code. Two variables of type `clock_t` are used: one is initialized using the cycle counter register at the start of the block of code to be measured, and the second is set to the cycle counter register value when the block ends. The difference between the two variables is divided by the clock speed determined by the `CLOCKS_PER_SEC` macro, providing an accurate value for the time taken to execute the code.

An example is given in Listing 3.

```
#include <time.h>
#include <stdio.h>
extern int main(void)
{
     volatile clock t clock start;
     volatile clock t clock stop;
     double secs;
     clock_start = clock();

  Some Function Or Code To Measure();

     clock stop = clock();
     secs = ((double) (clock stop -
clock start)) / CLOCKS PER SEC;
     printf("Time taken is %e
seconds\n",secs);
}
```

Listing 3. Cycle count using time.h example code

Cycle Counting in Assembly

In order to measure cycle counts using code within an assembly project, you can simply access the cycle count registers directly. For Blackfin, see Listing 4; for TigerSHARC processors, see Listing 5; for SHARC processors, see Listing 6.

```
R2 = 0;
CYCLES = R2;
CYCLES2 = R2;
R2 = SYSCFG;
BITSET(R2,1);
SYSCFG = R2;

/*Insert code to be benchmarked here*/

R2 = SYSCFG;
BITCLR(R2,1);
SYSCFG = R2;
R2 = CYCLES
R1 = CYCLES2
```

Listing 4. Cycle counting in Blackfin assembly

```
xr2 = CCNT0;;
/*Insert code to be benchmarked here*/
xr3 = CCNT0;;
xr4 = r3 - r2;;
```

Listing 5. Cycle counting in TigerSHARC assembly

```
R6 = EMUCLK;
/*Insert code to be benchmarked here*/
R7 = EMUCLK;
R8 = R7 – R6;
```

Listing 6. Cycle counting in SHARC assembly

Accuracy of Results

The following sections describe differences that affect profile accuracy.

STDIO, Breakpoints, and Stepping Through Code on Hardware

When running your application on a hardware target, file I/O, STDIO, and stepping cause the processor to halt, and on every halt the pipeline

is flushed. When single-stepping, the pipeline flush occurs for every instruction that is stepped over. For file I/O and STDIO operations, pipeline flushes occur due to the run/halt that primio uses to transfer information. A typical example is the use of `printf()`, which performs three operations when it is used: an open, a read, and then a close. At **each** of these events, the processor is halted by a hidden breakpoint, the pipeline is flushed, the requested operation is performed, and then the processor is run again.

The cycle count macros can produce values that fluctuate greatly, depending on the manner in which the application is being executed with respect to the use of I/O functions and stepping; whenever a pipeline flush occurs, additional cycles are added to the cycle count registers. This can produce vastly different cycle counts versus the real performance of the application.

With respect to the profilers, the halts that occur when using STDIO and stepping cause the profile-gathered information to be discarded. If the code being profiled relies heavily on STDIO or is halted frequently (either by stepping or by breakpoints), the profile will never gather enough data samples to make a profile.

If the profile data is discarded, the VisualDSP++ 5.0 IDDE will generate the following warning message.

```
Statistical profiling information has
been discarded. For more details, see
online help topic: "Statistical
Profiling of Short Run Programs"
```

Comparing Linear and Statistical Profiling

Simulator vs. Hardware (ICE/EZ-KIT Lite)

It is often the case that the results within the profilers differ between simulator and hardware targets. Depending on the target you are using, you may see different results from the profiler or cycle count macros compared to other targets. One possible reason for the differences between the simulator cycle counts and the ICE/EZ-KIT Lite cycle counts is the way that the simulator initializes the model of the chip.

On the TigerSHARC simulators, this is done by running the initialization part of the loader kernel (see the Output window message "`[Info: si1108] Running Default Loader: {file name}`"). This code enables the cache and the branch target buffer (BTB). When running on TigerSHARC hardware, this does not happen. So, in order to have comparable results, include cache initialization and enable BTB in your code.

The Blackfin simulators configure a number of architectural registers to reset values. Refer to the processor's *Hardware Reference* manual.

The SHARC simulators initialize many of their registers based on values defined in the `.xml` files located within the `System\Archdef` folder of the VisualDSP++ installation.

Hardware targets also initialize many of their registers based on values defined in the `.xml` files located within the `System\Archdef` folder of the VisualDSP++ installation.

As the values in the `.xml` file may differ from those used to initialize the Blackfin or TigerSHARC simulator, hardware may come out of reset in a different state to the simulators. SHARC simulators differ, as they set their registers the same as the hardware targets do.

If the code being benchmarked relies heavily on bus activity and/or external memory, cycle counts within the simulator and on hardware may vary. Although the simulator does attempt to model external memory latencies and delays caused by external memory accesses such as back to back load/store, this is not cycle-accurate. The same is true of many peripheral interactions.

Linear vs. Statistical Profiling

Sometimes, where the Linear Profiler returns a full profile, a function is missing within the Statistical Profiler or the Statistical Profiler returns no profile at all. The reason for this

difference is the way in which the Linear Profiler and the Statistical Profiler gather the profile data.

As the Linear Profiler samples every PC, all functions are present and accurately represented in the resultant profile. The Statistical Profiler, on the other hand, samples the PC periodically, which can result is smaller functions being missed by the sampling and hence missed in the final profile. Additionally, if the application being profiled terminates in a short amount of time, the Statistical Profiler may not gather enough data points to profile the execution, resulting in a blank profile.

Additionally, if the application being profiled results in a huge number of samples at different PC addresses, the sample rate of the Statistical Profiler drops significantly, which can cause unreliable results.

Hardware Effects

The hardware can also affect the likelihood of a blank or incomplete profile. The results may vary in relation to how fast the different emulators and EZ-KIT Lite Debug Agent can collect profile information. The HPPCI-ICE and HPUSB-ICE collect profile samples quickly, while the USB-ICE and EZ-KIT Lite Debug Agent collect samples at a greatly reduced rate. If too few profile samples are collected, the Profiling window will not show any information, or the execution will be inaccurately represented.

As a general rule when using the Statistical Profiler, the program must be run continually for a long enough time to (a) collect samples, and (b) collect enough samples for the profile to become stable.

Blackfin Simulators: Cycle-Accurate vs. Compiled

For Blackfin processors, there are two simulators: the cycle-accurate simulator and the compiled (or "fast functional") simulator. Both simulators are functionally correct; however, each has separate applications. The cycle-accurate simulator is the only simulator that can approximate the latencies in your application and obtain real-world cycle counts and performance figures.

The fast functional simulator does not attempt to be cycle-accurate. It is purely a super-fast functional simulator. It does not model latencies (beyond some basics ones related to sequencer operations) and it does not account for them in the cycle count.

The fast functional simulator still supports the cycle counter because the register is present architecturally; however, you can think of it as an instruction counter, not a cycle counter in this simulator.

Additionally, although the profiling API is supported by both simulators, it is not cycle-accurate on the fast functional simulator.

Appendix A – Profiling Example

Purpose

The example code that accompanies this EE-Note is for use with EZ-KIT Lite Debug Agent and simulator sessions. It is intended to demonstrate the differences between profiling results returned from the simulator and on hardware.

Expected Results

Functions `a()` and `b()` perform the same task as one another, while function `c()` performs half as many operations, and `d()` half again. Within the simulator, the profiler should return results that closely match

the expected ratio for a:b:c:d of 4:4:2:1. You should also notice functions such as `start` and `main()` listed in the final profile, in spite of their using very few cycles.

On the EZ-KIT Lite evaluation systems via Debug Agent, the results should differ from the simulator. Running and profiling the project several times should show different percentages for the functions on each run, and on occasion you should also see function `d()`, and possibly even others, missed from the profile completely. This is due to the function simply not consuming enough processor time, and hence not being sampled often enough by the periodic sampling technique employed by the Statistical Profiler to be included as part of the profile. In addition, note that the functions such as `start` and `main()` do not show up in the profile on the EZ-KIT Lite session.

If you have access to a high-performance ICE, such as the HPUSB-ICE or the HPPCI-ICE, running this project will consistently return a profile that includes all four functions; however, the results will still not be as consistent as those returned by the Linear Profiler within the simulator.

Appendix B – Cycle Counting Example

Purpose

The cycle counting code example that accompanies this EE-Note is for use with any appropriate Blackfin, SHARC and TigerSHARC processor sessions, and demonstrates the usage of the different cycle counting facilities provided in the libraries.

Expected Results

When the cycle counting examples are executed, they should print the results of each of the three cycle counting methods to STDIO. Example output, taken from the ADSP-BF533 processor cycle counting example, is given in Listing 7.

```
Cycle Counting Facilities
-------------------------
Basic Cycle Count:
Number of cycles:420036
-------------------------
Cycle Count With Statistics:
     AVG   : 420036
     MIN   : 420036
     MAX   : 420036
     CALLS : 10
-------------------------
Cycle Count Using time.h:
Time taken was 7.071499e-04 seconds
-------------------------
```

Listing 7 - Example Cycle Count Output from the ADSP-BF533 processor

References

[1] *VisualDSP++ 5.0 C/C++ Compiler and Libraries Manual for Blackfin Processors.* Rev 5.0, August 2007. Analog Devices Inc.

[2] *VisualDSP++ 5.0 C/C++ Compiler Manual for SHARC Processors.* Rev 1.0, August 2007. Analog Devices Inc.

[3] *VisualDSP++ 5.0 C/C++ Compiler and Libraries Manual for TigerSHARC Processors.* Rev 4.0, August 2007. Analog Devices Inc.

[4] *VisualDSP++ 5.0 User's Guide.* Rev 3.0, August 2007. Analog Devices Inc.

Readings

Information on profiling and cycle counts is available from VisualDSP++ Help at the following locations:

[5] Contents -> Manuals -> Software Tool Manuals -> User's Guide -> VisualDSP++ 5.0 User's Guide -> Debugging -> Code Analysis Tools

[6] Contents -> Graphical Environment -> IDDE -> How To -> Debugging Tools and Techniques -> Statistical Profiles and Linear Profiles

[7] ***[Blackfin]:*** Manuals -> Software Tool Manuals -> Blackfin C/C++ Compiler and Library Manual -> DSP Run-Time Library -> DSP Run-Time Library Guide -> Measuring Cycle Counts

[8] ***[SHARC]:*** Manuals -> Software Tool Manuals -> SHARC C/C++ Compiler and Library Manual -> C/C++ Run-Time Library -> C and C++ Run-Time Libraries Guide -> Measuring Cycle Counts

[9] ***[TigerSHARC]:*** Manuals -> Software Tool Manuals -> TigerSHARC C/C++ Compiler and Library Manual -> C/C++ Run-Time Library -> C and C++ Run-Time Libraries Guide -> Measuring Cycle Counts

Document History

Revision	Description
Rev 2 – March 27th, 2008 by C. Gilchrist	Added Appendix B and associated example code. Updated Linear vs. Statistical Profiling section.
Rev 1 – November 29th, 2007 by C. Gilchrist	Initial release