Profiling Nios II Systems

February 2006, ver. 1.2 Application Note 391

Introduction

This application note describes a variety of ways to measure the performance of a Nios® II system with three tools: the GNU profiler, called **nios2-elf-gprof**, the timestamp interval timer peripheral, and the performance counter peripheral. Two tutorials give detailed examples of using these tools to measure performance.

The profiler tool is explained first. Measurements with the profiler tool do not require any hardware changes to your Nios II system, because the measurement is performed in software by compiler instrumentation of application code with calls to functions in the profiler library.

Next, minimally intrusive methods are examined. The performance counter peripheral and the timestamp peripheral are described and compared. The addition of both hardware peripherals and source code changes to start and stop these peripherals are necessary. The benefit of using hardware peripherals to measure performance is the accuracy of the measurement results.

Compiler speed optimizations affect functions to widely varying degrees. Compiler size optimizations also affect functions in different ways. These differences impact cache usage and resource contention, which can amplify the relative timing behavior of functions. For these reasons, profiling should be performed on release-mode code to gain the most insight on how to improve an application in its final form.

Tools

The tutorials use the following tools to measure the performance of a Nios II system.

GNU Profiler

Minimal source code changes are required to take measurements for analysis with the GNU profiler. The only changes needed are as follows:

1. Add the profiler library via a checkbox in the Nios II IDE.
2. Change the `main()` function to call `exit()`.
3. Rebuild the project.

Performance Counter Peripheral

A performance-counter unit is just a block of big counters in the hardware that are used for timing "sections" in the software. A performance counter peripheral can track up to seven sections (the default is three). Two counters are used to track each section:

- *Time*—a 64-bit time (clock-tick) counter
- *Occurrences*—a 32-bit event counter

You can change the maximum number of sections to track by editing the performance counter peripheral in SOPC Builder.

These counters let you accurately measure the execution time taken by blocks of C code. Simple, efficient, minimally-intrusive macros enable you to mark the start and end of blocks-of-interest in your program. Each block-of-interest is called a "section." The performance counter peripheral has up to seven measurement `BEGIN/END` features that let you measure each section as a fraction of some larger program. Each section must be wrapped with `BEGIN/END` by hand, so performance counters are best suited for analyzing determinism and run-time issues.

High Resolution Timer

A high resolution timer does not use a large number of logic elements (LEs) on your FPGA, nor does it require heavy instrumentation of every function call in your code to get performance measurements. Timers do require specific calls to read the timer in the sections of the source code that you want to measure, so their use is better suited for pinpointing measurements. Source code instrumentation is done by hand, but because it is less pervasive, it is also less intrusive. Many more CPU cycles are required to make two function calls—one to read the time at the beginning of a code section, and one to read the time at the end—than are consumed by the performance counter peripheral macros.

Use the GNU Profiler to Measure Code Performance

The following sections explain the advantages and limitations of using the GNU profiler for performance analysis. A tutorial is provided that demonstrates the use of the profiler to collect and analyze performance data.

GNU Profiler Advantage

The major advantage to measuring with the profiler is that it provides an overview of the entire system. Although there is some overhead, it is distributed evenly throughout the system. The functions that are identified as consuming the most CPU time will still consume the most CPU time when run at full speed without profiler instrumentation.

GNU Profiler Drawback

Adding instructions to each function call for use with the profiler affects the code's behavior. Each function is slightly larger. Each function calls another function to collect profiling information. Pulling the profiling function into instruction cache memory will generate more i-cache misses than source code that is not changed. Memory used to record the profiling data can change the behavior of the data cache. The sum of these effects, along with the longer execution time of each function call's entry and exit, can mask a time-sensitive issue that you are trying to uncover through profiling.

The profiler interpolates the percentage of time spent within each function based on periodic samplings of the program counter. The periodic samples are tied to the system clock's timer tick. The profiler is not able to take samples while interrupts are disabled and therefore not able to record the CPU cycles spent within interrupt routines.

Profiling cannot be done for individual functions. Profiling must be done for the entire system, or not at all.

The gprof profiling data is only a sampling of the program counter taken at the resolution of the system timer tick. Therefore, it is an estimation, not an exact representation, of where the CPU time is spent. The statistical significance of the sampling can be improved by increasing the frequency of the system timer tick. However, increasing the frequency of the tick costs additional overhead due to the additional time spent recording samples.

Nios II is capable of generating complete and accurate program counter trace information, although this information is not used by the Profiler. To generate this information requires a Nios II core configured with a JTAG Debug Module Level 3 or greater. Level 3 creates on-chip trace data that can be viewed in the Nios II IDE Trace View. Approximately a dozen

instructions can be captured in the on-chip trace buffer. A much larger trace can be obtained by configuring a Nios II core with a JTAG Debug Module Level 4 to generate off-chip trace information. The collection of this off-chip trace data requires the FS/2 or Lauterbach hardware.

When using the GNU profiler with your custom hardware designs, be sure to include a system clock timer, or else the profiler will not produce proper output.

Software Considerations

The profiler instruments your source code with functions to track CPU consumption.

Profiler Mechanics

The checkbox to **Link with Profiling Library** automatically turns on the `-pg` compiler switch and links profiling library code within the software component **altera_nios2** with the system library. This code counts the number of times each profiled function is called.

The `-pg` compiler option forces the compiler to insert a call to the function `mcount` (located in **altera_nios2\HAL\src\alt_mcount.S**) at the beginning of every function call. This call to `mcount` tracks every parent and child function call relationship, enabling the construction of the call graph. The option also installs a function called `nios2_pcsample` (located in **altera_nios2\HAL\src\alt_gmon.c**) that samples the foreground program counter on every system clock interrupt. When the program is executed, data is collected on the host in a file, **gmon.out**. The Nios II IDE Profiling perspective views, as well as the **nios2-elf-gprof** utility, can read this file and display profiling information about the program. The operation of the profiling code on the target is as follows:

- The compiler instruments function prologues with a call to `mcount` so it can work out which function called which other function. This data is known as function call arcs in the gprof documentation.
- An alarm is registered with the timer interrupt handler to capture information about which foreground function was executing when it was called (this is known as histogram data).
- The profiling data is stored in target memory allocated from the heap.
- When the user's code exits with a `BREAK 2` instruction, the profiling data is copied from the target to the host by **nios2-download**.

- **nios2-elf-gprof** needs both the function call arc data and the histogram data to work correctly.

Profiler Overhead

Using the profiler impacts both memory and CPU cycles.

Memory
The code overhead (the size of the .text section) is increased when profiling is enabled, due to the addition of the `nios2_pcsample` and `mcount` functions. The system timer gets instrumented with a call to `nios2_pcsample()`. Also, every function gets instrumented with a call to `mcount()`. The .text section is further increased by the size of these two functions. The impact to the .text section can be viewed by comparing changes to the .text section in the **objdump** file when profiling is enabled.

The profiler uses buckets to store data on the heap during profiling. Each bucket is two bytes in size. One bucket is created to represent samples for every 32 bytes of code in the .text section. The total number of profiler buckets allocated from the heap is the size of the .text section divided by 32. The total heap consumed by profiler buckets is therefore:

((.text section size) / 32) * 2 bytes

The profiler measures all functions in the object code that are compiled with profiling information. This includes library source code, including the system library, but does not include the Altera-provided run-time library functions. These run-time libraries are pre-built. They are not compiled with profiling information, so time spent in the run-time library functions is not tracked by the profiler.

CPU Cycles
The impact to the .text section size of the profiling information is proportional to the number of small functions in the application. Because the profiler tracks each individual function via a call to `mcount()`, the more that the application code is divided into small functions, the larger the impact in terms of both CPU time and code size. This disadvantage is offset by the higher resolution of the profiled data. To calculate the additional CPU time consumed when profiling with `mcount()`, multiply the amount of time that it takes to execute `mcount()` by the number of function invocations in the application.

For every clock tick, there is a call to `nios2_pcsample()`. To find the additional CPU time that is consumed when profiling with `nios2_pcsample()`, multiply the length of time it takes to execute this function by the number of ticks.

To arrive at a total time for additional CPU cycles consumed by profiling, add the overhead for `mcount()` (one call to `mcount()` per profiled function invocation) to the overhead for `nios2_pcsample()`.

Hardware Considerations

The profiler just needs a system timer. No special peripherals are required. You do not need to change your Nios II hardware design.

Tutorial 1: Program the Standard Hardware Design to an FPGA

For the first tutorial, use the reference example standard hardware design without modification. If your Nios development board contains another hardware design, follow the next few steps to program the standard hardware design. If the Nios development board already has the standard hardware design programmed, go to "Create the Profiler_Project Software Design".

1. Run the Quartus® II software, version 5.1.
2. Open the Quartus II project file for the standard Nios II hardware design project for your board. For example, the Stratix® Edition standard project file name for the 1S40 device is **standard.qpf**, located in the directory *<Nios II kit path>***\examples\verilog\niosII_stratix_1s40\standard**.
3. On the Tools menu, click **Programmer**.
4. Turn on **Program/Configure,** located on the same row as **standard.sof**.
5. Click **Start** to download the Nios II SRAM Object File **standard.sof** to the FPGA.

If the **Start** button is greyed out, or the USB-Blaster™ cable is not listed, refer to the *Introduction to Quartus II* manual for more details on the Programmer tool.

Create the Profiler_Project Software Design

The following steps illustrate the creation of the profiler_project example as a Nios II IDE project.

1. Run the **Nios II IDE** software, version 5.1 from the Windows Start menu.

2. If the **Workspace Launcher** window opens, leave the default workspace, *<Nios II kit path>***bin\eclipse\workspace**, selected and click **OK**.

3. Create a new project.

   a. On the File menu, point to New and click **C/C++ Application**.

   b. In the Name field, type `profiler_project`.

   c. In the Select Target Hardware box, click **Browse** to set the SOPC Builder System. Select the PTF file to the standard hardware design for the Nios development board you are using. For example, the PTF file for the Stratix Edition standard Nios II hardware design for the 1S40 device is located at **<***Nios II kit path***>\examples\verilog\niosII_stratix_1s40\standard\std_1s40.ptf**.

   d. In the Select Project Template box, select **Blank Project**.

   e. Click **Finish**.

4. From Windows Explorer, drag the **profiler_project.c**, **checksum_test.c**, and **checksum_test.h** source files (included with this tutorial) into the **profiler_project** folder on the **C/C++ Projects** tab in the Nios II IDE.

5. On the **C/C++ Projects** tab, right-click **profiler_project** and click **Properties**.

6. In the Properties window, select **C/C++ Build** in the left column. In the **Active Configuration** box, select **Release**.

7. Click **OK**.

8. In the **C/C++ Projects** tab, right-click **profiler_project_syslib** and click **Properties**.

9. In the Properties window, select **System Library** in the left column. In the **System Library Contents** box, select **Link with profiling library**.

10. In the Properties window, select **C/C++ Build** in the left column. In the **Active Configuration** box, select **Release**.

11. Click **OK**.

You have created the profiler_project.

Create the Profiler Report Based on the Profiler_Project Design

After creating the project, follow these steps to run it and create the profiler report:

1. In the **C/C++ Projects** tab, right-click **profiler_project**. On the Run As menu, click **Nios II Hardware.** The build is performed automatically.

   The project execution causes a **gmon.out** file to be written that contains profiler information. The **gmon.out** file can be analyzed with the **nios2-elf-gprof** tool.

2. Run the Nios II SDK Shell and navigate to the software project directory, **profiler_project/Release**, as follows:

   $[SOPC Builder]$ `cd /cygdrive/c/altera/kits/nios2/examples/verilog/niosII_stratix_1s40/standard/software/profiler_project/Release`

3. Run **nios2-elf-gprof**, passing in the **profiler_project.elf** file and **gmon.out** profiler data file as follows:

   $[SOPC Builder]$ `nios2-elf-gprof profiler_project.elf gmon.out > report.txt`

   This action generates a flat profile report and a call graph, which are captured in the file **report.txt**.

4. Use any editor to view the **report.txt** file. The profiler report excerpts shown in the following section were generated on a Nios development board, Stratix Edition, containing a Stratix 1S40 device with a Nios II version 5.1 standard hardware design running at 50 MHz.

Analyze the Command Line Generated Profiler Report

Information in the profiler report is conveyed in the following two ways:

- The **flat profile** portion of the report identifies the child functions in the order in which they consume processing time.
- The **call graph** portion of the report describes the call tree of the program sorted by the total amount of time spent in each function and its children. Each entry in this table consists of several lines. The

line with the index number at the left hand margin lists the current function. The lines above it list the functions that called this function, and the lines below it list the functions this one called, with caveats that are detailed further in both the report itself and the full GNU profiler documentation.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 77.68      2.48     2.48        1     2.48     2.48  checksum_test
  9.08      2.77     0.29                             alt_dcache_flush
  1.14      3.16     0.04       10     0.00     0.00  alt_busy_sleep

... (deleted portion) ...

                Call graph (explanation follows)

granularity: each sample hit covers 32 byte(s) for 0.31% of 3.19 seconds

index % time    self  children    called     name
                2.48    0.00       1/1          main [3]
[4]     77.7    2.48    0.00       1          checksum_test [4]
                0.00    0.00     300/300         alt_dcache_flush_all [22]

... (deleted portion) ...

                0.00    0.00     321/321         alt_irq_handler [19]
[20]     0.0    0.00    0.00     321          alt_avalon_timer_sc_irq [20]
```

The call graph report shows that the `checksum_test` function call (index [4]) consumed 77.7% of the processing time during execution of the profiler_project design.

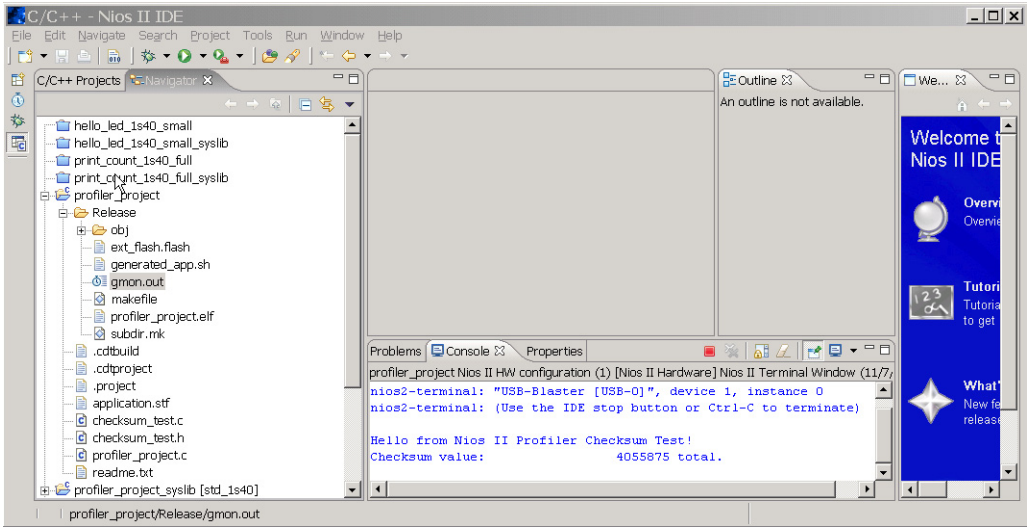
The granularity statement in the call graph report states that the report covers 3.19 seconds, or 3,190 milliseconds. Our Nios II system has a 10 millisecond timer, so the timer interrupt handler will be called once at the beginning before a full clock period has elapsed and once every 10 milliseconds thereafter. An accurate reporting should show, therefore, that the timer interrupt handler was called 320 times. Index [20] shows that `alt_avalon_timer_sc_irq` was called 321 times, which is within the sampling range.

Use the Nios II IDE Profiling Perspective

The **Profiling Perspective** provides several organizational views into the timing behavior of your application.

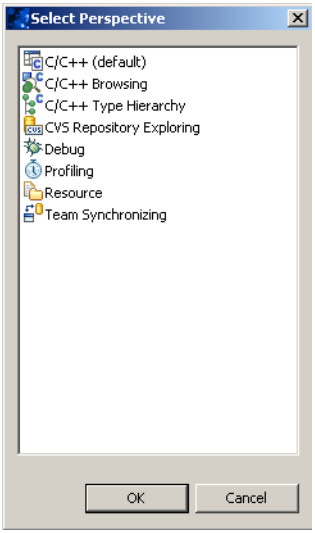
1. In the Nios II IDE C/C++ Perspective, click the **Navigator** tab.
2. In the **profiler_project\Release** directory, select the **gmon.out** file (Figure 1).

Figure 1. Select the gmon.out File



3. Right-click the **gmon.out** file. On the Open With menu, click **Gprof Viewer**.
4. If the Nios II IDE does not switch to the Profiling Perspective, on the Window menu, point to Open Perspective and click **Other**. Select **Profiling** and click **OK** (Figure 2).

Figure 2. Select Perspective Dialog Box

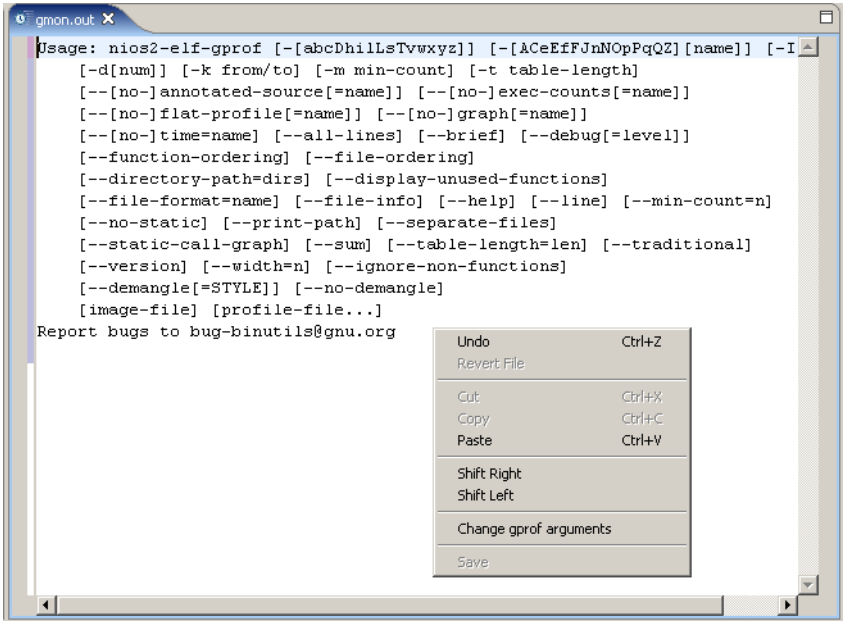


Editor View

When you open the **gmon.out** file with the Nios II IDE, the IDE automatically calls gprof and displays the standard gprof text output in the Editor view. By default, the output displayed in this window is identical to the **report.txt** file that you generated previously using the command line. You can apply gprof command line switches to the Editor view to modify the way profiling information is displayed. Use the **–help** parameter to show the complete list of options.

1. Right-click in the Editor view and click **Change gprof arguments**.
2. Type `--help` in the pop-up window to see all of the available options displayed in the Editor view (Figure 3).
3. Select **Change gprof arguments** again to remove the –help parameter. Gprof will regenerate the profiling information and display it in the Editor view.

Figure 3. Available Options in the Editor View



The Editor view shows all of the profiling information, just like the **report.txt** output. The **Profiling Perspective** provides two additional views, **Call Hierarchy** and **Samples**, that organize the profiling data in different ways to provide greater insight into the timing behavior.

Call Hierarchy View

The **Call Hierarchy** view displays the **gmon.out** call graph data in an easy-to-read tree format. In this view, you can follow the function call sequences more easily than by reading the **report.txt** file. There are two ways to view the call hierarchy data. Right-click in the view and click **Toggle Call Direction** to alternate between these display types:

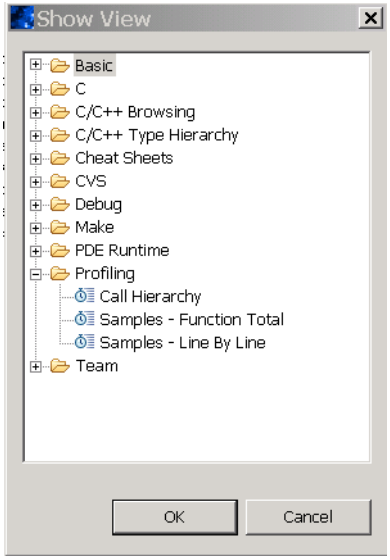
- **Top down** - lists the calling functions, with the functions they called nested below.
- **Inverted** - lists the called functions, with the functions that called them nested below.

When the call direction is **Top down**, the first entry is "spontaneous". Gprof uses this term when it cannot determine who the caller is. When the call direction is **Inverted**, the first entry is "inverted calls".

The **Call Hierarchy** view displays the time spent in each function from the standard **gmon.out** data. It also calculates and displays the percentage of time spent in each function. When the call direction is top down, there are two sets of actual time and percentage time figures for each function. The first set represents the time spent within the function. The second set represents the total time spent within the function plus all functions called by that function. Each indented line in the view drills down into the details of percentage time spent in each called function. When the call direction is inverted, only the first set of time and percentage numbers representing time spent within the function displays.

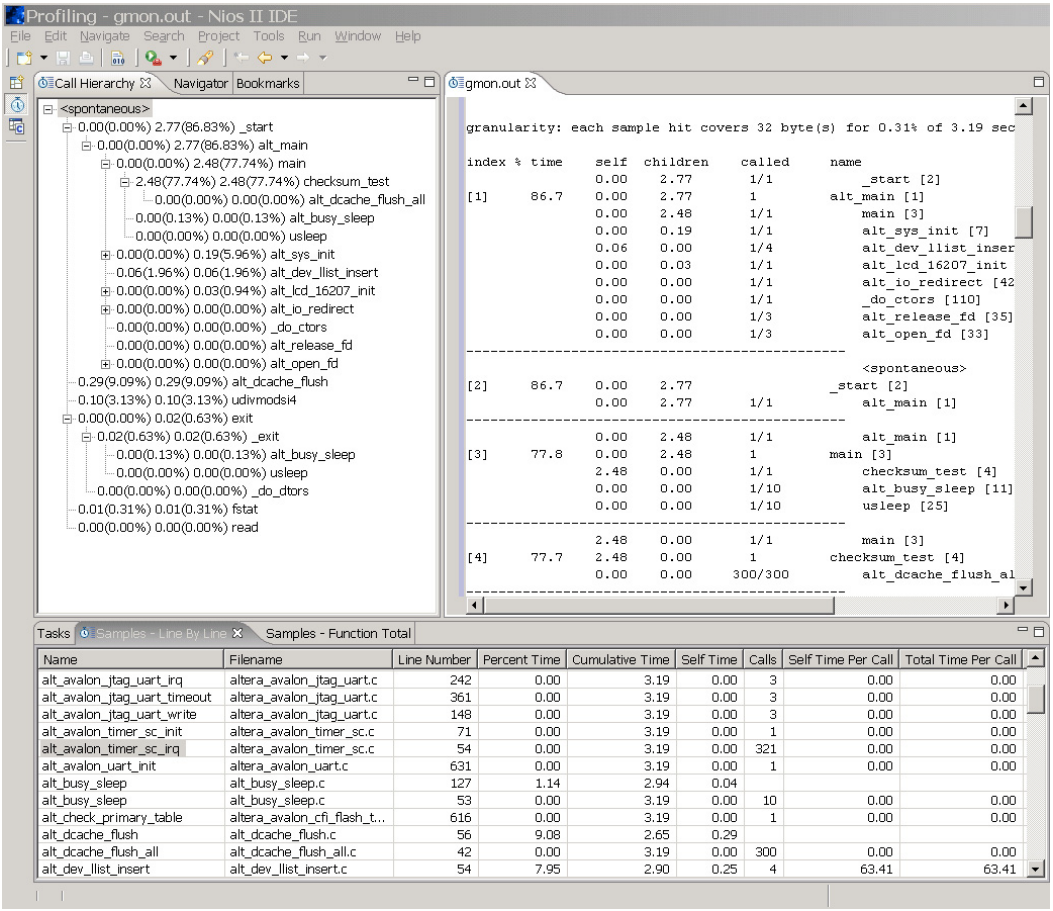
If the Nios II IDE does not show the **Call Hierarchy** view, on the Window menu, point to Show View, and click **Other**. Expand the **Profiling** folder and click **Call Hierarchy** (Figure 4).

Figure 4. Show View Dialog Box



The **Call Hierarchy** view shown in Figure 5 shows that `alt_avalon_timer_sc_irq()` is called by `alt_irq_handler()`. Additionally, this view shows that `alt_irq_handler()` is the *only* function that calls `alt_avalon_timer_sc_irq()`.

Figure 5. Different Views of Profiling Data



Samples – Line by Line View and Function Total

The **Samples – Line by Line** view (refer to Figure 5) breaks down the program execution by line of C source code executed. Each line of C source code sampled during profiling maps to one or more entries in the table.

Samples are taken of individual Nios II assembly instructions. The samples are collected into a fixed number of bins, regardless of program size. Each bin is shown as a single entry in the table. Therefore, the line by line samples view will show a higher degree of resolution for smaller

programs. For example, in a small program, every two assembly instructions might map to their own bin, while a larger program might map dozens of assembly instructions into a single bin.

Because a single C source line may get compiled into several assembler instructions, a single C source code line number could map to multiple entries in the table. In the profiler_project, the `checksum_test()` function in **checksum_test.c** has multiple table entries for C source code line number 83. The larger the program, the less likely the chance that a single C source code line will map to multiple table entries.

The **Samples – Function Total** view breaks down program execution by function. For example, to see the number of times the `alt_avalon_timer_sc_irq()` function is called, perform the following steps:

1. Click the **Samples – Function Total** tab.
2. Scroll down to the `alt_avalon_timer_sc_irq()` function.
3. In this example, the value for the **Calls** field for the `alt_avalon_timer_sc_irq()` function is 321. This value correlates with the granularity of 3.19 listed in the Flat Profile shown in the **Editor** view, because the 10 millisecond system timer frequency causes the timer interrupt to be invoked 100 times every second.

Click on a column heading to change the order in which the entries are sorted in that column. Click the heading again to reverse the sort order. Notice the '>' or '<' that precedes one of the column heading labels. The symbol indicates whether the samples are sorted in ascending or descending order. Their presence in a particular column label also identifies that the samples are sorted by that column.

Task View

During your profiling analysis, the **Task** view can be useful for recording notes about the timing behavior observed, such as the name of a function that is executing particularly slowly. Expending your programming efforts on improving this function may show a substantial increase in overall speed. You can use the **Task** view to record how changing system inputs impact the CPU consumption over successive executions.

Perform the following steps to use the **Task** view.

1. Right-click in the **Task** view and select **Add Task**.

2. Enter a description and select a priority for the new task.

3. Later, after you have improved the efficiency of the noted function, turn on the check box to indicate that this task has been completed.

Context Sensitive Help

The Nios II IDE Profiling Perspective offers context sensitive help that specifically details the various views. After clicking in the **Call Hierarchy** view or one of the Samples views, press **F1**. A pop-up window appears that provides a summary of the view as well as a link to the online help for that view.

Use Performance Counters and Timers

After the profiler has identified areas of code that consume a lot of CPU time, a performance counter or timer can further analyze these functional bottlenecks.

The following sections explain the advantages and limitations of using performance counters and timers for performance analysis. A tutorial is provided that demonstrates the use of performance counters and timers to collect and analyze performance data.

Performance Counter Advantage

There are no other mechanisms available with the Nios II development kits that provide measurements with so little intrusion. Only one or two instructions are required for each `BEGIN` and `END` macro. A performance counter is an order of magnitude faster than the profiler. The only way to get measurement data that is less intrusive would be a completely hardware-based solution, such as a logic analyzer set up with triggers on particular bus addresses.

Timer Advantage

Unlike the performance counter, which can only time up to seven sections of code simultaneously, there is no limit on the number of sections that can be measured with a timer. The timer can be read 1,000 times and stored in 1,000 different variables as a start time for a section, and then compared to 1,000 end timer readings. The only practical limiting factors are memory consumption and complexity.

Performance Counter Drawback

One drawback to measuring performance with a performance counter is the counter's large size. The performance counter consumes a large number of LEs on the FPGA. On a 1S40 device, a single performance

counter peripheral with three section counters defined within a modified standard hardware design consumes 670 logic cells (LCs), and 420 LC registers. The same design with a single performance counter defined with seven section counters consumes 1,345 logic cells and 808 LC registers.

A performance counter should be removed from a system before the system is deployed.

Timer Drawback

A timer consumes hardware resources. It also introduces an additional interrupt source into the system that impacts interrupt latency.

Performance Counter & Timer Drawbacks

A drawback to both performance counters and timers is the lack of context awareness. If a timer interrupt occurs during the measurement of a section of code, the total time taken by the CPU to process the timer interrupt and return to the section is added to the total measurement time. This effect is much more pronounced in a multi-threaded operating system. Many threads may get scheduled to execute while the section of code is being measured, resulting in a very large, skewed measurement time. To avoid thread switch impacts, most multi-threaded operating systems have a system call to temporarily lock the scheduler. Interrupts can be disabled to completely avoid section measurement interruptions. Of course, disabling interrupts or locking the scheduler usually affects the behavior of your system, so these actions should be avoided whenever possible.

Adding performance counters and timers can also increase $\mathbf{f_{MAX}}$.

Performance Counter Software Considerations

`PERF_BEGIN` and `PERF_END` are the performance counter peripheral macros that record the beginning and ending times of a particular code section to be measured.

`PERF_BEGIN` and `PERF_END` are single writes to the performance counter peripheral. These macros are very efficient, requiring only two or three machine instructions. This method provides the fastest way possible to record the time. The only way to make a measurement that is less intrusive would require the use of an external measurement device, such as a logic analyzer with triggers set on particular memory addresses.

The macros used to begin and end each performance counter section are defined as follows:

```
#define PERF_BEGIN(p,n) IOWR((p),(((n)*4)+1),0)

#define PERF_END(p,n) IOWR((p),(((n)*4) ),0)
```

The Global Counter

This unit uses section #0 as a special "global" section, which counts the total time during which measurements are being taken. None of the other section-counters are allowed to run at all (not even the other event counters) when the global time-counter is stopped. Special macros (`PERF_START_MEASURING`, `PERF_STOP_MEASURING`) are defined to control the global counters. Do not manipulate the global counters directly through `PERF_BEGIN` and `PERF_END`.

Hardware Considerations

Performance counters and timers are SOPC Builder peripherals, so adding one to an existing system necessitates a change to the Nios II-generated SOF in the Quartus II software. Timers and performance counters can, like any hardware counters, eventually overflow.

Tutorial 2: Use Performance Counters and Timers to Measure Code Performance

This tutorial demonstrates the use of performance counters and timestamp interval timers to further measure the performance of a Nios II system and pinpoint sections of code that use a lot of CPU time.

The software part of this second tutorial can be done without creating the standard_perf_counter Nios II hardware design. Instead, you can use the full_featured hardware reference design. Refer to "Appendix A: Full_Featured Reference Design" on page 25.

Create the standard_perf_counter Hardware Design

The following steps demonstrate the creation of the standard_perf_counter example as a Nios II hardware design.

1. Create a copy of the standard hardware design for a Nios development board. The copy will be modified to change the frequency of the interval timer and to add the performance counter.

   For example, the Stratix Edition standard reference design for the 1S40 device is located at *<Nios II kit path>***examples\verilog\niosII_stratix_1s40\standard**. Replacing the directory names for your Nios development board and hardware language type as appropriate, copy this directory to:

<Nios II kit path>\examples\verilog\niosII_stratix_1s40\standard_perf_counter

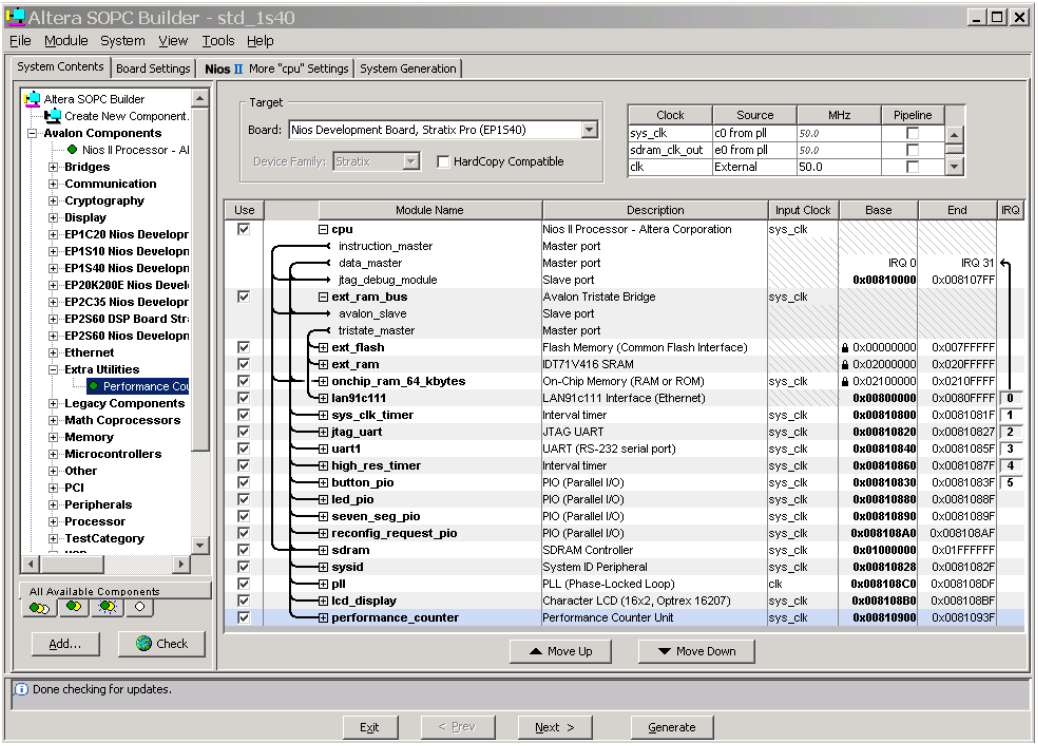
2. Run the Quartus II software, version 5.1.
3. Open the Quartus II project file for the new project in the folder you have just copied, **standard.qpf**.
4. On the Tools menu, click **SOPC Builder**.

   The SOPC Builder window appears.
5. In the **System Contents** tab, under the list of **Avalon Components**, click the '+' symbol to expand the choices under **Extra Utilities**.
6. Click **Performance Counter Unit**. Click **Add**.
7. Leave the default value of **Number of simultaneously-measured sections** at 3. Rename this instance of the performance counter to **performance_counter**. Click **Finish**.

   A performance counter module is added to the hardware design.
8. Under the list of Module Names that make up the hardware design, select the interval timer named **high_res_timer**.
9. Right-click **high_res_timer** and click **Edit**.
10. Under **Timeout Period**, leave the **Initial Period** value number set to 1, but change the units from **msec** (milliseconds) to **usec** (microseconds).
11. Click **Finish**.

    Figure 6 shows the SOPC Builder system.

Figure 6. SOPC Builder Window



12. Click the **System Generation** tab to generate the hardware design.

13. Click **Generate**. The generation phase will take a few minutes.

14. The last message should state "SUCCESS: SYSTEM GENERATION COMPLETED". When the system generation is complete, click **Exit**. The hardware design is now ready to be compiled by the Quartus II software.

15. In the Processing menu, click **Start Compilation**.

    When the compilation is complete, the Compilation Report provides full details. For example, the number of logic cells used to create the performance counter peripheral is reported in the **Resource Utilization by Entity** section of the Analysis & Synthesis folder.

Program the standard_perf_counter Hardware Design to an FPGA

Now you can program your new design into the FPGA.

1. On the Tools menu, click **Programmer**.
2. Turn on **Program/Configure**, located on the same row as **standard.sof**.
3. Click **Start** to program the **standard.sof** hardware design to the FPGA.

If the Start button is greyed out, or the USB-Blaster cable is not listed, refer to the *Introduction to Quartus II* manual for more details on the Programmer tool.

Create the Performance_Project Software Design

Create a software project to test the new hardware design.

1. On the Tools menu, click **SOPC Builder**.

   The SOPC Builder window appears.
2. Click the **System Generation** tab.
3. Click **Run Nios II IDE**.
4. Leave the default workspace, <*Nios II kit path*>**\bin\eclipse\workspace**, selected and click **OK**.
5. Close any projects that are open (except leave the Nios II Device Drivers project open).
6. Create a new project.
   a. On the File menu, point to New and click **C/C++ Application**.
   b. In the Name field, type `performance_project`.
   c. Verify that the SOPC Builder System file is the one specified for the standard_perf_counter hardware just created. For example, the PTF file for the 1S40 device is located at <*Nios II kit path*>**\examples\verilog\niosII_stratix_1s40\standard_perf_counter\std_1s40.ptf**.

d. In the Select Project Template box, select **Blank Project**.

e. Click **Finish**.

7. From Windows Explorer, copy the **high_res_timestamp_performance_project.c**, **checksum_test.c**, and **checksum_test.h** source files (included with this tutorial) into the **performance_project** folder on the **Navigator** tab in the Nios II IDE.

8. On the **C/C++ Projects** tab, right-click **performance_project** and click **Properties**.

9. In the Properties window, select **C/C++ Build** in the left column. In the **Active Configuration** box, select **Release**.

10. Click **OK**.

11. In the **C/C++ Projects** tab, right-click **performance_project_syslib** and click **Properties**.

12. In the Properties window, click **System Library.** From the menu for **Timestamp timer**, select **high_res_timer**.

13. In the Properties window, select **C/C++ Build** in the left column. In the **Active Configuration** box, select **Release**.

14. Click **OK**.

Build and Run the Performance_Project Software Design

1. In the **C/C++ Projects** tab, right-click **performance_project**. On the Run As menu, click **Nios II Hardware.** The build is performed automatically.

2. The performance counter report will be printed to STDOUT (JTAG UART). The following performance counter report was generated on a Nios 1S40 development board with Nios II version 5.1 standard hardware design running at 50 MHz with one performance counter added, and the **high_res_timer** modified to 1 microsecond.

```
Hello from Nios II Performance Checksum Test!
timestamp measurement for checksum_test = 128932865 ticks
timestamp measurement overhead = 73 ticks
Actual time in checksum_test = 128932792 ticks
Timestamp timer frequency = 50000000
--Performance Counter Report--
Total Time: 5.15913 seconds (257956281 clock-cycles)
+-----------------+--------+-----------+--------------+-----------+
| Section         |   %    | Time (sec)| Time (clocks)|Occurrences|
+-----------------+--------+-----------+--------------+-----------+
|1st checksum_test|      50|    2.57868|     128933869|          1|
+-----------------+--------+-----------+--------------+-----------+
|pc_overhead      |6.98e-06|    0.00000|            18|          1|
+-----------------+--------+-----------+--------------+-----------+
|ts_overhead      |2.79e-05|    0.00000|            72|          1|
+-----------------+--------+-----------+--------------+-----------+
Goodbye from Nios II - returning from main()!
```

`pc_overhead` is the performance counter peripheral overhead of a single invocation to `BEGIN MACRO` for that peripheral. This number is inclusive of the `BEGIN` and `END MACRO` pair invocation used to take the measurement with a performance counter.

`ts_overhead` is the timestamp overhead of a single function call to read the timer. This number is inclusive of the performance counter overhead used to measure the timestamp overhead.

Conclusion

The Nios II development environment provides a variety of ways to analyze the performance of your project. Depending on your needs, you can take the software-only GNU profiler approach. For the more deterministic real-time performance issues, you can leverage a hardware timer or performance counter. The wide range of tools available means that time should be taken to consider the class of problem that you are trying to solve in order to choose the best tool for the job.

Troubleshooting

The following sections describe several problems that might occur, and suggest ways to deal with them.

nios2-elf-gprof –annotated-source Switch Has No Effect

`basic-block-count` information is not tracked, so switches such as `–annotated-source` will not work.

Writing to the Registers of a Non-Existent Section Counter

The following performance counter report shows the results of an attempt to use a non-existent section counter of the performance counter peripheral.

Suppose that a fourth section counter is specified for a performance counter peripheral that has been defined in SOPC Builder to have only three section counters (the default value).

In this case, the test was performed on a hardware design that did not have any other peripheral defined with registers mapped immediately after the performance counter peripheral's registers, so no other peripheral was impacted. Depending on how the peripheral register base addresses have been configured in SOPC Builder for a particular hardware design, unpredictable system behavior could occur.

```
--Performance Counter Report--
Total Time: 5.78751 seconds (289375582 clock-cycles)
+--------------------+--------+-------------+---------------+-----------+
|      Section       |   %    | Time (sec)  | Time (clocks) |Occurrences|
+--------------------+--------+-------------+---------------+-----------+
|sleep_tests         |    49.4|      2.86162|      143081026|          1|
+--------------------+--------+-------------+---------------+-----------+
|perf_begin_overhead | 7.6e-06|      0.00000|             22|          1|
+--------------------+--------+-------------+---------------+-----------+
|timestamp_overhead  | 7.6e-06|      0.00000|             22|          1|
+--------------------+--------+-------------+---------------+-----------+
|non_existent_counter|6.37e+12|368934881474.19104|        -1| 4294967295|
+--------------------+--------+-------------+---------------+-----------+
```

Output From a printf() or perf_print_formatted_output() Call Near the End of main() May Get Prematurely Truncated

This occurs when the Nios II application executes a `BREAK` instruction to transfer profiling data to the development workstation during the `exit()` or `return()` from `main()`.

As a workaround, call `usleep(500000)`. This action allows enough of a delay for the I/O to be transmitted over the JTAG UART before `main` returns (or calls `exit()`). If the output is still partially truncated, increase the delay value passed into `usleep()`. Use `#include <unistd.h>` for the `usleep()` function prototype.

Fitting a Performance Counter Into a Hardware Design That Consumes Most of an FPGA's Resources

The system could be measured in a larger FPGA for development than the size of the FPGA in a deployed system.

Configure a performance counter to have only one section counter to save the most resources.

The Histogram for the gmon.out File Is Missing, Even Though My main() Function Terminates

If no system timer is defined for the system, the `nios2_pcsample()` function will never get called, and the histogram for the **gmon.out** file will not be produced. Define a system timer on the system properties page in the Nios II IDE.

Further Reading

For information on the GNU profiler, gprof, refer to **c:\Altera\kits\nios2\documents\gnu-tools\binutils\gprof.html**. Altera has rewritten the lib-gprof library, so the information in this manual on how data is collected doesn't match Altera's implementation.

For information on the Nios II IDE Profiling Perspective views, use the Nios II IDE Help facility, and search for the word "Profiler".

For information on the performance counter, refer to **c:\Altera\kits\nios2\documents\performance_counter_readme.html**.

For information on the high-speed timer, refer to the *Timer Core with Avalon Interface* chapter in the *Altera Embedded Peripherals Handbook*.

Appendix A: Full_Featured Reference Design

This section demonstrates execution of the performance_project on the full_featured Nios II hardware design.

To open the project file, perform the following steps:

1. Run the Quartus II software, version 5.1.
2. Open the Quartus II project file for the full_featured Nios II hardware design project for your board. For example, the Stratix Edition full_featured project file name for the 1S40 device is **full_featured.qpf**, located in the directory *<Nios II kit path>*\ **examples\verilog\niosII_stratix_1s40\full_featured**.

Program the Full_Featured Hardware Design to an FPGA

Now you can program your new design into the FPGA.

1. On the Tools menu, click **Programmer**.
2. Turn on **Program/Configure**, located on the same row as **full_featured.sof**.
3. Click **Start** to program the **full_featured.sof** hardware design to the FPGA.

Create the Performance_Project Software Design

Create a software project to test the full hardware full_featured design.

1. On the Tools menu, click **SOPC Builder**.

   The SOPC Builder window appears.
2. Click the **System Generation** tab.
3. Click **Run Nios II IDE**.
4. Leave the default workspace, <*Nios II kit path*>**\bin\eclipse\workspace**, selected and click **OK**.
5. Close any projects that are open (except leave the Nios II Device Drivers project open).
6. Create a new project.
   a. On the File menu, point to New and click **C/C++ Application**.
   b. In the Name field, type `performance_project`.
   c. Verify that the SOPC Builder System file is the one specified for the full_featured hardware. For example, the PTF file for the 1S40 device is located at <*Nios II kit path*>**\examples\verilog\niosII_stratix_1s40\full_featured\full_1s40.ptf**.
   d. In the Select Project Template box, select **Blank Project**.
   e. Click **Finish**.

7. From Windows Explorer, copy the **high_res_timestamp_performance_project.c**, **checksum_test.c**, and **checksum_test.h** source files (included with this tutorial) into the **performance_project** folder on the **Navigator** tab in the Nios II IDE.

8. On the **C/C++ Projects** tab, right-click **performance_project** and click **Properties**.

9. In the Properties window, select **C/C++ Build** in the left column. In the **Active Configuration** box, select **Release**.

10. Click **OK**.

11. In the **C/C++ Projects** tab, right-click **performance_project_syslib** and click **Properties**.

12. In the Properties window, click **System Library.** From the menu for **Timestamp timer**, select **high_res_timer**.

13. In the Properties window, select **C/C++ Build** in the left column. In the **Active Configuration** box, select **Release**.

14. Click **OK**.

Build and Run the Performance_Project Software Design

1. In the **C/C++ Projects** tab, right-click **performance_project**. On the Run As menu, click **Nios II Hardware.** The build is performed automatically.

2. The performance counter report will be printed to STDOUT (JTAG UART). The following performance counter report was generated on a Nios 1S40 development board with Nios II version 5.1 full_featured hardware design running at 50 MHz.

```
Hello from Nios II Performance Checksum Test!
timestamp measurement for checksum_test = 51447341 ticks
timestamp measurement overhead = 49 ticks
Actual time in checksum_test = 51447292 ticks
Timestamp timer frequency = 50000000
--Performance Counter Report--
Total Time: 2.0589 seconds  (102944904 clock-cycles)
+-----------------+--------+-----------+--------------+-----------+
| Section         |   %    | Time (sec)| Time (clocks)|Occurrences|
+-----------------+--------+-----------+--------------+-----------+
|1st checksum_test|      50|    1.02897|      51448545|          1|
+-----------------+--------+-----------+--------------+-----------+
|pc_overhead      |1.75e-05|    0.00000|            18|          1|
+-----------------+--------+-----------+--------------+-----------+
|ts_overhead      |4.47e-05|    0.00000|            46|          1|
+-----------------+--------+-----------+--------------+-----------+
Goodbye from Nios II - returning from main()!
```

`pc_overhead` is the performance counter peripheral overhead of a single invocation to `BEGIN MACRO` for that peripheral. This number is inclusive of the `BEGIN` and `END MACRO` pair invocation used to take the measurement with a performance counter.

`ts_overhead` is the timestamp overhead of a single function call to read the timer. This number is inclusive of the performance counter overhead used to measure the timestamp overhead.

101 Innovation Drive
San Jose, CA 95134
(408) 544-7000
www.altera.com
Applications Hotline:
(800) 800-EPLD
Literature Services:
literature@altera.com