EDK Profiling User Guide

A Guide to Profiling in EDK

UG448 (v14.1) April 24, 2012

XILINX®

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
09/05/2007	1.0	Initial Xilinx Release for EDK 9.2i.
01/14/2008	2.0	EDK 10.1 release.
04/15/2009	3.0	EDK 11.1 release.
12/02/2009	3.1	EDK 11.2 release.
04/19/2010	4.0	EDK 12.1 release.
10/17/2011	13.3	EDK 13.3 release. Note version format change to match release number. Updated terminology and figures to match current software.
01/18/2012	13.4	EDK 13.4 release. • Added links to supporting documents UG667 and UG668 on page 5. • Removed information about compiling the BSP with the -pg option. This step is not necessary with this version of the tools.
04/24/2012	14.1	EDK 14.1 release. • Updated Setting Up the Hardware for Profiling, page 7. • Removed “Enabling the Profiling Timer in Your Application Code;” this is no longer necessary. • Zynq™ device documentation added to Appendix B, Additional Resources.

Table of Contents

XILINX®

Chapter 1

Introduction

The Xilinx® Embedded Development Kit (EDK) is a suite of tools and IP that enables you to design a complete embedded processor system for implementation in a Xilinx Field Programmable Gate Array (FPGA) device.

This user guide provides information about profiling software running on embedded systems built with EDK. Profiling is software-intrusive, and is based on the GNU `gprof` tool. This document details how profiling works, how to set up the hardware and software systems to perform profiling, and how to view the resulting profile data.

For information about creating a hardware platform and software application, refer to the following documentation:

- *Getting Started with the Spartan®-6 FPGA SP605 Embedded Kit* (UG667)
- *Getting Started with the Virtex®-6 FPGA ML-605 Embedded Kit* (UG668)

Profiling with GNU gprof

Profiling a program with GNU `gprof` provides two kinds of information that you can use to optimize the program:

- A histogram with which you can identify the functions in the program that take up the most execution time
- A call graph that shows what functions called which other functions, and how many times

For additional information about GNU `gprof`, refer to http://sourceware.org/binutils/docs-2.18/gprof/index.html.

How Profiling Works

The execution flow of the program is altered to obtain the data needed for `gprof`. Consequently, this method of profiling is considered "software-intrusive."

The program flow is altered in two ways:

- To obtain histogram data, the program is periodically interrupted to obtain a sample of its program counter location. This user-defined interval is usually measured in milliseconds. The program counter location helps identify which function was being executed at that particular sample. Taking multiple samples over a long interval of a few seconds helps identify which functions execute for the longest time in the program.
- To obtain the call graph information, the compiler annotates every function call to store the caller and callee information in a data structure.

The steps involved in profiling are as follows:

1. Compile and link the program for profiling by adding the `-pg` switch to the `gcc` compiler command line.
2. Run the program to generate profile data.
3. Process the profile data obtained from `gprof`.

These steps are explained in more detail in Chapter 2, "Using SDK for Profiling in EDK."

XILINX®

Chapter 2

Using SDK for Profiling in EDK

This chapter explains the steps involved in profiling an application in EDK. The following sections are included:

- Setting Up the Hardware for Profiling
- Building Applications with Profile Information
- Generating Profile Data

Setting Up the Hardware for Profiling

To profile a software application, you must have a timer device present and accessible from the processor that is running the program. The profiling process periodically interrupts the processor to identify what section of the code is running.

Xilinx profiling libraries that provide the profile interrupt handler support the xps/axi_timer core. When you run profiling on PowerPC® processors, you can also use the internal Programmable Interrupt Timer (PIT). Either the xps/axi_timer, SCU timer, or PIT must be available for exclusive use by the profile libraries. The timer interrupt signal is connected to the processor either directly or through an interrupt controller.

Note: The SCU timer is the only timer supported for profiling on Zynq™ devices.

Building Applications with Profile Information

Building an application for profiling involves the following steps:

1. Enable software intrusive profiling in the board support package (BSP).
   Refer to Enabling Profiling in the BSP, page 7.
2. Compile the application with the `-pg` flag.
   Refer to Compiling the Application with the -pg Option, page 9.

Enabling Profiling in the BSP

The first step in building an application for profiling is to enable the profile libraries in the board support package (BSP). Only applications built using the standalone BSP can be profiled. To enable profiling libraries in this BSP:

1. In SDK, select **Xilinx Tools > Board Support Package Settings** to open the Board Support Package Settings dialog box.
2. Select the BSP to configure and click **OK**.
3. Click **standalone** to configure the parameters for the standalone OS.
4. For the `enable_sw_intrusive_profiling` option, set the **Value** column to **true**.

5. For MicroBlaze™ processor based designs, you must also specify the timer to be used for profiling.

Refer to Figure 2-1, which displays a correctly configured BSP.

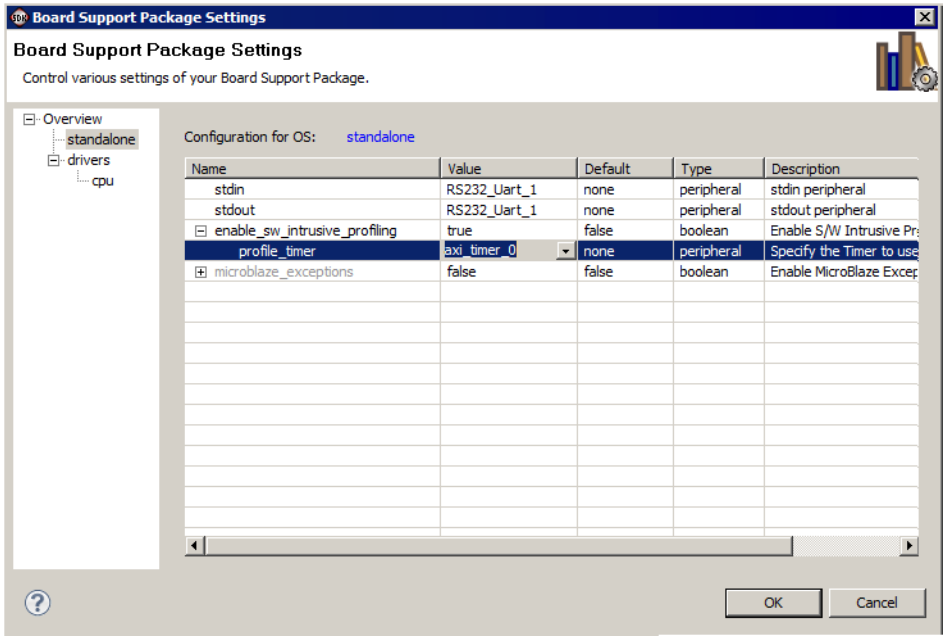

Figure 2-1: **BSP with Profiling Enabled**

6. On the **drivers > cpu** configuration page, add `-pg` to `extra_compiler_flags`.

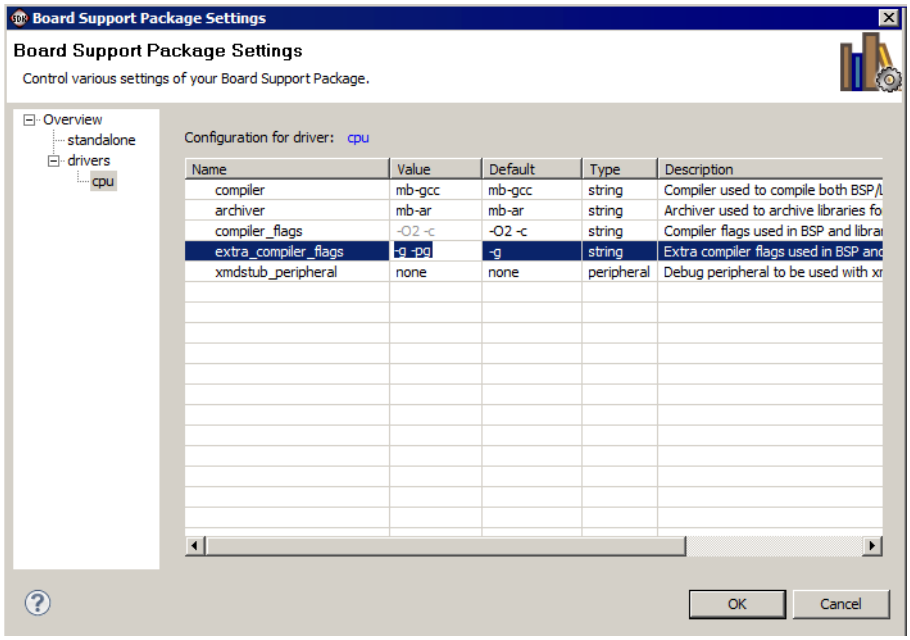

Figure 2-2: **cpu Configuration Page**

Compiling the Application with the -pg Option

Finally, you must compile the application with the `-pg` flag enabled. To do this:

1. In the Project Explorer, right-click the application and select **Properties**.
2. In the Properties dialog box for the application, open the **Settings** page for the C/C++ build.
3. In the Tool Settings tab, select the **Profiling** category for the compiler.
4. Click to select the **Enable Profiling (-pg)** check box.

Refer to Figure 2-3, which displays the Properties dialog box and the related settings.

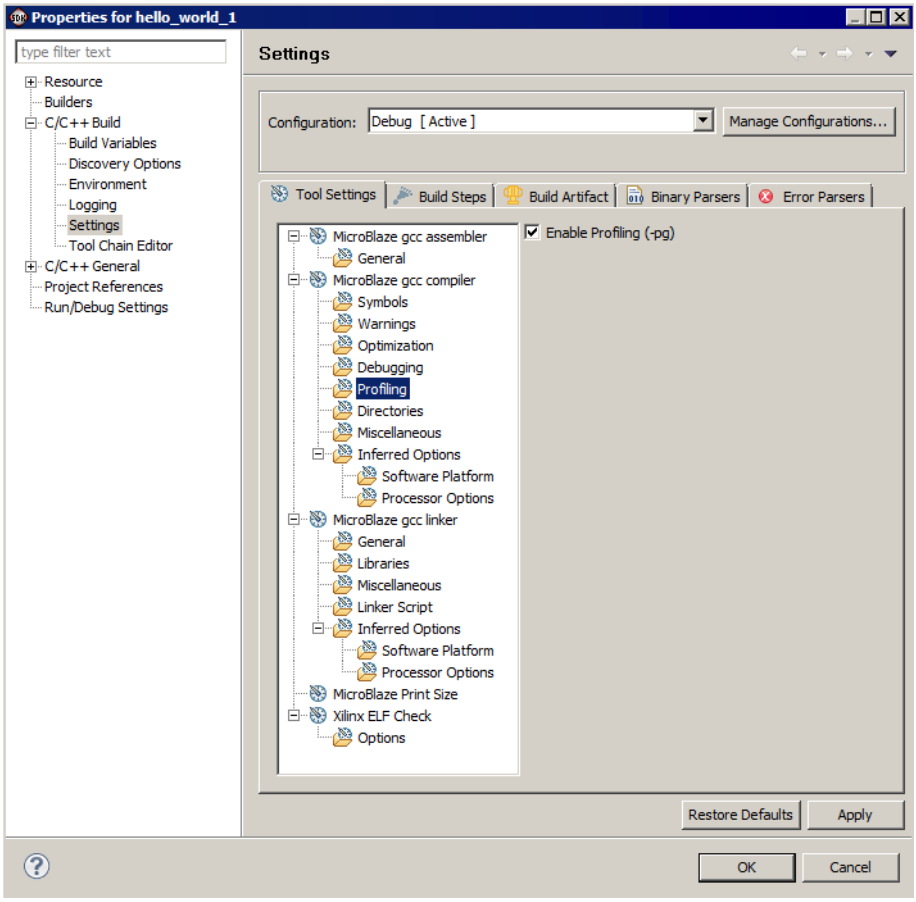

Figure 2-3: **Properties Dialog Box with Profiling Enabled**

Generating Profile Data

After compiling the application for profiling, you must run it once to obtain profile data. Start by creating a new run configuration in SDK:

1. In SDK, select **Run > Run** to open the Run dialog box.
2. In the Profiler tab:
   a. Select the **Enable Profiling** check box.
   b. Type values for the three profiling parameters. These are described in more detail in the following sections.
3. Save this configuration and run the application using the profile you created.

Figure 2-4 displays a Run Configuration with profiling enabled. The following sections describe the attributes shown here: Sampling Frequency, Bin Size, and Profile Memory.

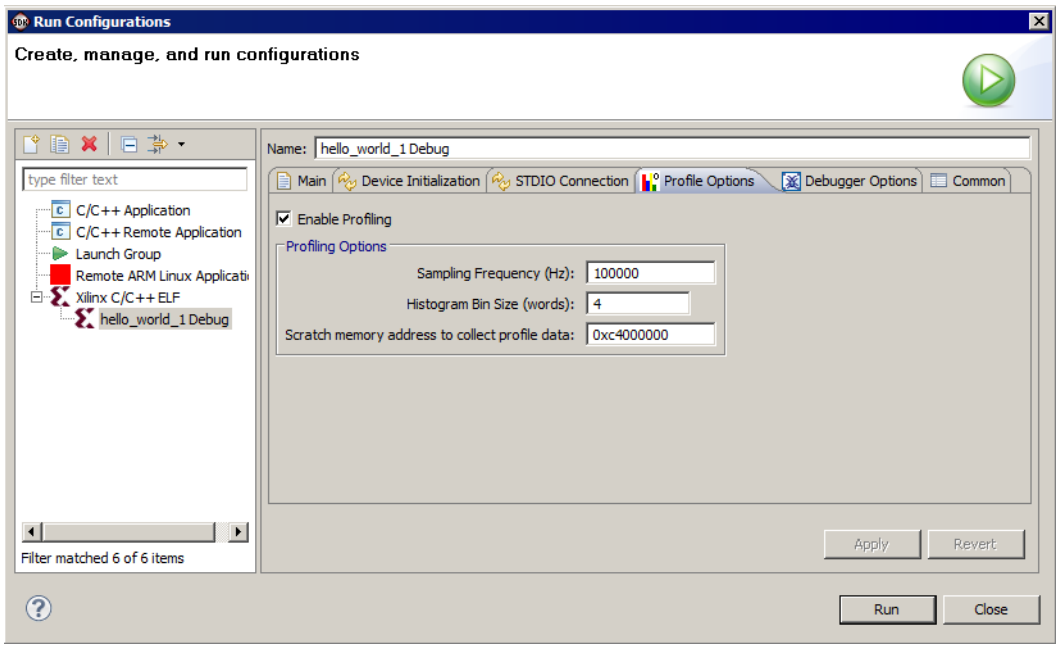

Figure 2-4: **Run Configuration with Profiling Enabled**

Sampling Frequency

The sampling frequency determines the frequency at which timer interrupts are generated. When you set a higher frequency, more samples are obtained. This provides more accuracy but is highly software-intrusive because of the number of interrupts. More calls are inserted to collect data.

Bin Size

The program text region is divided into multiple bins. When a program is interrupted because of the sampling frequency, the bin size determines how accurate the PC location is in the sample.

When you set a smaller bin size, the program text region is divided into a large number of small bins. This allows a more accurate sample because profile data can be attributed to a specific area of the text region. For example, if you set the bin size to 4 bytes, you can narrow down the specific instruction at which the program execution occurred to four bytes of the text region. The disadvantage to using a smaller bin size is that it requires a large number of bins to cover the entire text region, so a large amount of memory space is required for storing profile data.

When you set a larger bin size, the program text region is divided into a small number of large bins. This requires less memory space for storing profile data. However, it is much more difficult to identify specific text regions for the sample because of the larger bin size. For example, if you set the bin size to 40 bytes, you can only determine that the program was executing instructions between x and x+40 on each profile interrupt.

Profile Memory

The profile memory parameter indicates where in memory the profile data must be stored. This memory needs to lie outside the program memory area (including the text, data, heap and stack) and should not be overwritten.

Viewing Profile Data

Once the program reaches exit and completes execution or when you click the **Stop** button to stop the program, SDK automatically downloads the profile data and stores it in a file called `gmon.out`. The Profiling Results Saved dialog box displays this information.

The `gmon.out` file also becomes visible in the project explorer view, in the `Debug` or `Release` folder. To view the profile results, double-click the `gmon.out` file. SDK opens a dialog box requesting that you specify the ELF file associated with the profile information. By default, the ELF file points to the correct ELF file used to generate the profile information.

Click **OK** to view the profile results. The results open in a separate view at the bottom of the gprof tab, as shown in Figure 2-5. The results view comprises the following columns:

- **Name (location)** indicates the name of the function and the file in which it resides.
- **Samples** indicates the number of times the profile timer interrupt handler noticed that the program was currently executing the corresponding function.
- **Calls** indicates the number of calls made to the corresponding function.
- **Time/Call** indicates the time per function call invocation.
- **% Time** indicates the time spent executing this function as a percentage of total execution time.

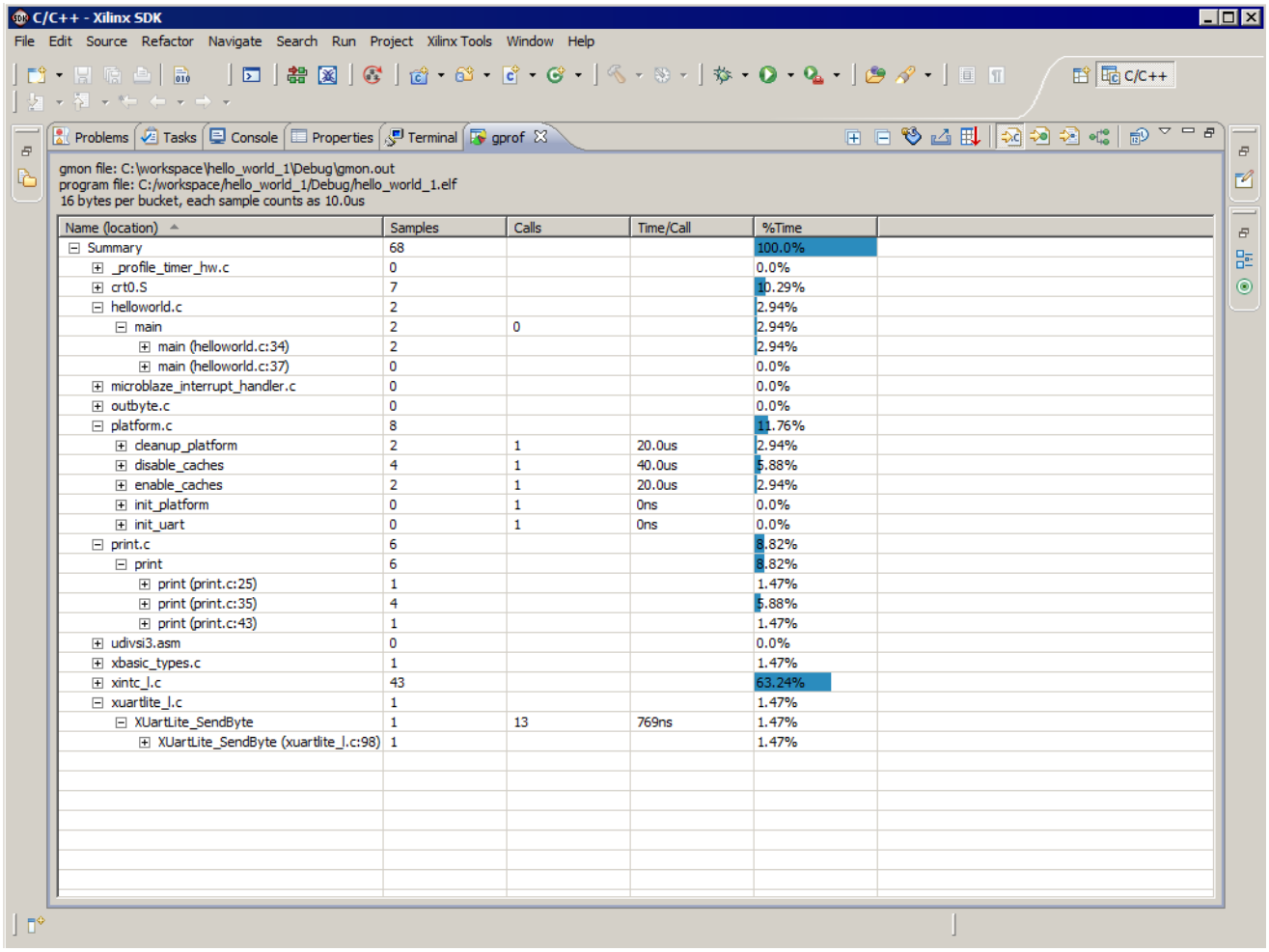

Figure 2-5: **Profiling Results View**

XILINX®

Appendix A

Profiling Restrictions

The following restrictions apply when profiling in EDK:

- Profiling does not measure the time spent in interrupt handlers because interrupt handlers typically disable further interrupts from occurring. Therefore, it is impossible for profiling interrupts to occur when the program is executing an interrupt handler.
- Profiling can only be done with the standalone platform; it cannot be done in the presence of an OS. This is because the profiling libraries are only available in the standalone BSP.
- Recursive functions are not supported.
- The call graph for functions inside C and Math libraries (`libc` and `libm`) are not generated because these libraries are not compiled with the `-pg` compiler profiling option.
- Ensure that memory used for collecting profile data is not used by any other function in the application.
- If you are using a custom linker script for a PowerPC® processor, it must include a `.vectors` section. This is because profiling is based on interrupts, and using interrupts requires a `.vectors` section.
- Profiling cannot be done while debugging. Enable profiling only when selecting the **Run** configuration in SDK.

XILINX®

Appendix B

Additional Resources

Xilinx Resources

- **Xilinx® Device User Guides**: http://www.xilinx.com/support/documentation/user_guides.htm
- **Glossary of Terms**: http://www.xilinx.com/company/terms.htm
- ***Xilinx Design Tools: Installation and Licensing Guide*** *(UG798)*: http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_1/iil.pdf
- ***Xilinx Design Tools: Release Notes Guide*** *(UG631)*: http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_1/irn.pdf
- **Product Support and Documentation:** http://www.xilinx.com/support

EDK Documentation

You can also access the entire documentation set online at: http://www.xilinx.com/support/documentation/dt_edk_edk14-1.htm

- ***EDK Concepts, Tools, and Techniques*** *(UG683)*: http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_1/edk_ctt.pdf
- ***Embedded System Tools Reference Manual*** *(UG111)*: http://www.xilinx.com/support/documentation/xilinx14_1/est_rm.pdf
- **MicroBlaze™ Processor User Guide** (UG081): http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_1/mb_ref_guide.pdf
- ***Platform Specification Format Reference Manual*** *(UG642)*: http://www.xilinx.com/support/documentation/xilinx14_1/psf_rm.pdf
- ***PowerPC 405 Processor Block Reference Guide*** *(UG018)*: http://www.xilinx.com/support/documentation/user_guides/ug018.pdf
- ***PowerPC 405 Processor Reference Guide*** *(UG011)*: http://www.xilinx.com/support/documentation/user_guides/ug011.pdf
- ***PowerPC 440 Embedded Processor Block in Virtex®-5 FPGAs*** *(UG200)*: http://www.xilinx.com/support/documentation/user_guides/ug200.pdf
- ***Zynq™ Concepts, Tools, and Techniques*** *(UG873)*: http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_1/ug873_zynq_ctt.pdf
- ***Zynq-7000 Software Developers Guide*** *(UG821)*: http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_1/ug821-zynq-7000-swdev.pdf

EDK Additional Resources

- **EDK Tutorials website**:
  http://www.xilinx.com/support/documentation/dt_edk_edk14-1_tutorials.htm
- **Platform Studio and EDK website**:
  http://www.xilinx.com/ise/embedded_design_prod/platform_studio.htm
- **XPS/EDK Supported IP website**:
  http://www.xilinx.com/ise/embedded/edk_ip.htm