SDK Tool Profiling

Zynq All Programmable SoC ZC702 or ZedBoard

2016.3

Abstract

This lab demonstrates how to profile a program, interpret reports, and verify performance with multiple calls. You will use profiling information to determine which tasks are consuming the most amount of time. The instructions will guide you through profiling a program and analyzing the results.

Objectives

After completing this lab, you will be able to:

- Profile an application by using the SDK tool
- Profile code and determine the most time-consuming functions via the SDK tool
- Thoroughly interpret the profiling results

Introduction

Profiling is a method by which the software execution time of each routine is determined. Routines that are frequently called are best suited for placement in fast memories; cache being one candidate.

TCF Profiler is a non-intrusive profiler based on PC sampling. An application is run that collects the list of PC recordings and maps them to the locations of subroutines. The number of occurrences of all of these subroutines is summed and reported to the user.

The following is an example of a non-intrusive profiler.

Given a main and a number of functions, TCF Profiler samples the functions using the PC.

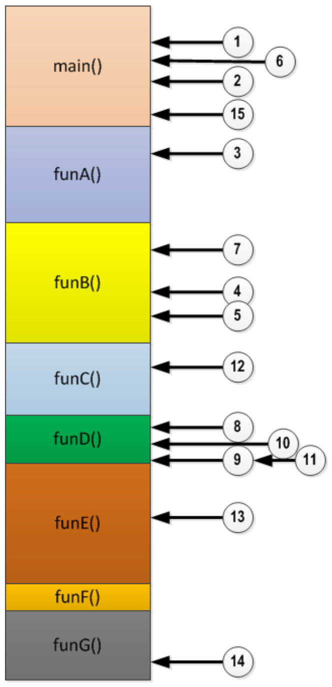


Figure 11-1: Example of Profiling Interrupts Against a Memory Map

After a sufficient amount of time passes and a statistically significant number of samples is collected (15 samples stated in this example is NOT enough; this is just an example), the data is extracted from the system memory.

function	freq
main()	4
funA()	1
funB()	3
funC()	1
funD()	4
funE()	1
funF()	0
funG()	1

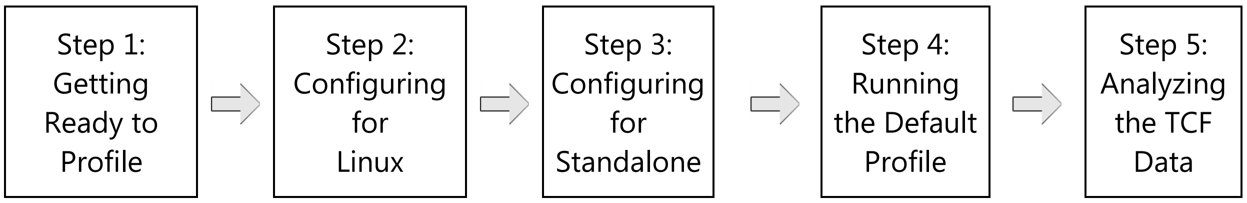
Figure 11-2: Frequency Table for Above Example

This information can be represented graphically, in terms of time, etc. One should note a few things from this example. First, funF() was never called. Really? Or were too few samples captured? Perhaps the sample interval was set too long and funF() executes very quickly and was simply never captured there?

The profiling tools enable you to set the granularity of the timer; but be careful, too fine a granularity might impact your code in other ways. If this is rerun, it is entirely possible that a slightly different set of results may be generated. This becomes a probabilistic analysis.

Neither profiling technique is perfect, but they do not need to be. Remember that profiling will give you a *feel* for how your code is executing, not a perfect representation.

General Flow


Step 1: Getting Ready to Profile
Step 2: Configuring for Linux
Step 3: Configuring for Standalone
Step 4: Running the Default Profile
Step 5: Analyzing the TCF Data

Getting Ready to Profile Step 1

You will begin by creating an SDK tool software application. The provided C source code will be used as the target for profiling. The software application will compile and link without errors.

You will use *C:\training\SDK_Profiling\lab* as the SDK tool workspace. The SDK development environment will automatically create a hardware platform depending on the development board that is selected.

Once the SDK tool is launched and a workspace is set up in this directory, you will not be able to move these directories.

1-1. Launch the SDK tool and set the workspace.

1-1-1. Select **Start** > **All Programs** > **Xilinx Design Tools** > **SDK 2016.3** > **Xilinx SDK 2016.3** to launch the tool.

Alternatively, you can launch the tool from its desktop shortcut, if available.

The Workspace Launcher opens after a moment.

The SDK tool creates a workspace environment that initially only contains a thin structure that tracks tool settings and maintains the SDK tool log file. In SDK, as projects are added, this workspace will update to include hardware projects, BSPs, and your software applications. Workspaces can be switched from within the SDK tool (select **File** > **Switch Workspace**).

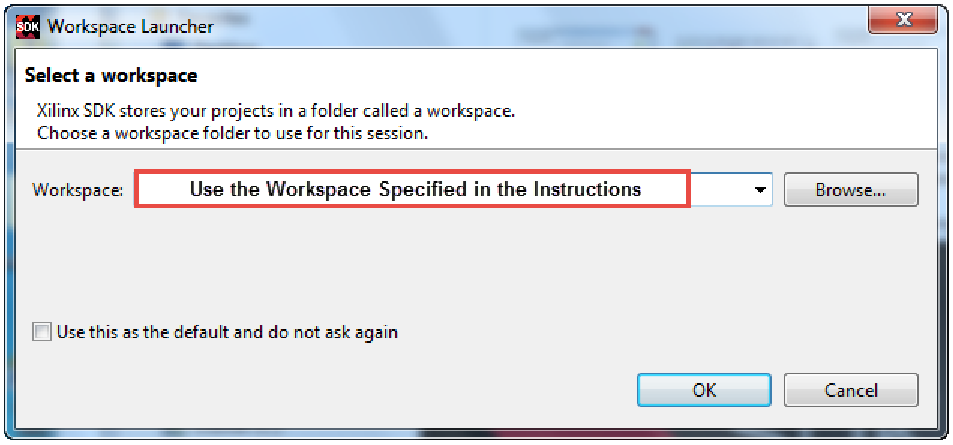
If it becomes necessary to move a software application to another location or computer, use the import and export features. Manually copying files is not recommended as workspace files are set to use absolute path names and this will cause the tool to become unstable.

The default location for the SDK software workspace (when launching from within the Vivado® Design Suite) is the root directory of your hardware project; however, a long path name can lead to problems on Windows-based machines. There is no default location for the tool projects. Placing your project at the root level or one hierarchical level below helps keep the path names as short as possible and is recommended.

Many of the Xilinx labs do not follow this guidance as it is important to keep a predictable structure through the various courses and labs. These labs have been tested to ensure that path name lengths do not cause problems.

1-1-2. Enter **C:\training\SDK_Profiling\lab** into the Workspace field or use the Browse button when the Workspace Launcher opens.

Note that when you use the Browse button, you will need to select the **C:\training\SDK_Profiling\lab** directory and click **OK**.


Workspace Launcher

Select a workspace

Xilinx SDK stores your projects in a folder called a workspace.
Choose a workspace folder to use for this session.

Workspace: Use the Workspace Specified in the Instructions

Figure 11-3: Setting Up the Workspace Environment Path

1-1-3. Click **OK** to close the Workspace Launcher dialog box and open the new workspace.

A workspace location and hardware platform are created when the **Export Hardware Design for SDK** command is performed from the Vivado Design Suite (or they can be created manually). While not a requirement, it is a good idea to keep the related files together.

Note that SDK must associate with a hardware system that has been previously exported so that an appropriate software platform or board support package can be built. However, the SDSoC™ development environment can take advantage of available platforms (for ZC702/ZedBoard). The hardware platform can be created for your custom hardware.

Usually, a platform provider builds the platform hardware using the Vivado Design Suite and IP integrator. For more information on platform creation, refer to the "SDSoC Platform Creation" topic cluster.

When the SDK tool is launched on its own, you must manually identify where you want the workspace and create (or import) the necessary hardware description to begin developing an application.

1-1-4. Close the **Welcome** tab if it appears.

This will give you more room to view your project. You may also want to maximize the SDK window, as there will be a lot to see.

If you are working in the SDK tool and do not have a hardware project and BSP specified, select the BSP and hardware project during the import process. SDSoC tool projects do not require a hardware platform nor a BSP.

One or more projects can be exported as a single zipped file. This is convenient when passing projects or parts of projects to teammates or clients. Once the recipient has received this archive, the zip file must be processed and one or more projects can then be imported.

1-2. Import an existing project.

1-2-1. Select **File** > **Import** to open the Import Wizard.

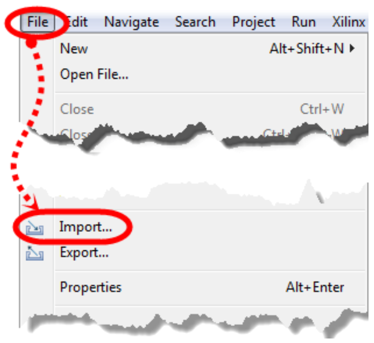
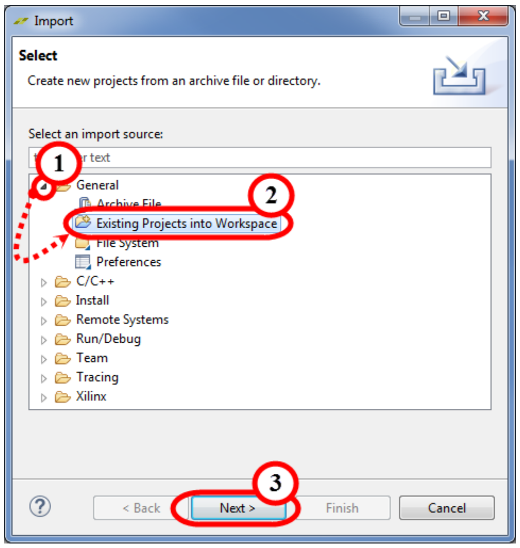


Figure 11-4: Accessing the Import Wizard

The Import dialog box opens.

1-2-2. Expand the **General** node to access the commonly used methods (1).

1-2-3. Select **Existing Projects into Workspace** as the goal is to import an existing project (2).


Import

Select

Create new projects from an archive file or directory.

Select an import source:

Figure 11-5: Choosing to Import an Existing Project into the Workspace

1-2-4. Click **Next** to enter project-specific data (3).

1-2-5. Select the **Select archive file** option (1).

Note that projects can be archived either as single zip files, or you can import from a directory. Typically, projects are preserved as archives as they are easier to move.

1-2-6. Click **Browse** to navigate to *C:\training\SDK_Profiling\support* (2).

1-2-7. Select **profLab_{standalone | linux}_(ZC702 | Zed)_start.zip**, which contains the archived projects.

1-2-8. Click **Open** to open the archive and list the various projects in that archive.

1-2-9. Select **all of the listed projects** to import (3).

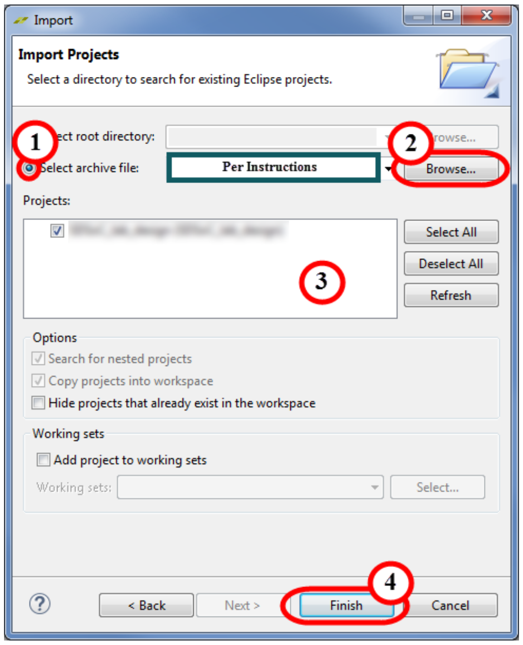


Figure 11-6: Import Settings for Archived Projects

1-2-10. Click **Finish** to perform the importing of the project (4).

Because the profiling operation slows program execution speed, the source code will be modified to ensure that the profiling completes in an acceptable time frame.

1-3. Open *SDSoC_lab_design_main.c* in the editor.

1-3-1. Locate **SDSoC_lab_design_main.c** using the Project Explorer pane.

Note: You may need to expand the branches of the tree (**project name** > **src**).

1-3-2. Double-click the source file to open it in the editor window.

Alternatively, you can right-click the source file name and select **Open**.

The **Open With** option provides access to other editors, including those outside the SDK tool environment.

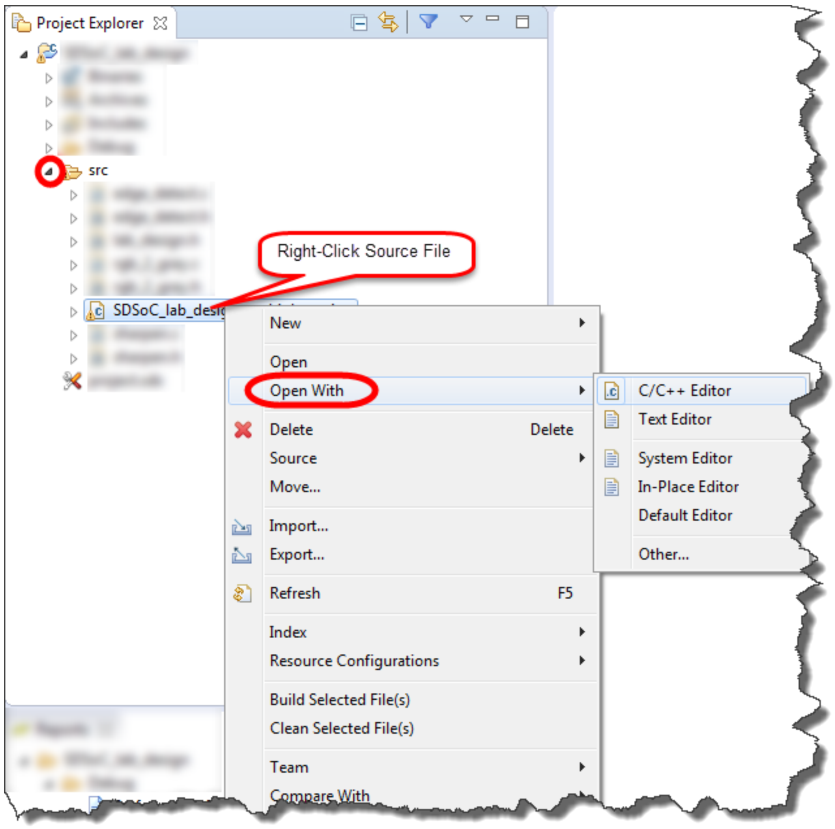


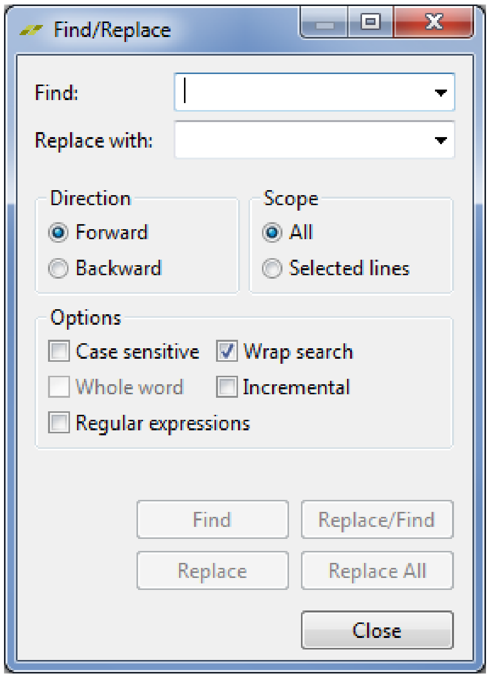
Figure 11-7: Opening a Source File via Right-Click

1-4. Find *#define LOOPS.*

The find/replace operation is accessible through both a click sequence and a keyboard shortcut.

1-4-1. Select **Edit** > **Find/Replace** or press <**Ctrl** + **F**>.

The Find/Replace dialog box opens.



Find:

Replace with:

Direction: [✓ Forward] Backward

Scope: [✓ All] Selected lines

Options: Case sensitive [✓ Wrap search] Whole word Incremental Regular expressions

Figure 11-8: Default Find/Replace Dialog Box

1-4-2. Enter **#define LOOPS** in the Find field.

1-4-3. Click **Find** or press <**Enter**> to find the next occurrence of *#define LOOPS*.

1-4-4. Continue clicking **Find** or pressing <**Enter**> until you locate the specific instance that you are looking for.

With the **Wrap search** option enabled, the file is treated as a continuous loop and the find operation will jump to the next occurrence at the top of the file (when searching forwards) or at the bottom of the file (when searching backwards). A "ding" sound is made when the search wraps around.

If you are looking for text within a specific region of the code you would first highlight the region to perform the search in, then launch the Find/Replace function as described in this topic.

1-4-5. Click **Close** to close the Find/Replace dialog box.

1-5. Set the macro *LOOPS* to 1.

This sets how many times to perform the image processing operations on the image.

For profiling in Standalone with stack tracing enabled (which this lab does), the program execution will be ~12 times slower than running normally. This is because the JTAG access to certain registers to gather performance data is limited by the JTAG cable speed. The profiling is still non-intrusive but the execution time will be reduced.

1-5-1. Edit this line to read:

#define LOOPS 1

Tip: Do not insert ';' at the end of this macro as this will make it difficult to diagnose compilation errors.

1-5-2. Save the file.

1-6. Find *return 0*.

You are finding return 0 to add a breakpoint here. With the Linux versions of the lab, when the profiling is complete, the debug session will end and the newly acquired profile data will be discarded. The breakpoint is to keep the debug session *active* so that the profile information can be inspected.

1-6-1. Using the technique you used previously, find the text "**return 0**".

Note: This is the final statement in the *main()* function.

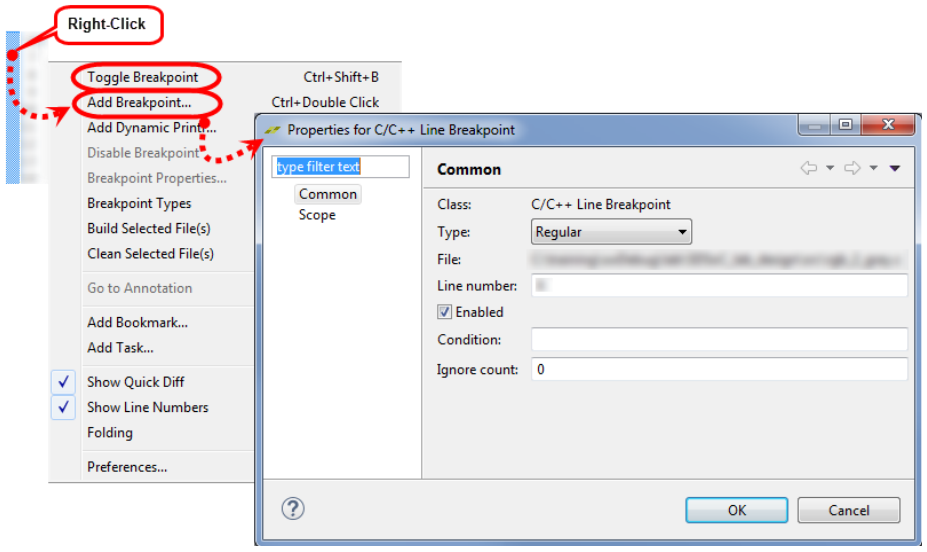
1-7. Add a breakpoint at the current line.

1-7-1. Right-click the left margin in the editor at the line at which to insert a breakpoint.

1-7-2. Select **Add Breakpoint** or **Toggle Breakpoint**.

If you select Add Breakpoint, then the Breakpoint Properties dialog box opens. Here you can specify the details for this breakpoint if necessary to create a conditional breakpoint.

If you select Toggle Breakpoint, then a simple breakpoint will be created (or removed if one already existed) on the line you specified using default breakpoint properties.



Class: C/C++ Line Breakpoint
Type: Regular
File:
Line number:
Enabled
Condition:
Ignore count: 0

Figure 11-9: Adding a Simple or Conditional Breakpoint

1-7-3. Click **OK** to create the breakpoint if Add Breakpoint is selected.

Note: This breakpoint is added to ensure that the profiling results are not automatically cleared when the program completes execution.

A Debug configuration defines how you want the system to work when performing a debug operation. Typically a debug operation switches to the Debug perspective. While there are a significant number of switches and options, the most common are shown below.

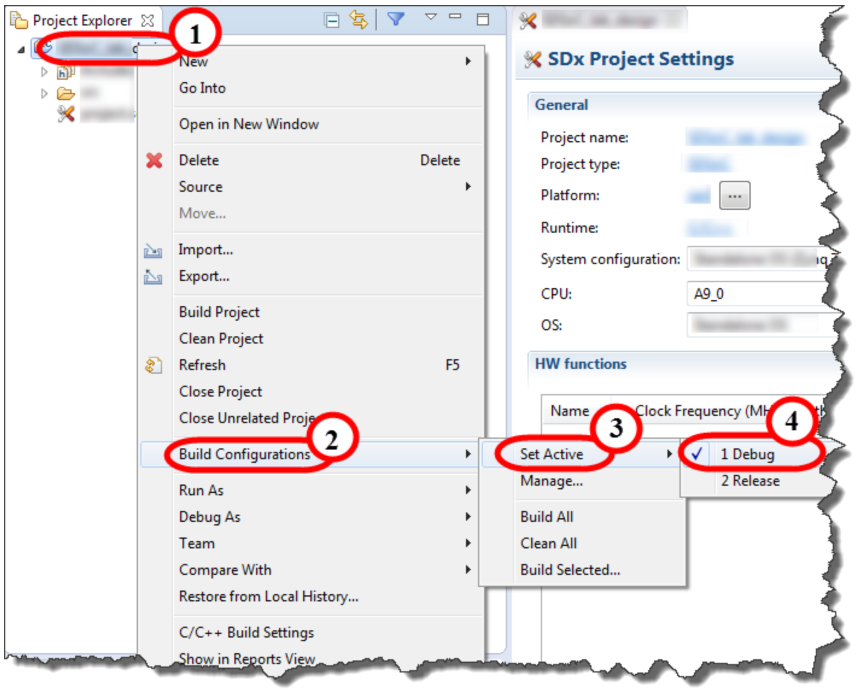
1-8. Set the build configuration to Debug.

1-8-1. Right-click the project name in the Project Explorer pane to open the context menu (1).

1-8-2. Select **Build Configurations** to view the options for building the different types of configurations (2).

1-8-3. Select **Set Active** to see the available configurations (3).

1-8-4. Select **Debug** to build the desired configuration (4).



Project name:
Project type:
Platform:
Runtime:
System configuration:
CPU: A9_0
OS:

Figure 11-10: Selecting the Debug Build Configuration (Example)

1-8-5. Right-click the project name and select **Build Project**.

This will take a minute or two to complete building the project.

Note: This is to recompile the newly modified source code. It will be recompiled with the SDDebug configuration for the SDSoC tool or Debug for the SDK tool and should take approximately two minutes.

Standalone users: You can skip to the "Configuring the SDK Tool for Standalone" step.

Linux users: In the SDSoC tool, once any configuration has been run (with the default *Build SD Card image* option on), then the tools build an SD card image that is located under <*project name*> > *[SDDebug* | *SDRelease]*.

Linux requires the board to already have been booted with Linux; otherwise the debug session will fail. There are three solutions to this issue. First, the project can be built as above and the SD card can be prepared. Second, you can allow the debug session to fail and then prepare the SD card, reboot the board, and relaunch the debug session. Third, you can have a generic Linux image already running on the board.

As the project has just been compiled and there is a generated Linux image ready to be copied to an SD card, this will be used to configure Linux on the development board so that the debug session will not fail on the first try.

For the SDK Linux project, there is no automatically generated Linux SD card image. This has been provided as part of the lab material.

Configuring the SDK Tool for Linux Step 2

Several tasks must be performed in order to successfully run Linux. These tasks may have been performed for you by your training provider in a classroom environment. If you are outside of this environment, you can refer to the following topics in the *Lab Setup Guide*:

- Configure the jumper settings on the board to boot from the SD card.
- Set the host's Ethernet address to a static IP (suggested: 192.168.1.11).
- Identify the proper COM port (must be done with the board powered on).

Note: If you have a preconfigured SD card available as part of the lab, the following *Preparing an SD card* instruction is optional. If you do not have a preconfigured SD card, it may be necessary to build the project and generate an SD card image (if using the SDSoC development environment) or copy from the default directory as shown below.

Linux must boot from a properly configured SD card. Many tools generate the necessary file images for Linux, FreeRTOS, and standalone that can be simply copied from a specified location on the host machine to the SD card.

2-1. Prepare an SD card.

2-1-1. Insert an SD card into the PC's SD card slot.

2-1-2. Browse to the SD card drive using Windows Explorer.

Optional: You may want to erase or reformat the SD card at this time, which may prevent any unwanted interaction with other files that may be on the card.

2-1-3. Open a second Windows Explorer window to browse to the files that you will copy to the SD card.

2-1-4. Browse to the image located at *C:\training\SDK_Profiling\support\Standard_[ZC702 | Zedboard]_SDCard_Linux_Image.zip*.

2-1-5. Drag-and-drop all the files from the source directory to the SD card.

Note: The files should go into the root of the SD card.

2-1-6. Close both Windows Explorer windows.

2-1-7. Remove the SD card from the PC card slot.

2-1-8. Turn off power to the hardware platform.

Note: The boot selection switches or jumpers must be properly set to boot from the SD card. See the appropriate section in the *Lab Setup Guide*.

Insert the SD card into its slot on the hardware platform.

2-2. Apply power to (turn on) the ZC702 or ZedBoard hardware platform.

2-2-1. Make sure that AC power is connected to the power brick.

2-2-2. Slide the power switch to the on position.

Some LEDs on the board will illuminate when the board is powered.

2-3. Launch and configure your serial port terminal emulator.

The Terminal tab can be used for the ZC702 board; however, there is an issue with drivers when information is sent to a ZedBoard. Tera Term is a freeware serial port terminal emulator that has been tested successfully with the target boards.

2-3-1. Identify the COM port associated with your board.

If you are unfamiliar with this process, refer to the *Lab Setup Guide*.

2-3-2. Set the serial port connection within the serial port terminal emulator to **115200** baud, **8** data bits, No parity, **1** stop bit.

If you are unfamiliar with how to use Tera Term, refer to the *Lab Setup Guide*.

If you are unfamiliar with how to use the Terminal tab, refer to the *Lab Reference Guide* under either **SDSoC** or **SDK Tool Operations** > **Configuring the Terminal**.

Now that the board is powered on and Linux has finished its boot process, the Linux environment must be configured by using the serial port emulator terminal.

Note: You may have to wait for a few minutes for Linux to boot.

2-4. Log in to Linux.

2-4-1. Log in with the user name and password of **root** / **root** when prompted in the serial port terminal.

2-5. Set the IP address of the board.

2-5-1. Enter the following at the Linux command prompt to change the IP address of the board to 192.168.1.10:

```
ifconfig eth0 192.168.1.10
```

Note: This can be any address other than the one that the host is configured to.

Optional: You can verify the IP address by entering the following command:

```
ifconfig eth0
```

The response should be *similar* to the following:

```
eth0     Link encap:Ethernet  HWaddr 00:0A:35:00:01:22
         inet addr:192.168.1.10  Bcast:192.168.1.255
Mask:255.255.255.0
         inet6 addr: fe80::20a:35ff:fe00:122/64 Scope:Link
         UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
         RX packets:23 errors:0 dropped:0 overruns:0 frame:0
         TX packets:9 errors:0 dropped:0 overruns:0 carrier:0
         collisions:0 txqueuelen:1000
         RX bytes:2913 (2.8 KiB)  TX bytes:1446 (1.4 KiB)
         Interrupt:54 Base address:0xb000
```

2-6. Verify the Ethernet connectivity between the host and the development board.

This will also indirectly verify that the host PC Ethernet port is set to an IP address of 192.168.1.11.

2-6-1. Ping the host using the Linux terminal console to verify connectivity between the host and the target:

```
ping 192.168.1.11 -c 1
```

If the ping was successful (indicated by a 0% packet loss with roundtrip times), you can continue to the next section.

If it was not successful, there are several possibilities:

- The host IP address is not set properly.
  - See instructions for configuring the host's IP address in the *Lab Setup Guide*.
- The Ethernet cable may not be properly connected.
  - Unplug and replug the cable on both ends; verify that the Ethernet LEDs are flickering (if available).
- The Windows firewall is blocking the connection. Follow the procedure below to disable the Windows firewall.
  - Access the Windows Firewall controls through the Control Panel (select **Start** > **Control Panel** > **System and Security** > **Windows Firewall**).
  - From the options in the left sidebar, select **Turn Windows Firewall on or off**.
  - Select the **Turn off Windows Firewall** option for both network location settings.
- The sub-net address for either the board or host may not be entered correctly.
  - If you need to configure your PC's Ethernet port, refer to "Configuring the PC's Ethernet Port for Remote System Explorer" section under SDK Operations > Linux and Remote System Explorer in the *Lab Reference Guide*.

Configuring the SDK Tool for Standalone Step 3

A couple of tasks must be performed in order to successfully run Standalone. These tasks may have been performed for you by your training provider in a classroom environment. If you are outside of this environment, refer to the following topics in the *Lab Setup Guide*:

- Configure the jumper settings on the board to boot from JTAG.
- Connect and power up the board.
- Identify the proper COM port (must be done with the board powered on).

3-1. Apply power to (turn on) the ZC702 or ZedBoard hardware platform.

3-1-1. Make sure that AC power is connected to the power brick.

3-1-2. Slide the power switch to the on position.

Some LEDs on the board will illuminate when the board is powered.

3-1-3. Connect the JTAG and UART cable from the host PC to the laptop.

3-2. Launch and configure Tera Term (or equivalent serial port terminal emulator). The SDx tool Terminal tab can be used for the ZC702 board; however, there is an issue with drivers when information is sent to a ZedBoard.

3-2-1. Identify the COM port.

If you are unfamiliar with this process, refer to the *Lab Setup Guide*.

3-2-2. Set the serial port connection to **115200 8N1**.

If you are unfamiliar with how to use Tera Term, refer to the *Lab Setup Guide*.

Now that the board is powered on, it is ready to be programmed.

Running the Default Profile Step 4

For Standalone users, proceed with the instruction below. For Linux users, skip to instructions below that begin with "For Linux users, proceed with the instructions below."

A Debug configuration defines how you want the system to work when performing a debug operation and maps an ELF object file to a target for execution. Typically, a debug operation switches to the Debug perspective. While there are a significant number of switches and options, the most common are shown below.

4-1. Set up a debug configuration for a specific application project.

4-1-1. Right-click the application project that you want to build the Debug configuration for from the Project Explorer pane (1).

4-1-2. Select **Debug As** to open the menu of predefined configurations and the configuration manager (2).

4-1-3. Select **Debug Configurations** to view all the available debug options (3).

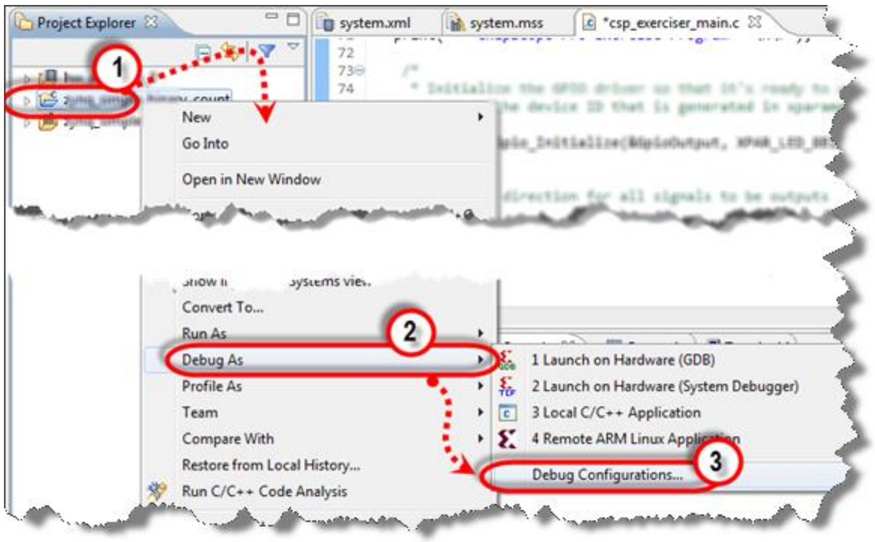


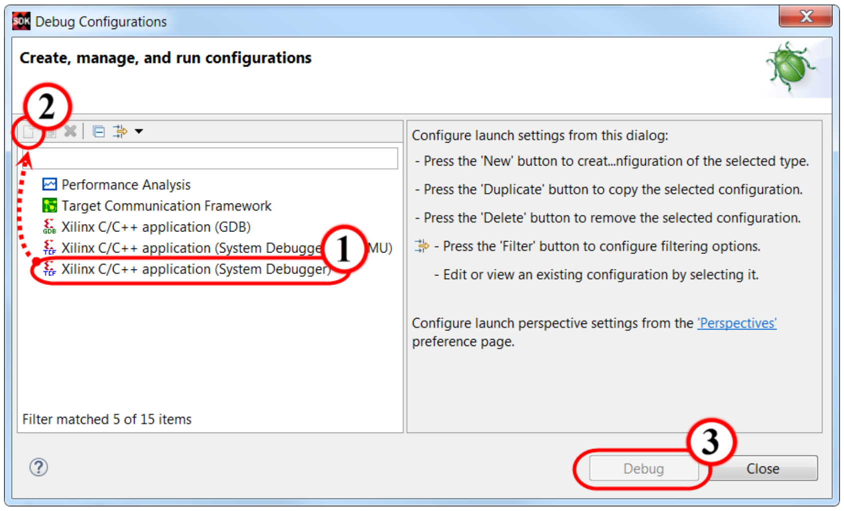
Figure 11-11: Creating a Debug Configuration

The Debug Configurations dialog box opens.

4-1-4. Select **Xilinx C/C++ application (System Debugger)** since you will be debugging this type of system (1).

GDB still works; however, it is considered deprecated for new designs.

4-1-5. Click the **Create New Configuration** icon to create the new configuration for your application (2).


Debug Configurations

Create, manage, and run configurations

Performance Analysis
Target Communication Framework
Xilinx C/C++ application (GDB)
Xilinx C/C++ application (System Debugger on QEMU)
Xilinx C/C++ application (System Debugger)

Filter matched 5 of 15 items

Configure launch settings from this dialog:
- Press the 'New' button to creat...nfiguration of the selected type.
- Press the 'Duplicate' button to copy the selected configuration.
- Press the 'Delete' button to remove the selected configuration.
- Press the 'Filter' button to configure filtering options.
- Edit or view an existing configuration by selecting it.

Configure launch perspective settings from the 'Perspectives' preference page.

Figure 11-12: Creating a New Debug Configuration

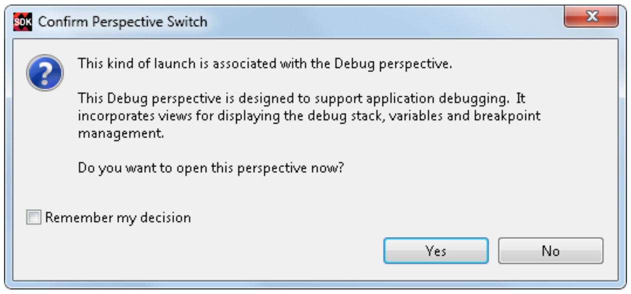
A new Debug Configuration is created. The previous figure depicts an SDK workspace that does not yet have any debug or run configurations defined. If other configurations did exist, or after the creation of a first configuration, the new configuration will appear under the type of configuration.

Note the default configuration name and other parameters that are automatically filled in. In most cases, you just need to click the Debug button to begin the session. This Debug Configuration menu is useful when you want to later change debug parameters.

The new configuration will appear with other existing configurations and have the name of your application. You will also note that a number of fields are automatically filled in for you using the name of your application as the basis (that is, if your application is named "XYZ", then the Name field will be populated with XYZ Debug, and the C/C++ Application field will be populated with Debug\XYZ.elf).

4-1-6. Click **Debug** to close the window and launch the debugging session (3).

If the Confirm Perspective Switch dialog box appears, click **Yes**.


Confirm Perspective Switch

This kind of launch is associated with the Debug perspective.

This Debug perspective is designed to support application debugging. It incorporates views for displaying the debug stack, variables and breakpoint management.

Do you want to open this perspective now?

Figure 11-13: Confirming Switch of Perspective

4-1-7. The Debug Perspective view opens.

For Linux users, proceed with the instructions below.

Run and Debug configurations are application project objects that contain communication, hardware, and execution options for running an application on a hardware or emulation platform. The selections for *Run* and *Debug* are identical and only differ in that *Run* just executes the application and *Debug* opens a debug perspective and launches a debug program.

There are different types of Run and Debug configurations based on the operating system (or Standalone libraries) and the SDK download/debug tools that you want to use. Multiple configurations can be defined for any project. This facilitates enhanced development and debugging.

4-2. Create a new Linux Debug configuration. Debug and Run configurations associate an ELF object file to a target (typically a hardware board) for execution. In this case, the target is a hardware board accessed over the Ethernet TCP/IP connection.

4-2-1. Click the **C/C++** tab in the upper-right corner of the GUI to return to the C/C++ perspective.

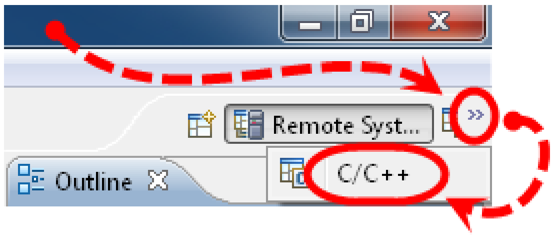


Figure 11-14: Changing Perspective

This brings back access to the software projects. It is accessed by first clicking the **>>** in the same location. If this tab is not available, you can also return to the perspective by selecting **Window** > **Open Perspective** > **Other** > **C/C++ (default)**.

4-2-2. Right-click **profLab** in the Project Explorer window (1).

4-2-3. Select **Debug As** (2) > **Debug Configurations** (3).

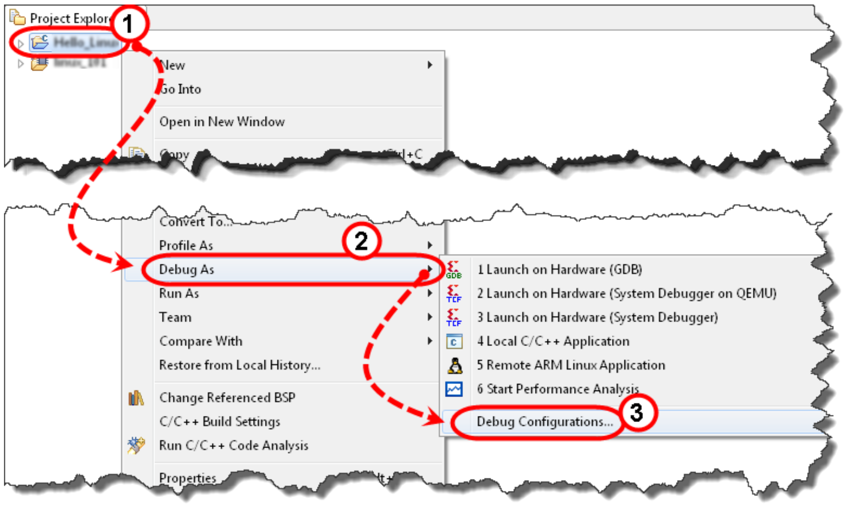

Project Explorer
1
New
Go Into
Open in New Window
Copy
Convert To...
Profile As
2
Debug As
Run As
Team
Compare With
Restore from Local History...
Change Referenced BSP
C/C++ Build Settings
Run C/C++ Code Analysis
Properties
1 Launch on Hardware (GDB)
2 Launch on Hardware (System Debugger on QEMU)
3 Launch on Hardware (System Debugger)
4 Local C/C++ Application
5 Remote ARM Linux Application
6 Start Performance Analysis
Debug Configurations...
3

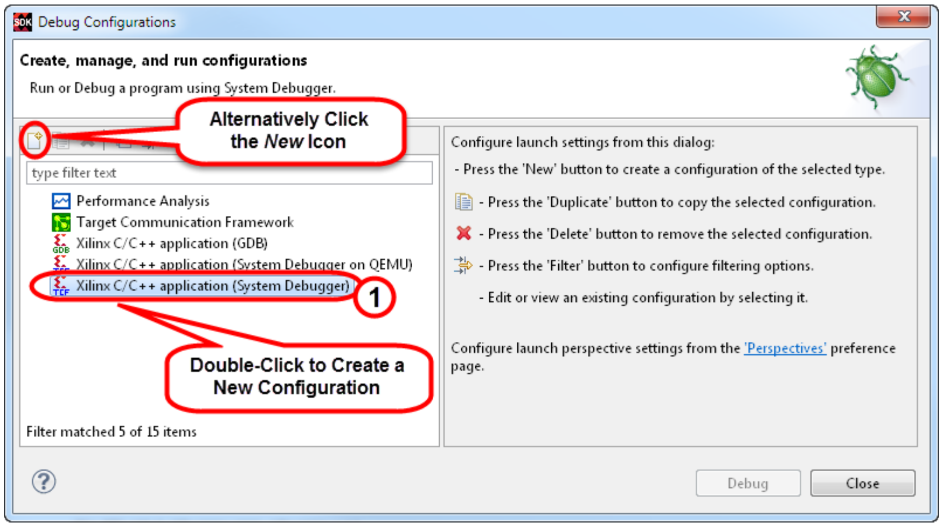
Figure 11-15: Selecting Debug Configurations

The Debug Configurations dialog box opens.

4-2-4. Double-click **Xilinx C/C++ application (System Debugger)** to create a launch configuration (1).

Alternatively, you can select **Xilinx C/C++ application (System Debugger)** and click the **New** configuration button.

If this is the first debug configuration being created for the project, a welcome type dialog opens. If one or more configurations exist, then the last open configuration will be displayed. In either case, a new configuration can always be added. Existing configurations are shown in the left pane and can be selected for debugging.


Debug Configurations
Alternatively Click the New Icon
1
Double-Click to Create a New Configuration
Debug
Close

Create, manage, and run configurations

Run or Debug a program using System Debugger.

type filter text

Performance Analysis
Target Communication Framework
Xilinx C/C++ application (GDB)
Xilinx C/C++ application (System Debugger on QEMU)
Xilinx C/C++ application (System Debugger)

Filter matched 5 of 15 items

Configure launch settings from this dialog:
- Press the 'New' button to create a configuration of the selected type.
- Press the 'Duplicate' button to copy the selected configuration.
- Press the 'Delete' button to remove the selected configuration.
- Press the 'Filter' button to configure filtering options.
- Edit or view an existing configuration by selecting it.

Configure launch perspective settings from the 'Perspectives' preference page.

Figure 11-16: Creating a New System Run/Debug Configuration

4-3. Configure the debug type and host connection.

4-3-1. Select **Linux Application Debug** from the Debug Type drop-down list (2).

This will engage the proper debug tool to use.

4-3-2. Click **New** next to the Connection drop-down list to launch the Target Connection Details dialog box (3).

A new connection will be defined from the debugger to the hardware (or emulator) target.

4-3-3. Enter **ZynqBoard** in the Target Name field as the name of the connection (4).

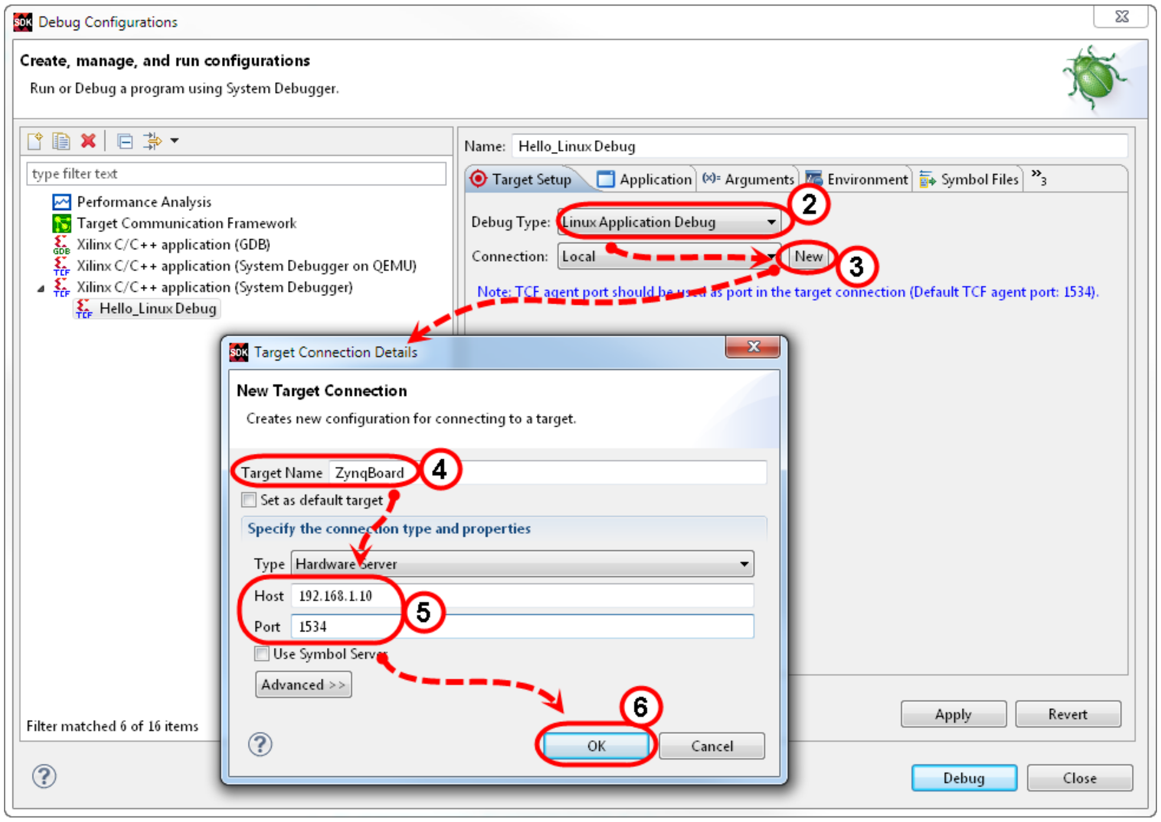
4-3-4. Enter **192.168.1.10** in the Host field (5).

This is the IP address of the RSE connection that was set up earlier.

4-3-5. Enter **1534** in the the Port field (5).

This is the default TCP/IP port number for the connection.

4-3-6. Click **OK** (6).


Debug Configurations
Create, manage, and run configurations
Run or Debug a program using System Debugger.
type filter text
Performance Analysis
Target Communication Framework
Xilinx C/C++ application (GDB)
Xilinx C/C++ application (System Debugger on QEMU)
Xilinx C/C++ application (System Debugger)
Hello_Linux Debug
Filter matched 6 of 16 items
Target Setup
Application
Arguments
Environment
Symbol Files
Apply
Revert
Debug
Close
Target Connection Details
New Target Connection
Creates new configuration for connecting to a target.
Set as default target
Specify the connection type and properties
Use Symbol Server
Advanced >>
OK
Cancel
2
3
4
5
6

Name: Hello_Linux Debug

Debug Type: Linux Application Debug

Connection: Local

Note: TCF agent port should be used as port in the target connection (Default TCF agent port: 1534).

Target Name ZynqBoard

Type Hardware Server

Host 192.168.1.10

Port 1534

Figure 11-17: Selecting the Debug Type and Connection

4-4. Select the software application ELF file to debug and its file path on the remote target board where it is to be copied.

The actual software application debugging takes place on the Linux platform running on the target hardware, so it is necessary to put a copy of the ELF file on it.

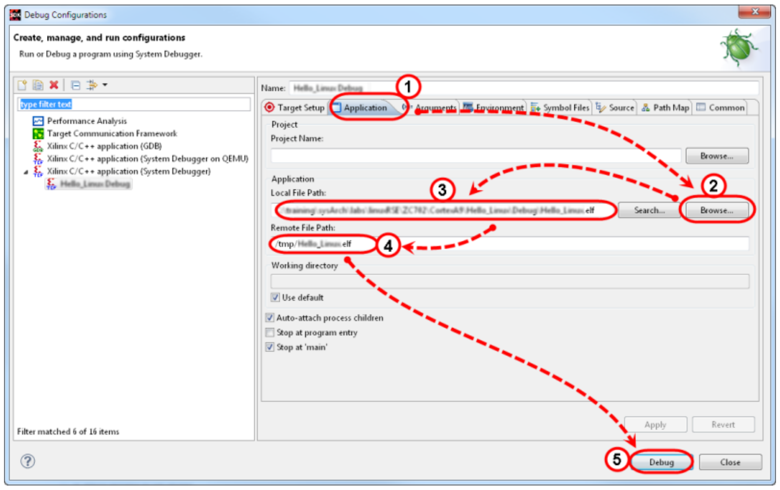
4-4-1. Select the **Application** tab (1).

This is where the software application is selected and allows configuration of where the application is placed in the remote file system.

4-4-2. Click **Browse** next to the Local File Path field to select the software application ELF file (2).

4-4-3. Navigate to and select the file **C:\training\SDK_Profiling\lab\profLab\Debug\profLab.elf** (3).

4-4-4. Enter **/tmp/profLab.elf** in the Remote File Path field as the location on the target platform where the application ELF file will be copied (4).


Debug Configurations

Create, manage, and run configurations

Run or Debug a program using System Debugger.

type filter text
- Performance Analysis
- Target Communication Framework
- Xilinx C/C++ application (GDB)
- Xilinx C/C++ application (System Debugger on QEMU)
- Xilinx C/C++ application (System Debugger)
  - [illegible]

Filter matched 6 of 16 items

Name: [illegible]

Target Setup | Application | Arguments | Environment | Symbol Files | Source | Path Map | Common

Project
Project Name:

Application
Local File Path: [illegible].elf
Remote File Path: /tmp/[illegible].elf

Working directory
- [✓] Use default

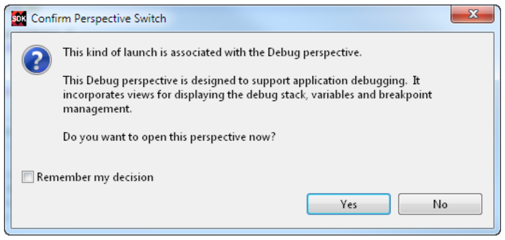
- [✓] Auto-attach process children
- [ ] Stop at program entry
- [✓] Stop at 'main'

Figure 11-18: Selecting the Local File Path and Remote File Path

The remaining options will be accepted at their default values.

4-4-5. Click **Debug** (5).

4-4-6. Click **Yes** to confirm opening the Debug perspective view.


Confirm Perspective Switch

This kind of launch is associated with the Debug perspective.

This Debug perspective is designed to support application debugging. It incorporates views for displaying the debug stack, variables and breakpoint management.

Do you want to open this perspective now?

- [ ] Remember my decision

Figure 11-19: Confirming Perspective Switch

The Debug perspective opens.

4-5. Optional for Linux: Highlight the process for debug.

Sometimes the process for debugging is not highlighted in the Debug view and the flow controls such as pause, resume, single step, etc. will be grayed out. Follow the procedure below to enable the controls.

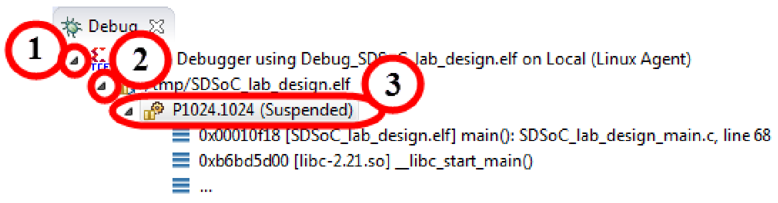
4-5-1. Click the down arrow in the Debug view, if it is not already expanded, to expand the top-level debug session (1).

4-5-2. Click the down arrow, if it is not already expanded, to expand the next level showing the binary that is being executed (2).

Note: The location of the process on the remote device (**/tmp/profLab.elf**) is also visible.

4-5-3. Click the process that is currently suspended (3).

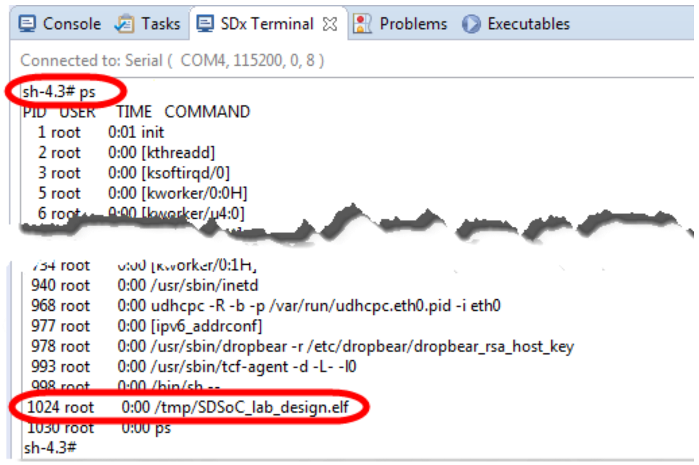
Note: This will tell the Debug tools which remote application/process to debug and enable the flow control tools.



```
Debugger using Debug_SDSoC_lab_design.elf on Local (Linux Agent)
/tmp/SDSoC_lab_design.elf
P1024.1024 (Suspended)
0x00010f18 [SDSoC_lab_design.elf] main(): SDSoC_lab_design_main.c, line 68
0xb6bd5d00 [libc-2.21.so] __libc_start_main()
...
```

Figure 11-20: Highlighting the Debug Session Application

Tip: In the figure above and below, P1024 is the process ID (PID) of the application running under Linux; your PID may be different. Issuing the `ps` command through the terminal will show all of the system processes that are currently running. There will be a listing with the same PID that matches the number of your suspended application.



```
Connected to: Serial ( COM4, 115200, 0, 8 )
sh-4.3# ps
PID  USER      TIME  COMMAND
  1 root      0:01 init
  2 root      0:00 [kthreadd]
  3 root      0:00 [ksoftirqd/0]
  5 root      0:00 [kworker/0:0H]
  6 root      0:00 [kworker/u4:0]

 734 root      0:00 [kworker/0:1H]
 940 root      0:00 /usr/sbin/inetd
 968 root      0:00 udhcpc -R -b -p /var/run/udhcpc.eth0.pid -i eth0
 977 root      0:00 [ipv6_addrconf]
 978 root      0:00 /usr/sbin/dropbear -r /etc/dropbear/dropbear_rsa_host_key
 993 root      0:00 /usr/sbin/tcf-agent -d -L- -l0
 998 root      0:00 /bin/sh --
1024 root      0:00 /tmp/SDSoC_lab_design.elf
1030 root      0:00 ps
sh-4.3#
```

Figure 11-21: Linux Terminal – ps Command

You will now set up and run profiling for the project.

4-6. Open the TCF Profiler to view where the processor is spending time in the code.

4-6-1. Select **Window** (1) > **Show View** (2) > **Other** (3).

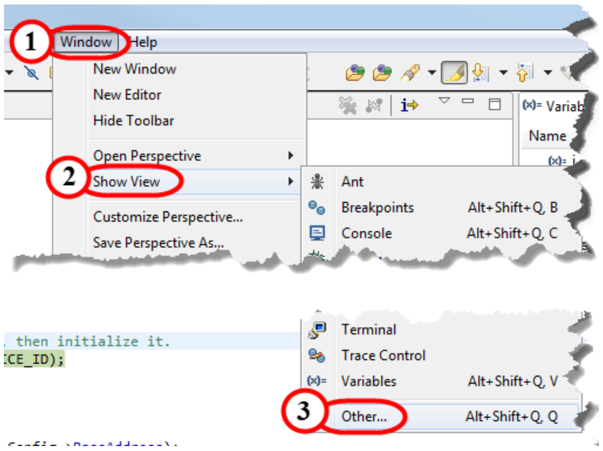


Figure 11-22: Viewing Hotspots

The Show View dialog box opens.

4-6-2. Expand **Debug** and select **TCF Profiler**.

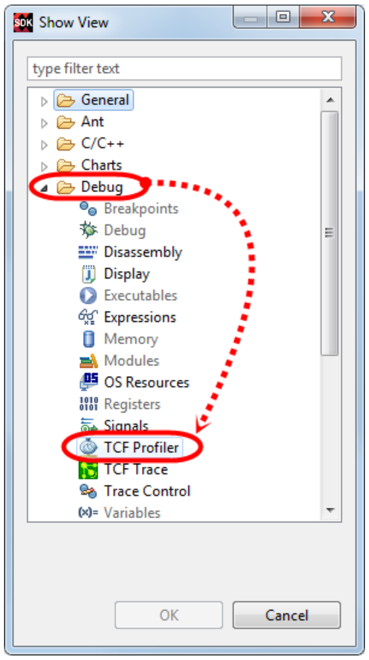
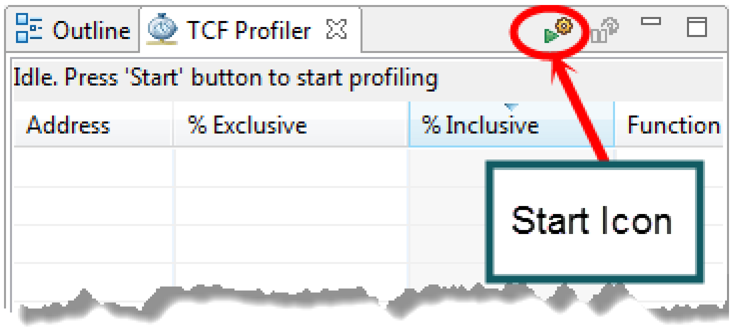


Figure 11-23: Selecting TCF Profiler

4-6-3. Click **OK**.

This opens the TCF Profiler view in the SDK tool.

4-6-4. Click **Start** in the TCF Profiler view in the tool on the right side.



Idle. Press 'Start' button to start profiling

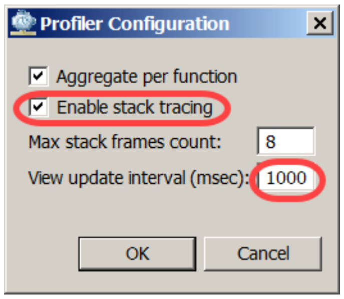
Figure 11-24: Starting the TCF Profiler

4-6-5. Select **Aggregate per function** in the Profiler Configuration dialog box.

4-6-6. Select the **Enable stack tracing** option if it is not selected.

The update rate refers to how often the information on the screen is updated. The default setting is four seconds.

The *Enable stack tracing* and *Max stack frames count* options are useful for debugging stack overflows.



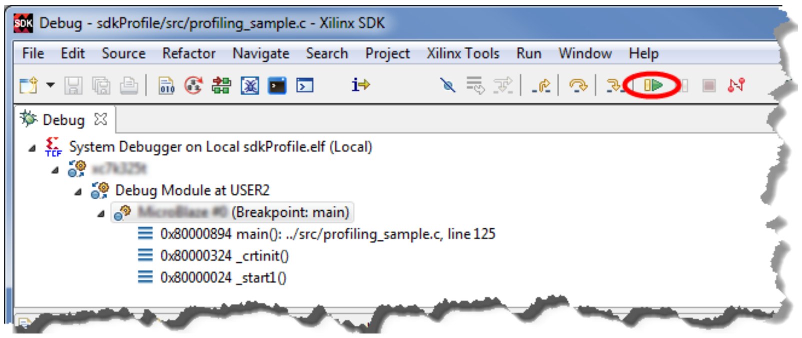
[✓] Aggregate per function
[✓] Enable stack tracing
Max stack frames count: 8
View update interval (msec): 1000

Figure 11-25: Selecting the Profiler Configuration

4-6-7. Click **OK** to accept these settings and arm the profiler.

4-7. Run the software application.

4-7-1. Click the **Resume** icon to run the program.


Debug - sdkProfile/src/profiling_sample.c - Xilinx SDK
File Edit Source Refactor Navigate Search Project Xilinx Tools Run Window Help
Debug

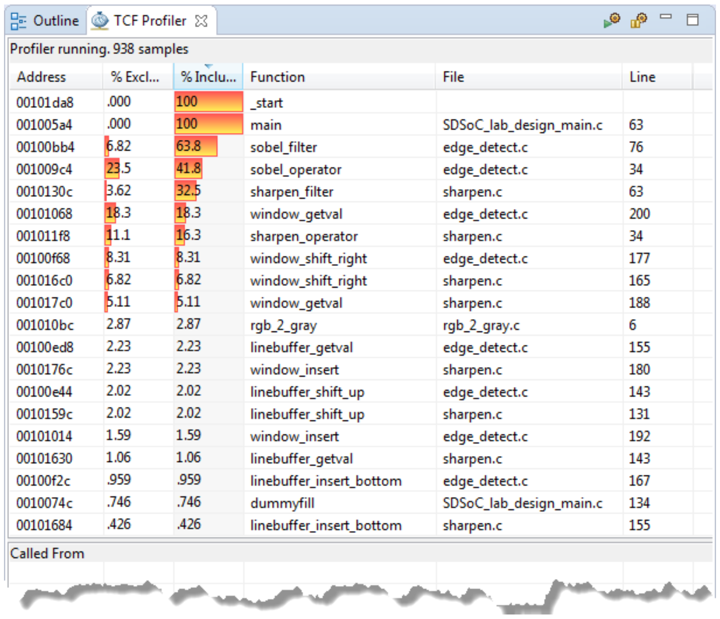
```
System Debugger on Local sdkProfile.elf (Local)
  [redacted]
    Debug Module at USER2
      [redacted] (Breakpoint: main)
        0x80000894 main(): ../src/profiling_sample.c, line 125
        0x80000324 _crtinit()
        0x80000024 _start1()
```

Figure 11-26: Resuming the Program

The program will stop at the previously enabled breakpoint in the *main* function.

4-7-2. Select the **TCF Profiler** window.

Note: You can drag this tab to the main view or resize the current pane to see the contents more clearly.



Profiler running. 938 samples

Address	% Excl...	% Inclu...	Function	File	Line
00101da8	.000	100	_start		
001005a4	.000	100	main	SDSoC_lab_design_main.c	63
00100bb4	6.82	63.8	sobel_filter	edge_detect.c	76
001009c4	23.5	41.8	sobel_operator	edge_detect.c	34
0010130c	3.62	32.5	sharpen_filter	sharpen.c	63
00101068	18.3	18.3	window_getval	edge_detect.c	200
001011f8	11.1	16.3	sharpen_operator	sharpen.c	34
00100f68	8.31	8.31	window_shift_right	edge_detect.c	177
001016c0	6.82	6.82	window_shift_right	sharpen.c	165
001017c0	5.11	5.11	window_getval	sharpen.c	188
001010bc	2.87	2.87	rgb_2_gray	rgb_2_gray.c	6
00100ed8	2.23	2.23	linebuffer_getval	edge_detect.c	155
0010176c	2.23	2.23	window_insert	sharpen.c	180
00100e44	2.02	2.02	linebuffer_shift_up	edge_detect.c	143
0010159c	2.02	2.02	linebuffer_shift_up	sharpen.c	131
00101014	1.59	1.59	window_insert	edge_detect.c	192
00101630	1.06	1.06	linebuffer_getval	sharpen.c	143
00100f2c	.959	.959	linebuffer_insert_bottom	edge_detect.c	167
0010074c	.746	.746	dummyfill	SDSoC_lab_design_main.c	134
00101684	.426	.426	linebuffer_insert_bottom	sharpen.c	155

Called From

Figure 11-27: Profiling Sample Results

Note: You may see slightly different values due to the sampling.

Address is the location of the function in the Disassembly view (i.e., its location in memory).

% Exclusive is the percentage of samples encountered by the profiler for that function only (does not take into consideration samples of child functions). This can also be seen as the exclusive percentage for that particular function. This parameter can be of help in identifying performance bottlenecks.

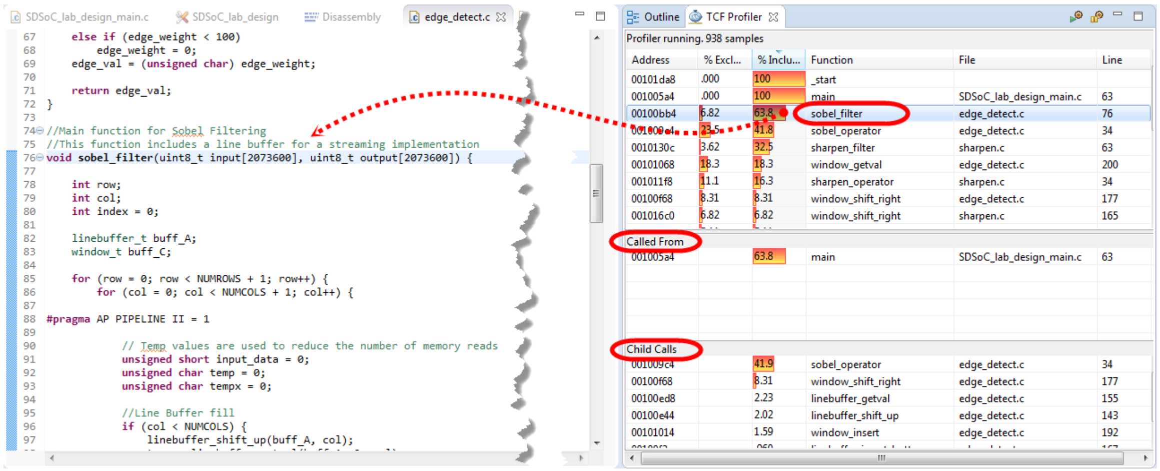
% Inclusive is the percentage of samples of a function, including samples collected during execution of child functions.

Function is the function being sampled.

File is the file containing the function definition.

Line is the line where the function is defined in the said file.

4-7-3. Click the function name **sobel_filter** in the TCF Profiler window.



```
    else if (edge_weight < 100)
        edge_weight = 0;
    edge_val = (unsigned char) edge_weight;

    return edge_val;
}

//Main function for Sobel Filtering
//This function includes a line buffer for a streaming implementation
void sobel_filter(uint8_t input[2073600], uint8_t output[2073600]) {

    int row;
    int col;
    int index = 0;

    linebuffer_t buff_A;
    window_t buff_C;

    for (row = 0; row < NUMROWS + 1; row++) {
        for (col = 0; col < NUMCOLS + 1; col++) {

#pragma AP PIPELINE II = 1

            // Temp values are used to reduce the number of memory reads
            unsigned short input_data = 0;
            unsigned char temp = 0;
            unsigned char tempx = 0;

            //Line Buffer fill
            if (col < NUMCOLS) {
                linebuffer_shift_up(buff_A, col);
```

Profiler running. 938 samples

Address	% Excl...	% Inclu...	Function	File	Line
00101da8	.000	100	_start		
001005a4	.000	100	main	SDSoC_lab_design_main.c	63
00100bb4	6.82	63.8	sobel_filter	edge_detect.c	76
001009c4	23.5	41.8	sobel_operator	edge_detect.c	34
0010130c	3.62	32.5	sharpen_filter	sharpen.c	63
00101068	18.3	18.3	window_getval	edge_detect.c	200
001011f8	11.1	16.3	sharpen_operator	sharpen.c	34
00100f68	8.31	8.31	window_shift_right	edge_detect.c	177
001016c0	6.82	6.82	window_shift_right	sharpen.c	165

Called From

001005a4		63.8	main	SDSoC_lab_design_main.c	63

Child Calls

001009c4		41.9	sobel_operator	edge_detect.c	34
00100f68		8.31	window_shift_right	edge_detect.c	177
00100ed8		2.23	linebuffer_getval	edge_detect.c	155
00100e44		2.02	linebuffer_shift_up	edge_detect.c	143
00101014		1.59	window_insert	edge_detect.c	192

Figure 11-28: Viewing the Sobel Filter from the Profiler

This will open the *edge_detect.c* file in the editor and the selected function will be highlighted.

Selecting the function in the TCF Profiler can also show where it is called from and who its child calls are.

Question 1

What are the child functions of *main*?

Question 2

What are the parent and child functions of *sobel_operator*?

Question 3

How many times are x_weight and y_weight calculated every time *sobel_operator* is called?

Question 4

How many times will *sobel_operator* be called for the image of size 1920 * 1080?

Question 5

How many times will x_weight and y_weight be calculated for each image passed through *sobel_filter*? **Hint:** Review the answers to the previous two questions.

Question 6

What does the program *main()* do?

Question 7

What are the % Exclusive values for the functions *sobel_filter()*, *sharpen_filter()*, and *rgb_2_gray()*?

Analyzing the TCF Data Step 5

Using the results from the TCF Profiler, you will analyze the function's exclusive and inclusive values, determine which uses the most execution time, and trace the program hierarchy.

The following diagram provides a visual representation of exclusive and inclusive.

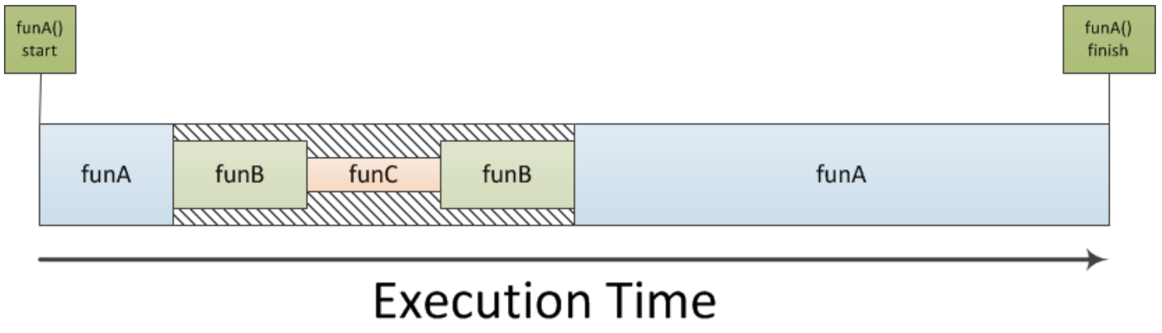


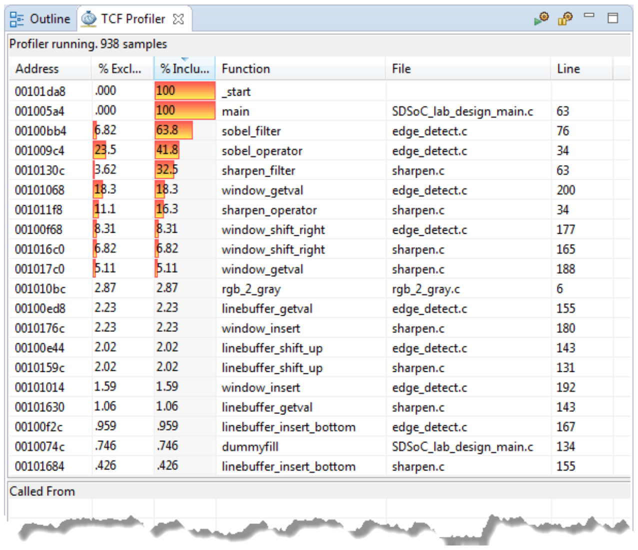
Figure 11-29: Exclusive vs Inclusive Pipelined

Exclusive: The amount of execution time spent in *funA* alone. Referencing the above diagram, the exclusive time for *funA* is represented by the combined execution time of the funA blocks only.

Inclusive: The amount of execution time spent in *funA* and all of its sub-function calls. From the diagram, this is the exclusive time of funA combined with the hatched area during which time *funB* and *funC* are executing.

5-1. View the sample results.

5-1-1. Open the **TCF Profiler** viewer with your profile results.



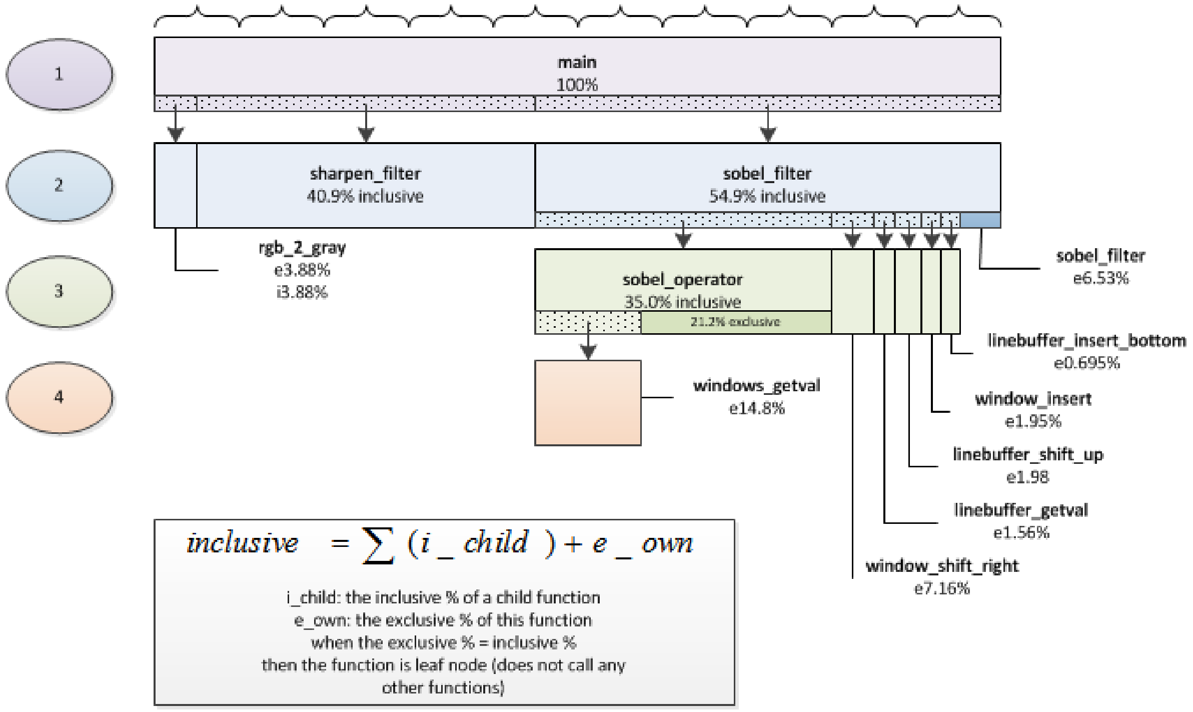
Profiler running. 938 samples

Address	% Excl...	% Inclu...	Function	File	Line
00101da8	.000	100	_start		
001005a4	.000	100	main	SDSoC_lab_design_main.c	63
00100bb4	6.82	63.8	sobel_filter	edge_detect.c	76
001009c4	23.5	41.8	sobel_operator	edge_detect.c	34
0010130c	3.62	32.5	sharpen_filter	sharpen.c	63
00101068	18.3	18.3	window_getval	edge_detect.c	200
001011f8	11.1	16.3	sharpen_operator	sharpen.c	34
00100f68	8.31	8.31	window_shift_right	edge_detect.c	177
001016c0	6.82	6.82	window_shift_right	sharpen.c	165
001017c0	5.11	5.11	window_getval	sharpen.c	188
001010bc	2.87	2.87	rgb_2_gray	rgb_2_gray.c	6
00100ed8	2.23	2.23	linebuffer_getval	edge_detect.c	155
0010176c	2.23	2.23	window_insert	sharpen.c	180
00100e44	2.02	2.02	linebuffer_shift_up	edge_detect.c	143
0010159c	2.02	2.02	linebuffer_shift_up	sharpen.c	131
00101014	1.59	1.59	window_insert	edge_detect.c	192
00101630	1.06	1.06	linebuffer_getval	sharpen.c	143
00100f2c	.959	.959	linebuffer_insert_bottom	edge_detect.c	167
0010074c	.746	.746	dummyfill	SDSoC_lab_design_main.c	134
00101684	.426	.426	linebuffer_insert_bottom	sharpen.c	155

Called From

Figure 11-30: Profiling Sample Results

Note: The values in the figure below came from a separate run of the TCF Profiler than those in the screenshot above. As the Profiling results are probabilistic, variation in results between runs is expected. Your results will not exactly match those provided in the figure and diagram.



$$inclusive = \sum (i\_child) + e\_own$$

Figure 11-31: Visual Analysis of TCF Profiler

In the figure above a detailed hierarchy of the function *sobel_filter* is shown. Focusing on the *sobel_filter* hierarchy, note that there are four levels of function calls. The root function is *main,* which runs from level 1. As all other functions are called from *main*, *main* takes up 100% of the inclusive execution time, yet ~ 0% of the exclusive execution time. The function *main* calls *sobel_filter*, *sharpen_filter*, and *rgb_2_gray*, all level 2 functions, which take up 54.9%, 40.9%, and 3.88% of the execution time, respectively.

For this illustration, you are focusing on the *sobel_filter* function. The exclusive value is reported to be 6.53%. Referring to the diagram above, this means that *sobel_filter* calls several level 3 functions that take up (54.9% - 6.53%) = 48.37% of the execution time. All of the level 3 functions besides *sobel_operator* are leaf nodes (their exclusive value equals their inclusive value) and they make no calls to any other functions.

sobel_operator is a level three function that takes up 35.0% of the execution time inclusive and 21.2% exclusive. (35.0 - 21.2) = 14.8% of the execution time accounted for by *sobel_operator* are from level 4 functions.

Note: The equation provided in the above diagram may be of interest when you are working out the inclusive value of any given function. It is a recursive equation where the base case occurs at the leaf nodes.

Note: All exclusive percentages are out of the total 100% regardless of the level of the function. For example, *windows_getval* is reported to have an exclusive % of 6.97. This is 6.97% of 100% from *main* and not 6.97% of the 35.0% from its parent function *sobel_operator*.

Note: Due to sampling differences and binning errors, there may be some deviance in the numbers.

Based on the above diagram, drawn from the TCF Profiler results, it can be seen that the level 3 function *sobel_operator* takes up 21.2% of the execution time. Also the functions *windows_getval* and *window_shift_right* take up ~15% and ~7% each. Combined, this is ~43% of the total execution time of the system.

As the function *sobel_filter* is concerned only with edge detection, and three components of this image process take up ~43% of the execution time, these functions make good candidates to analyze for acceleration.

5-2. Analyze the *sharpen_filter* hierarchy.

The purpose of this instruction is for you to inspect the *sharpen_filter* function, fill out the diagram below and answer some questions on the information collected. The TCF Profiler results provides all of the required information to be able to fill out the diagram below.

You can reference the diagram for the *sobel_operator* above for a template on how to fill out the values. The inclusive and exclusive values for the sharpen filter have been provided as a starting point.

Question 8

Fill out the hierarchy diagram below.

- Insert all of the function names.

Tip: Do not get caught up matching exclusive % values to block sizes.

- Insert the exclusive percentages for all functions by filling in the e____% fields underneath the function names filled in from the previous task.
- Insert the inclusive percentage for *sobel_operator*.

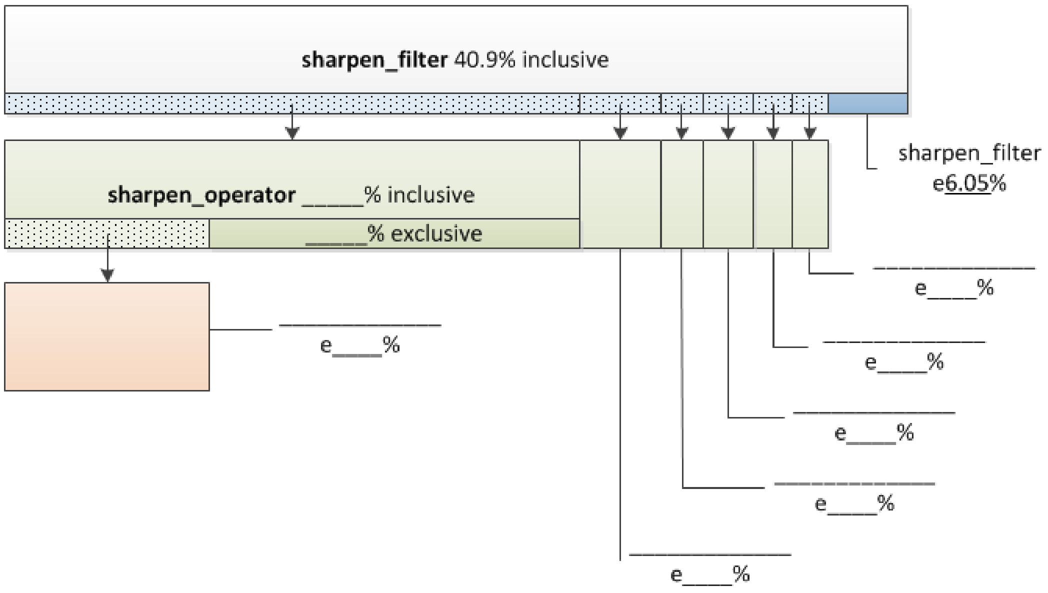


Figure 11-32: Sharpen Filter Exercise

Question 9

What function is the next largest consumer of the exclusive execution time?

____

____

____

Question 10

What is the difference between exclusive and inclusive?

Question 11

Why do all of the inclusive values shown in your results add up to more than 100% and all of the exclusive values add up to 100%?

Question 12

Why are there not any MACROs shown in the TCF Profiler results? (MACROs are used frequently in this project.)

Question 13

Why are the exclusive and inclusive values of the function *rgb_2_gray* equal?

Summary

You profiled several software algorithms and performed a more detailed analysis of one of these that may be a candidate for hardware implementation.

Answers

Answers listed represent sample solutions only. Your results may differ depending on the version of the software, service pack, or operating system that you are using.

1. What are the child functions of *main*?

   sobel_filter, *sharpen_filter*, *rgb_2_gray*, xlnkAllocBufInternal (if running Linux) and *dummyFill*.

2. What are the parent and child functions of *sobel_operator*?

   The parent is *sobel_filter* and the only child is *window_getval*.

3. How many times are x_weight and y_weight calculated every time *sobel_operator* is called?

   Nine times. The WINDOW_HEIGHT and WINDOW_WIDTH are each 3 respectively, resulting in 3*3 = 9.

4. How many times will *sobel_operator* be called for the image of size 1920 * 1080?

   sobel_operator will be called 1920 * 1080 times because it must perform its calculation for each and every pixel in the image. This is 2,073,600 times that the function is called.

5. How many times will x_weight and y_weight be calculated for each image passed through *sobel_filter*? **Hint:** Review the answers to the previous two questions.

   x_weight and y_weight are each calculated (2,073,600 * 9) = 18,662,400 times. Combined, this is nearly 40 million operations that this function alone needs for an image of 1920 * 1080.

6. What does the program *main()* do?

   It creates image buffers, then fills the color image with data, converts this to grayscale, and performs a sharpen and edge detect on the grayscale image. This is a computationally taxing sequence of processes, especially for large images.

7. What are the % Exclusive values for the functions *sobel_filter()*, *sharpen_filter()*, and *rgb2gray()*? **Note:** Your values may differ.

   - *sobel_filter()* ~6.24%
   - *sharpen_filter()* ~3.42%
   - *rgb2gray()* ~3.42%

8. Fill out the hierarchy diagram below. **Note:** Your values may differ.

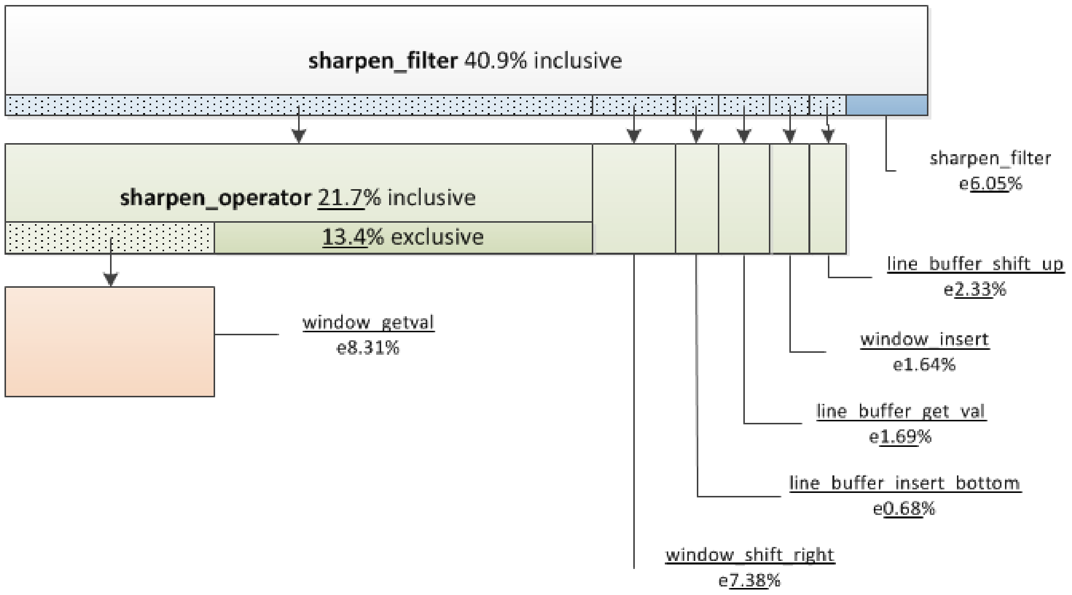

sharpen_filter 40.9% inclusive
sharpen_operator 21.7% inclusive
13.4% exclusive
sharpen_filter e6.05%
line_buffer_shift_up e2.33%
window_getval e8.31%
window_insert e1.64%
line_buffer_get_val e1.69%
line_buffer_insert_bottom e0.68%
window_shift_right e7.38%

Figure 11-33: Sharpen Filter Exercise - Completed

9. What function is the next largest consumer of the exclusive execution time?

   sharpen_operator utilizes ~13 exclusive and ~ 21% inclusive.

10. What is the difference between exclusive and inclusive?

    Exclusive only considers the execution time for a given function in its own level. Inclusive considers the execution time of a function and all of its child function calls.

11. Why do all of the inclusive values shown in your results add up to more than 100% and all of the exclusive values add up to 100%?

    The exclusive values represent the actual time taken to run the function. As the execution times of all functions in an application add up to the total execution time of an application, this is 100%. The inclusive values take into account the time taken to run that function and all child function calls. When the inclusive values are added together, the execution times of lower level functions are added multiple times, resulting in a value that is greater than 100%.

12. Why are there not any MACROs shown in the TCF Profiler results? (MACROs are used frequently in this project.)

    MACROs are unrolled and inserted into the code by the preprocessor—meaning that wherever a MACRO is called, its actual function is determined during preprocessing and inserted into the code in place of the MACRO call. A function call, unless it is forced inline, will be a separate call that the TCF profiler can analyze.

13. Why are the exclusive and inclusive values of the function rgb_2_gray equal?

    This is a leaf node and calls no other functions.