# Profiling and Performance Improvement

XILINX.

## Objectives

> **After completing this module, you will be able to:**

- Describe what profiling is and how it works
- Use the SDK profiling perspective
- Use profiling reports to evaluate software efficiency
- Discuss software tradeoffs to hardware
- Describe the function of the gprof tool
- List methods of improving performance

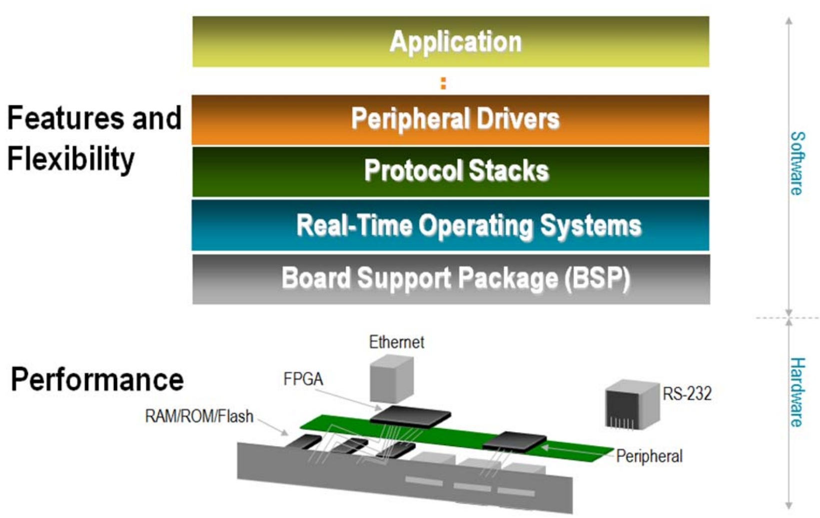
## Outline

- > *Introduction*
- > Software Profiling
- > Profiling in SDK
- > Performance Improvement
- > Summary



XILINX

## Embedded Systems






XILINX

# Hardware and Software Partitioning

- Determine the software "critical path" by profiling
  - Profiling measures where the CPU is spending its cycles on a function-by-function or task-by-task basis
  - Similar to timing analysis in hardware
  - Informs the system designer which software routine may be a candidate to hardware-accelerate
- Functions can be rewritten to improve efficiency in a number of ways
  - Implementation in assembly code rather than C
  - Writing faster C code, for example limit pointer use

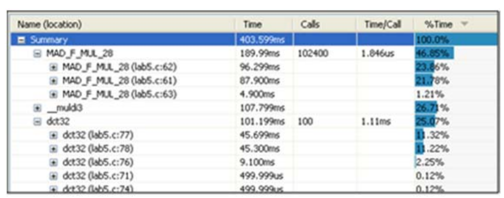




XILINX.

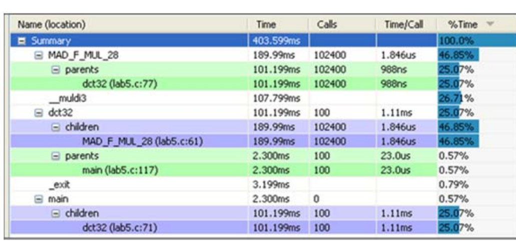
# What is Profiling?

- Profiling is an analysis of software performance
  - Where routine time is being spent
  - How many times functions are being called
  - Included tool in SDK
  - Which algorithms to consider moving to hardware
- Results in two useful formats



| Name (location) | Time | Calls | Time/Call | %Time |
|---|---|---|---|---|
| Summary | 403.599ms | | | 100.0% |
| MAD_F_MUL_28 | 189.99ms | 102400 | 1.846us | 46.85% |
| MAD_F_MUL_28 (lab5.c:62) | 96.299ms | | | 23.86% |
| MAD_F_MUL_28 (lab5.c:61) | 87.900ms | | | 21.78% |
| MAD_F_MUL_28 (lab5.c:63) | 4.900ms | | | 1.21% |
| __muldi3 | 107.799ms | | | 26.71% |
| dct32 | 101.199ms | 100 | 1.11ms | 25.07% |
| dct32 (lab5.c:77) | 45.699ms | | | 11.32% |
| dct32 (lab5.c:78) | 45.300ms | | | 11.22% |
| dct32 (lab5.c:76) | 9.100ms | | | 2.25% |
| dct32 (lab5.c:71) | 499.999us | | | 0.12% |
| dct32 (lab5.c:74) | 499.999us | | | 0.12% |

**Samples per function: How much time is spent in each routine**



| Name (location) | Time | Calls | Time/Call | %Time |
|---|---|---|---|---|
| Summary | 403.599ms | | | 100.0% |
| MAD_F_MUL_28 | 189.99ms | 102400 | 1.846us | 46.85% |
| parents | 101.199ms | 102400 | 988ns | 25.07% |
| dct32 (lab5.c:77) | 101.199ms | 102400 | 988ns | 25.07% |
| __muldi3 | 107.799ms | | | 26.71% |
| dct32 | 101.199ms | 100 | 1.11ms | 25.07% |
| children | 189.99ms | 102400 | 1.846us | 46.85% |
| MAD_F_MUL_28 (lab5.c:61) | 189.99ms | 102400 | 1.846us | 46.85% |
| parents | 2.300ms | 100 | 23.0us | 0.57% |
| main (lab5.c:117) | 2.300ms | 100 | 23.0us | 0.57% |
| _exit | 3.199ms | | | 0.79% |
| main | 2.300ms | 0 | | 0.57% |
| children | 101.199ms | 100 | 1.11ms | 25.07% |
| dct32 (lab5.c:71) | 101.199ms | 100 | 1.11ms | 25.07% |

**Function call graph: Which routine call, which function, and how many times**





XILINX.

# Software Profiling



## How Does Profiling Work?

- Hardware/software intrusive
  - Requires a hardware timer
  - Requires a dedicated area in memory
  - Executable is modified with profiler routines
- A dedicated hardware timer interrupts the processor at a fixed interval
  - The interrupt routine keeps track of the program counter at each interrupt
  - A histogram of PC locations is kept in profile RAM
  - Interrupt interval time is programmable
- Every function call in the software application is annotated by the compiler to track which functions are being called

## Coding Style Can Impact Profiling

- **Effective profiling is based on how much time is spent in functions, and how often they are called**
  - If your code is just a fall-through main, profiling is not useful because 100 percent of execution time will be in *main( )* with no calls to other functions
  - Carefully architect the application with a structured architecture by using functions
  - Complier does not consider *macros* as functions – the macro will be expanded and treated as in-line code
  - Separate algorithms logically into functions that will help you analyze the flat profile view
  - Think ahead when architecting code—Is this algorithm a candidate for implementing in programmable logic?




## Profiling Procedure

- **Set profiling for the BSP**
  - Enable software intrusive profiling
  - Enable the -pg option
- **Set profiling for the application**
  - Enable the compiler for profiling with the –pg option
  - Configure the profiler memory
  - Set the interrupt frequency and *bin* value
- **Compile, link, and generate the ELF executable**
- **Download the executable into a hardware or software simulator**
- **Run the software application until completion or for an "amount of time"**
- **Execute the GNU *gprof* tool to view the generated profile report**

# *Profiling in SDK*

XILINX.



## Configuring the Software Platform Settings

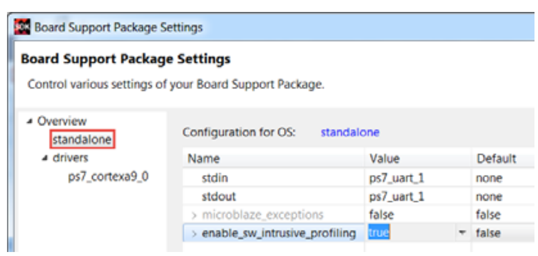
> **Select Xilinx Tools > Board Support Package Settings**

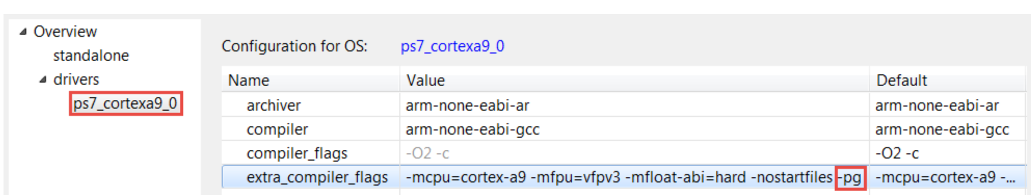
> **Select standalone**

  >> Enable software profiling

> **Select ps7_cortexa9_0**

  >> Add -pg to the Value column for the extra_compiler_flags option








XILINX.

# Profile Configuration: Create a Run Configuration

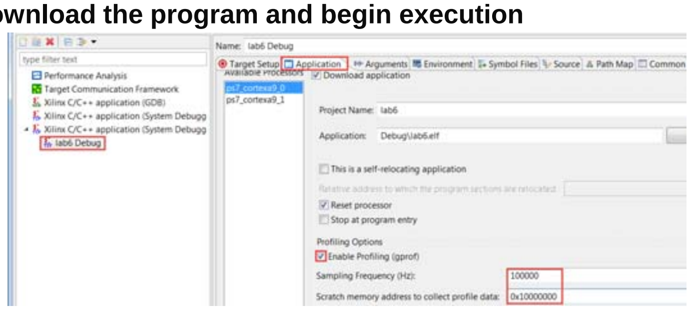
- **If any of the embedded design resides in programmable logic, download the bitstream to the programmable logic**
  - Select Xilinx Tools > Program FPGA
- **Select Run > Run Configurations and create a new configuration**
  - Give appropriate name
  - Select the elf file that was compiled with –pg




XILINX.

# Set Profile Option in Run Configuration

- **In the Application tab**
  - Enable profiling
  - Set the sampling frequency at which the timer will interrupt
    - Higher speed will require more memory but will give a finer resolution
  - Set the location of RAM that the profiler can use
    - make sure that the software application is not using this memory
- **Click Run to download the program and begin execution**





© Copyright 2018 Xilinx

XILINX.

# Profiling Reports

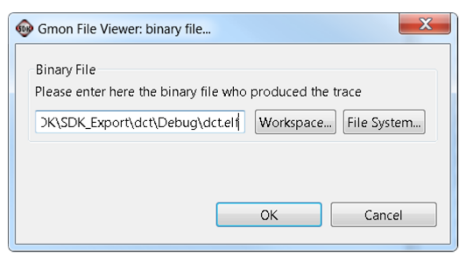
- Profile scratch memory is populated with statistics while the program is executing
  - Intrusive profiling routines and the fixed interval timer interrupt use this memory
  - Stored in *gmon.out* upon completion or execution halt
- The *gprof* tool is launched by the tools after execution completion
  - gprof reads *gmon.out* and assembles the information into a user configurable report

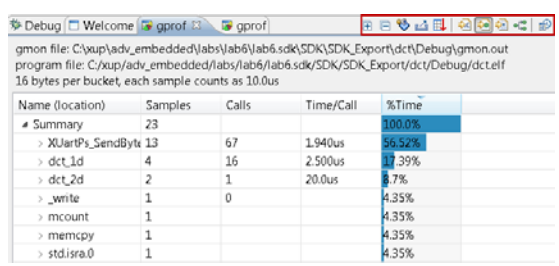




# Viewing Profiling Reports: Launching *gprof*

- **Double-click *gmon.out* to launch *gprof***
- **Point to executable ELF; usually selected by default**
- ***gprof* report launches**
- **Report toolbar control report options and view capabilities**
  - Sort samples per file
  - Sort samples per function
  - Sort samples per line
  - Display function call graph
  - Switch sample/time



## Profiled Output in SDK


| Name (location) | Samples | Calls | Time/Call | %Time |
|---|---|---|---|---|
| Summary | 23 | | | 100.0% |
| ?? | 23 | | | 100.0% |
| XUartPs_SendE | 13 | 67 | 1.940us | 56.52% |
| dct_1d | 4 | 16 | 2.500us | 17.39% |
| dct_2d | 2 | 1 | 20.0us | 8.7% |
| _write | 1 | 0 | | 4.35% |
| mcount | 1 | | | 4.35% |
| memcpy | 1 | | | 4.35% |
| std.isra.0 | 1 | | | 4.35% |

**1: Sort Samples per File**

| Name (location) | Samples | Calls | Time/Call | %Time |
|---|---|---|---|---|
| Summary | 23 | | | 100.0% |
| XUartPs_SendByte | 13 | 67 | 1.940us | 56.52% |
| dct_1d | 4 | 16 | 2.500us | 17.39% |
| dct_2d | 2 | 1 | 20.0us | 8.7% |
| _write | 1 | 0 | | 4.35% |
| mcount | 1 | | | 4.35% |
| memcpy | 1 | | | 4.35% |
| std.isra.0 | 1 | | | 4.35% |

**2: Sort Samples per Function**

| Name (location) | Samples | Calls | Time/Call | %Time |
|---|---|---|---|---|
| Summary | 23 | | | 100.0% |
| XUartPs_SendByte | 13 | | | 56.52% |
| dct_1d (??:-1) | 4 | | | 17.39% |
| dct_2d (??:-1) | 2 | | | 8.7% |
| _write (??:-1) | 1 | | | 4.35% |
| mcount (??:-1) | 1 | | | 4.35% |
| memcpy (??:-1) | 1 | | | 4.35% |
| std.isra.0 (??:-1) | 1 | | | 4.35% |
| Xil_In32 (??:-1) | 0 | | | 0.0% |
| Xil_Out8 (??:-1) | 0 | | | 0.0% |

**3: Sort Samples per Line**

| Name (location) | Samples | Calls | Time/Call | %Time |
|---|---|---|---|---|
| Summary | 23 | | | 100.0% |
| XUartPs_SendByte | 13 | 67 | 1.940us | 56.52% |
| dct_1d | 4 | 16 | 2.500us | 17.39% |
| dct_2d | 2 | 1 | 20.0us | 8.7% |
| _write | 1 | 0 | | 4.35% |
| mcount | 1 | | | 4.35% |
| memcpy | 1 | | | 4.35% |
| std.isra.0 | 1 | | | 4.35% |
| IRQInterrupt | 0 | 0 | | 0.0% |

**4: Display Function Call Graph**

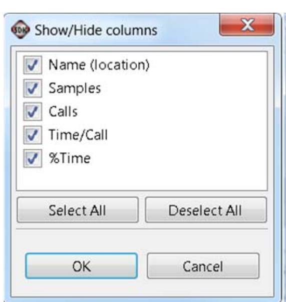


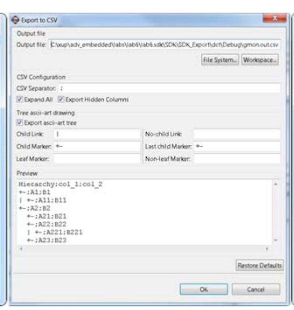


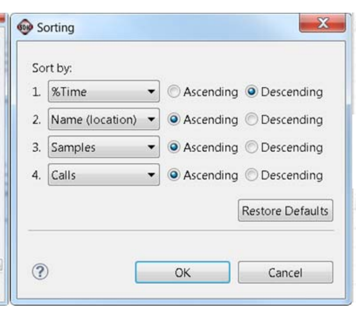
XILINX.

## Profiling Report Options


> **Gprof report options allow report view flexibility and export**




| Name (location) | Samples | Calls |
|---|---|---|
| Summary | 23 | |
| XUartPs_SendByte | 13 | 67 |
| dct_1d | 4 | 16 |
| dct_2d | 2 | 1 |
| _write | 1 | 0 |

| Name (location) | Time | Calls |
|---|---|---|
| Summary | 230.0us | |
| XUartPs_SendByte | 130.0us | 67 |
| dct_1d | 40.0us | 16 |
| dct_2d | 20.0us | 1 |
| _write | 10.0us | 0 |

1. Show/hide columns
2. Export to CSV
3. Sorting
4. Switch time<>Samples





XILINX.

# *Performance Improvement*

XILINX



## Task Implementation Decision

> **Keep it in software**
- Not in critical path
- Enough "free" cycles
- Easier to code in software than in hardware
  - Uses math library functions
- NEON co-processor
  - Supports integer vector operations
  - Single floating-point operations

> **Move to hardware**
- Programmable logic co-processor
  - Customized to user's needs
  - Excellent for iterative and pipelined processing
- Add soft core processor in PL
  - Both Cortex-A9 and MicroBlaze processors can co-exist in the SoC





XILINX

## Software to Programmable Logic

- **Slow software tasks can be accelerated by taking them to hardware**
  - Start with functions where the software spends most of its time
  - Consider the hardware implementation and if there is potential benefit implementing in hardware
- **Many mechanisms**
  - Dual-port block RAM
  - Custom AXI peripheral
  - Code optimization: use of macros, increasing compiler optimization
  - Enabling caching if (by default) it is turned OFF



XILINX

## Using Block RAM

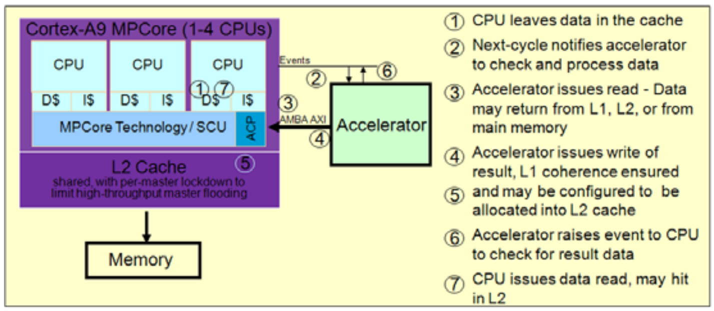
- **Leverage the dual-port nature of Xilinx block RAM**
- **Useful for data in block or frame format**
  - Video
  - 2D matrix maps
- **Advantages**
  - Low silicon overhead
  - Fast and deterministic latency




XILINX

# Enhanced Accelerator SoC Integration

- ARM MPCore: accelerator coherence port (ACP)
  - Sharing benefits of the ARM MPCore optimized coherency design
  - Accelerators gain access to CPU cache hierarchy
  - Compatible with standard un-cached peripherals and accelerators








XILINX

## *Summary*

XILINX

## Summary

- Profiling allows you to analyze the software and determine where the CPU's time is spent
- Profiling can help you rearrange or rewrite the code or even help you consider if a function can be targeted to hardware
- Enabling cache can improve performance
- The gprof tool is used to generate a profiling report from collected statistics
- A hardware timer and memory are required to use the profiling tool
- Profiling in SDK is provided by the Standalone BSP as a GNU service




XILINX

**Adaptable.**
**Intelligent.**

XILINX