Hands-On Workshop: Performance Optimization Hints and Tips for Power Architecture®

Curt Hillier  | Automotive MCU Applications Engineer

FTF-AUT-F0338

A P R . 2 0 1 4

Compiler and SuperTrace™ support provided by Green Hills Systems
Introduction, Agenda, and Course Objectives
In the Hands-On Workshop session today, we will do the following:

- **Review MPC57xx Architecture**

- **Target example software (executable code) to execute from Flash, RAM, and Internal Memory (I-MEM) to understand performance benefits**
  - Measure the cache-on / cache-off performance
  - Measure Flash performance and compare it to System RAM performance
  - Measure Internal Memory performance and compare this to Flash, System RAM options.

- **Configure compiler optimization levels in the Example Project**
  - Execute code and measure execution time
  - Record code size for each optimization level setting
  - Build knowledge of execution performance vs. code size trade-offs
Hands-on (continued):

- Measure execution time – how is this done in the MPC57xx?
- Build an “Target location performance” table and “Compiler optimization performance and code size” table as shown below:

Table 1. Target location performance

<table>
<thead>
<tr>
<th>Code Location</th>
<th>Exec Speed</th>
<th>Trade-off</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flash</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sys RAM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I-MEM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cache</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2. Compiler optimization performance and code size

<table>
<thead>
<tr>
<th>Optimization Level</th>
<th>Exec Speed</th>
<th>Code Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Onone</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-Ogeneral</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-Osize</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-Ospeed</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Introduction (3 of 3)

• If time allows, we will cover:

• Diagnostics
  – **Nexus Aurora Trace** to high speed external port or trace to internal memory:
    ▪ Program trace, Data bus trace, Instruction bus trace, DMA trace, Data Write / Read trace
  – **Performance Monitor registers**

• Profiling
  – Measure specific functions and how often they are executed. This is useful when optimizing as you can move frequently executed code to IMEM and frequently accessed Data to DMEM.

• IMEM Base Address remapping
Agenda

• Introduction:
  - MPC5777M Architecture Overview
  - Tool Overview (GHS MULTI IDE and SuperTrace Probe)
  - Basic operation for the workshop – compile code, load ELF, run, measure performance

• Hands-on Lab: Evaluate two methods to improve performance:
  1) utilize MPC5777M memory system features
  2) utilize an optimizing compiler

• Review a real world example
  - Combines both code relocation (to faster memories) and compiler optimization settings

• Summary and Conclusions
  - Performance increases by using faster memories and cache
  - Significant performance gains can be achieved by using optimizing compilers
Objectives

• Become familiar with MPC57xx CPU architecture, specifically the MPC5777M

• Learn how to use Performance Monitor counters to determine execution time and CPU load

• Study and implement 2 major performance optimization mechanisms:
  - 1) Compiler optimization levels
  - 2) Targeting instruction and data to MPC57xx memories and using cache.

• use linker settings to locate instruction code and data to various MPC57xx memories
Objectives

• At the end of this class, you will be able to:
  – Compile code (easy)
  – Change linker file settings
  – Set Optimization levels
  – Determine code size by reading a *.map file
  – Understand the basics of compiler optimization settings and their trade-offs
  – Relocate code to fast Instruction Memory (IMEM)
  – Remap IMEM base address to remove the need for far call patching
  – Configure Performance Monitor registers to measure elapsed time for a function
• Introduction:
  – MPC5777M Architecture Overview - memories
  – Describe Tool Overview (GHS MULTI IDE and SuperTrace Probe)
  – Describe Basic operation for the workshop – compile code, load ELF, run, measure performance
  – Describe Performance Monitor Features

• Use a simple EEMBC benchmark project, configure Compiler Optimization settings to improve performance

• Show results from a real world example

• Summary and Conclusions
Power Architecture® Tool Vendor Support

Debugger Support
- Lauterbach
- Green Hills Software
- pls
- PE micro

Compilers
- Yokogawa Digital Computer Corporation
- GAIO
- WindRiver

Calibration
- ETAS
- ATI
- Vector
- Yokogawa Digital Computer Corporation
Powertrain Market Trends

**Exponential MCU performance demands**
- Need 3-5x increase in performance to support software and emission filtering
- Consolidation of external ASICs into MCU
- On-chip DSP to meet tighter emission regulations
- High speed inter-processor communications, tightly coupled memory and Nexus Aurora allows 4x faster debug/trace capability

**Emerging safety and security standards**
- Enable ASIL-C or ASIL-D functional safety
  - Simplify ISO 26262 adoption
- Tamper detection and encryption to deter code tampering

**Lower Power**
- Power targets at 60% of previous generation MCUs
- Multi-core processing enables current reduction and simultaneous performance increase
Introducing the New Qorivva MPC5777M Multicore MCU

Performance & Architecture Efficiency
Unprecedented performance and integration for powertrain control within a power envelope of previous-generation MCUs and lower system costs.

Distributed Processing
On-chip I/O processor offloads primary cores and allows higher parallel throughput. Sigma delta converters allows customers to do advanced filtering using on-chip knock hardware.

Functional Safety & Security
Enables highest level of functional safety (ASIL-D) and offers strong on-chip security protection using HSM for SHE and TDM for tamper proofing.
Unprecedented Performance

• Industry’s highest performance
  - MPC5777M achieves 50% more frequency performance than Freescale’s benchmark-setting MPC5746M MCU
    ▪ 3x more performance than MPC5674F at 15% less power

• Built on 4th generation Power Architecture® e200z7 cores
  - MPC5777M has three cores operating at 300 MHz each
    ▪ One I/O core operating at 200 MHz helps to offload primary cores to increase overall throughput using parallel processing

• On-chip DSP, Ethernet, Zipwire, ∑ADCs and 4x faster Nexus Aurora debug
MPC5777M Key Features

• Two independent **300 MHz Power Architecture e200z7** computational cores
  – Single 300 MHz Power Architecture z7 lockstep
  – Delayed lock-step for ASIL-D safety
• Single I/O Core **200 MHz Power Architecture e200z4** core
• **8 MB Flash with ECC**
• **596 kB total SRAM with ECC**
  – 404 kB of system RAM (incl. 64k standby)
  – 192 kB of tightly coupled data RAM
• **10 ΣΔ ADCs** for knock detection, **12 SAR ADCs**
  – 84 total ADC channels
• **Generic Timer Module (GTM)** – 248 timer channels
• **Direct Memory Access (eDMA)** controller – 128 channels
**MPC5777M Core Complex**

### Core Details

<table>
<thead>
<tr>
<th>Core</th>
<th>Clock (MHz)</th>
<th>IMEM / DMEM</th>
<th>Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>e200z4</td>
<td>200</td>
<td>Yes</td>
<td>I-Cache only</td>
</tr>
<tr>
<td>e200z7</td>
<td>300</td>
<td>Yes</td>
<td>I / D Cache</td>
</tr>
<tr>
<td>e200z7</td>
<td>300</td>
<td>Yes</td>
<td>I / D Cache</td>
</tr>
</tbody>
</table>

- **Base Address:**
  - IMEM / DMEM: 0x4000_0000
  - Flash: 0x0100_0000

---

**Diagram Overview**

- **Core Complex**
  - Core 1: e200z7, I-Cache, D-MEM, I-MEM
  - Core 0: e200z7, I-Cache, D-MEM, I-MEM
  - Core 2: e200z4, I-Cache, D-MEM, I-MEM

- **Switches**
  - Slow Cross Bar Switch
  - Fast Cross Bar Switch

- **Memory**
  - System RAM (total 384KB for MPC5777M)
  - Flash (up to 8M)

---

**Notes**

- External Use
- Clocks (MHz): 200, 300
- IMEM / DMEM: Yes
- Cache: I-Cache only, I / D Cache
MPC5777M
Core Complex

• Clock domains

Memory accesses:
0 Wait State, fast speed
2 to 3 Wait States, 100 MHz
5 Wait States, 100 MHz
Tools For The Hands-On Workshop

Compilers, Debuggers, Hardware Evaluation Boards

- Debugger: Green Hills MULTI + SuperTrace probe
- Optimizing Compiler: Green Hills
- Freescale MPC5746MMB Motherboard
- Freescale MPC5777M-416DS Daughtercard
Evaluation Block Diagram

- Freescale Base Project + EEMBC code
- GHS MULTI Software debugger

GHS SuperTrace Probe

USB

Aurora

Freescale EVB Platform
HARDWARE: MPC5746MMB Motherboard and MPC5777M-416DS Daughter Card EVB
GHS Software Compiler Environment
```c
// Global frame - points into the base address of the big txbuf
MyEthernetFrame *frame;

#include "myVariables.h"

/**************************** Main *******************************/
int main()
{
    int i = 0;
    int portholeAddress = 0x400003C0;
    uint8_t * ptr;
    ptr = (uint8_t *) portholeAddress;

    MC_MODE_INIT(); /* Configure Clocks and Modules Via Mode Entry */

    for (i = 0; i < 10; i++)
    {
        SIUL2.GPDO[4].B.PDO = ~SIUL2.GPDO[4].B.PDO;
        SIUL2.GPDO[4].B.PDO = ~SIUL2.GPDO[4].B.PDO;
    }

MULTI>
```
Agenda

• Introduction:
  – MPC5777M Architecture Overview - memories
  – Describe Tool Overview (GHS)
  – Describe Basic operation for the workshop – compile code, load ELF, run, measure performance
  – Describe Performance Monitor Features

• Hands-on Lab: Evaluate two methods to improve performance:
  1) utilize MPC5777M memory system features
  2) utilize an optimizing compiler

• Show results from a real world example

• Summary and Conclusions
Base Project Review (1 of 2)

• **Definition:** Base Project refers to a device specific project provided by Freescale applications engineers. The Base Project contains initialization code for the device.

• **Purpose:** It assists users with bringing up their EVB and allows for easy integration of customer software.
Base Project Review (2 of 2)

- **Major components:**
  - `crt0.s`: assembly code to initialize stack, memory, BTB, cache. At completion, the code branches to main
  - `main()`: main routine
  - `mcu_init()`: Clock initialization code
  - Interrupt software, including prologue, epilogue
  - Optional: simple code for initializing peripherals
  - Linker Directive Files
Code Structure:

_EEMBC code has been merged into MPC5777M Base Project_
Utilizing the MPC5777M Memory System Features
Flow of Development

1) Start with Base Project provided by Freescale (*Today, we will use the EEMBC CoreMark example code*)
2) Compile and Link (*Linker file is provided in base project, review it briefly*). An ELF file will result.
3) Load the *.ELF file to the target MCU using a JTAG debug tool
4) Execute the code
5) Read the execution time monitor (performance monitor register)
6) Next, change optimization level, recompile, reload, and measure.
1. Start with Freescale MPC5777M Base Project
2. Compile project into *.ELF locatable executable file
3. Load *.ELF via debug tool to EVB
4. Execute Code
5. Read Results
1. Open the Project

- Double-click on mpc5777m-coremark.gpj
- MULTI Project Manager will open the top level project
- There will be 4 sub-projects
2. Build the ram.gpj Project

- Press F7, or select **Build → Build program ram.elf**
- Look for “Build successful” in the Status window
- Repeat for flash.gpj, imem.gpj, and cache.gpj
3. Review the *.map File

- Open ram.map using the MULTI Editor

- `.vle` size = 19,116 Bytes

- `.vle` is located in System RAM (starting at 0x4000_0274)

- `.data`, `.rodata`, and `.bss` are also in System RAM

- `.stack` is in DMEM

**QUIZ**: Can you see a way we could speed up the processing for the e200z4 use case?
For our example, only Core 2 (e200z4) will execute code. The example can be expanded to run on all cores.
4a. Start the Debugger and Connect To Target

- From MULTI, highlight the project you want to debug, and then select: Debug → Debug ram.elf
- Repeat for flash.elf, imem.elf, and cache.elf

**Note:** These will show up as Unconnected Executables until we move to the next step.
4b. Start the Debugger, and Connect To Target

- In the MULTI Debugger, select **Target → Connect**. When prompted for “**Use Which Connection / CPU?**” Select the following:
5. Verify We Have All 4 Cores Visible

```c
91
92    #else
93 1    MAIN_RETURN_TYPE main(int argc, char *argv[]) {
94 2    #endif
95 3
96 4    ee_u16 i,j=0,num_algorithms=0;
97 5    ee_u16 known_id=-1,total_errors=0;
98 6    ee_u16 seed_crc=0;
99 7    CORE_TICKS total_time;
100 8    core_results results[MULTITHREAD];
101 9    #if (MEM_METHOD==MEM_STACK)
102 10      ee_u8 stack_memblock[TOTAL_DATA_SIZE*MULTITHREAD];
103 11    #endif
104 12    /* first call any initializations needed */
105 13    portable_init(&results[0].port), &argc, argv);
106 14    /* First some checks to make sure benchmark will run ok */
```

Initializing `C:\ghs\comp_201354\mpserv -usb 0'.

MULTI>
6. Load and Execute the Code

- Right click on ram.elf, and select “Prepare Target...”. This will load the code and set the PC to _start_core2
- Click on the green arrow to execute the code
7. Read the final_score and final_time

• Select **View → Locals Variables...** This will bring up the Data Explorer window

• In the Data Explorer window, select **Edit → Add Variable**

• Type in “final_score” and “final_time”

- final_score contains an execution score (estimated CoreMark)
- final_time contains the execution duration
Instruction execution from Flash and System RAM is similar. Both Flash and System RAM have optimized interfaces to support e200z4 and e200z7 execution.

I-MEM and Cache have similar performance, about 3 times faster than execution from System RAM / Flash.
9. Conclusions for this exercise

<table>
<thead>
<tr>
<th>Code Location</th>
<th>final_score</th>
<th>final_time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flash (burst)</td>
<td>115.01</td>
<td>434</td>
</tr>
<tr>
<td>Flash (non-burst)</td>
<td>58.4</td>
<td>856</td>
</tr>
<tr>
<td>Sys RAM</td>
<td>101.58</td>
<td>492</td>
</tr>
<tr>
<td>I-MEM</td>
<td>280.87</td>
<td>178</td>
</tr>
<tr>
<td>Cache</td>
<td>292.9</td>
<td>170</td>
</tr>
</tbody>
</table>

- Utilizing the strengths of the MPC577xx Memory sub-system design is key to gaining best possible performance.

- MPC57xx Flash design supports:
  - Burst access to reduce single read access latency
  - Mini-cache and pre-fetch providing zero wait state access for data / instructions residing in cache

This provides Flash access performance on par with System RAM performance in the use case above. This performance is dependent on the software running in the device.
Flash Controller

- 256-bit read data bus + 64-bit write data bus

1) Read buffer and line pre-fetched supported by mini-cache, can provide 0 cycle read access for lines in cache.

2) Burst support provides up to 4x bandwidth on Flash reads
Simplified view of Burst vs. Non-Burst

- Non-burst Flash bandwidth:
  \[ BW_{\text{non-burst}} = 32 \text{ bit read} / 5 \text{ Wait states} \times 100 \text{ MHz} \]
  \[ BW_{\text{non-burst}} = 0.6 \text{ Gbps} \]

- Burst Flash bandwidth:
  \[ BW_{\text{burst}} = 256 \text{ bit read} / 5 \text{ Wait states} \times 100 \text{ MHz} \]
  \[ BW_{\text{burst}} = 5.1 \text{ Gbps} \]
Utilizing an Optimizing Compiler
Key Points for Compiler Optimization *Levels* and *Flags*:

1. Optimizing compilers are powerful tools and allow you to control code execution speed and code size via optimization levels and flags.

2. GCC supports a large number of compiler flags.

3. Compiler vendors and GCC have 'optimization levels' that abstract the large number of flags into higher level options - which are easier for everyday programmers to understand.
Compiler Optimizations

• From Green Hills documentation

• “Begin optimizing your code by selecting an optimization strategy, an option that groups several optimizations together into one setting. The strategies provided by Green Hills strike different balances among speed, size, and debugging ability that cover the needs for many different types of projects and development cycles”

Source: “MULTI: Building Applications for Embedded Power Architecture”
## GHS Optimization Levels

<table>
<thead>
<tr>
<th>Code size and speed</th>
<th>Compilation speed</th>
<th>Intended use</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>Not optimized</td>
<td>fastest when compilation speed is your highest priority</td>
</tr>
<tr>
<td>General</td>
<td>Optimal for general performance</td>
<td>Production, when speed and size are equally important</td>
</tr>
<tr>
<td>Size</td>
<td>Optimal size, moderately fast</td>
<td>Production, for size-restricted projects</td>
</tr>
<tr>
<td>Speed</td>
<td>Moderately small, optimal speed</td>
<td>Production, when speed is more important than size</td>
</tr>
</tbody>
</table>

*Source: Adapted from GHS “MULTI: Building Applications for Embedded Power Architecture”*
## EEMBC Core Mark Tests –
Execution Time and Code Size

<<Fill in this chart as part of the lab exercises>>

<table>
<thead>
<tr>
<th>Optimization Level</th>
<th>final_score (coreMark estimation)</th>
<th>final_time (ms)</th>
<th>Code Size (kB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case One</td>
<td>None</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Case Two</td>
<td>General</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Case Three</td>
<td>Size</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Case Four</td>
<td>Speed</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
1. Make 3 Copies of cache.gpj

• Make 3 copies of cache.gpj. Rename copies as shown below:
2. In the Top Level Project, Add New Projects

- Right click on mpc5777m-coremark.gpj, add **MULTI Project Files**
- Add: cache_Ogeneral.gpj, cache_Osize.gpj, and cache_Ospeed.gpj
3. Modify Optimization Level

• Right click on project, select Modify Build Options…

Documentation for **Optimization Strategy**

Specifies the high-level optimization strategy that MULTI uses when compiling your project. Permitted settings for this option are:

- **Maximum Debugging And No Inlining** (-Om3debug)
  - Disables inlining and all optimizations.
  - **Debugging ability** — excellent
  - **Code size and speed** — not optimized
Build Options Settings:

- Check to make sure you modified:
  - Output Filename
  - Object File Output Directory
  - Optimization Strategy
4a. Load Each New Project into the Debugger

• Then connect to e200z425 Core, Id2
• Load
• Execute
• Read Results: final_time and final_score
4b. Check `cache_general.elf` Example

```c
338 302  # Jump to GHS start code
339 303   # e_b1_start
340 304
341 305  # Jump to Main
342 306    e_b1 main
343 307
344 308  done:
345 309    e_b done
346 310
347 311
```

Source: `src\cxt0_core2_flash_cache.s`

Reset and halted
PAM attempt = 1
halting process... done.
MULTI>
5. Execute Tests and Record Results

- Repeat connect, load, execute, and measure for each of the new projects. Record the results.
### EEMBC Core Mark tests – Execution Time and Code Size

<<<Fill in this chart as part of the lab exercises>>>

<table>
<thead>
<tr>
<th>Optimization Level</th>
<th>final_score</th>
<th>final_time (ms)</th>
<th>Code Size (kB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case One</td>
<td>-Onone</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Case Two</td>
<td>-Ogeneral</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Case Three</td>
<td>-Osize</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Case Four</td>
<td>-Ospeed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Case Five</td>
<td>-Ospeed*</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* -Ospeed option with additional GCC flags: -Owholeprogram_libs, -Ounrollbig, -Ounroll, and -O1
## EEMBC Core Mark tests – Execution Time and Code Size

<table>
<thead>
<tr>
<th>Optimization Level</th>
<th>final_score</th>
<th>final_time (ms)</th>
<th>Code Size (kB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case One</td>
<td>-Onone</td>
<td>293</td>
<td>170</td>
</tr>
<tr>
<td>Case Two</td>
<td>-Ogeneral</td>
<td>399</td>
<td>125</td>
</tr>
<tr>
<td>Case Three</td>
<td>-Osiz</td>
<td>369</td>
<td>135</td>
</tr>
<tr>
<td>Case Four</td>
<td>-Ospeed</td>
<td>614</td>
<td>81</td>
</tr>
<tr>
<td>Case Five</td>
<td>-Ospeed*</td>
<td>732</td>
<td>68</td>
</tr>
</tbody>
</table>

* -Ospeed option with additional GCC flags: -Owholeprogram_libs, -Ounrollbig, -Ounroll, and -O1
Agenda

• Introduction:
  − MPC5777M Architecture Overview - memories
  − Describe Tool Overview (GHS Compiler, SuperTrace™ probe)
  − Describe Basic operation for the workshop – compile code, load ELF, run, measure performance
  − Describe Performance Monitor Features

• Hands-on Lab: Evaluate two methods to improve performance:
  1) utilize MPC5777M memory system features
  2) utilize an optimizing compiler

• Show results from a real world example

• Summary and Conclusions
# Real World Application – Execution Time and Code Size (MPC5777M e200z4)

<table>
<thead>
<tr>
<th>Optimization Level</th>
<th>Execution Time (ms)</th>
<th>Code Size (kB)</th>
<th>Compile Time</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Case One</strong></td>
<td>-Onone</td>
<td>37.4</td>
<td>603</td>
</tr>
<tr>
<td><strong>Case Two</strong></td>
<td>-Ogeneral</td>
<td>28.7</td>
<td>410</td>
</tr>
<tr>
<td><strong>Case Three</strong></td>
<td>-Osize</td>
<td>31.4</td>
<td>328</td>
</tr>
<tr>
<td><strong>Case Four</strong></td>
<td>-Ospeed</td>
<td>28.3</td>
<td>427</td>
</tr>
</tbody>
</table>
Higher Performance, Consider Optimization Options Used for MPC56xx Device Performance Benchmarking

-Ospeed + the following:

================================

-Owholeprogram_libs: compiles all libraries together with the rest of the program, and applies wholeprogram optimization on all source code at once

-Ounrollbig: Consider larger loops for unrolling.

-Ounroll: Loop Unrolling optimization.

-OI: Enables all the General Use optimizations together with two-pass inlining.
## Real World Application – Execution Time and Code Size (MPC5777M e200z4)

<table>
<thead>
<tr>
<th>Case</th>
<th>Optimization Level</th>
<th>Execution Time (ms)</th>
<th>Code Size (kB)</th>
<th>Compile Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case One</td>
<td>-Onone</td>
<td>37.4</td>
<td>603</td>
<td>Quick</td>
</tr>
<tr>
<td>Case Two</td>
<td>-Ogeneral</td>
<td>28.7</td>
<td>410</td>
<td>Moderate</td>
</tr>
<tr>
<td>Case Three</td>
<td>-Osize</td>
<td>31.4</td>
<td>328</td>
<td>Moderate</td>
</tr>
<tr>
<td>Case Four</td>
<td>-Ospeed</td>
<td>28.3</td>
<td>427</td>
<td>Moderate</td>
</tr>
<tr>
<td>Case Five</td>
<td>-Ospeed *</td>
<td>26.9</td>
<td>552</td>
<td>Moderate</td>
</tr>
<tr>
<td>Case Six</td>
<td>-Ospeed **</td>
<td>25.7</td>
<td>573</td>
<td>Moderate</td>
</tr>
</tbody>
</table>

* Compiler flags = -Owholeprogram_libs -Ounrollbig -O1 -Omax
** Compiler flags = -Owholeprogram_libs -Ounrollbig -O1 -OB -Omax
Execution Time: Application Loop Time
In Milliseconds

Execution time vs. Optimization level

Real App: ~30% improved
## Real World Application – Execution Time and Code Size (MPC5777M e200z4)

<table>
<thead>
<tr>
<th>Optimization Level</th>
<th>Execution Time</th>
<th>Code Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(ms)</td>
<td>(%)</td>
</tr>
<tr>
<td>Case One</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-Onone</td>
<td>37.4</td>
<td>---</td>
</tr>
<tr>
<td>Case Two</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-Ogeneral</td>
<td>28.7</td>
<td>23%</td>
</tr>
<tr>
<td>Case Three</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-Osise</td>
<td>31.4</td>
<td>16%</td>
</tr>
<tr>
<td>Case Four</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-Ospeed</td>
<td>28.3</td>
<td>24%</td>
</tr>
<tr>
<td>Case Five</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-Ospeed *</td>
<td>26.9</td>
<td>28%</td>
</tr>
<tr>
<td>Case Six</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-Ospeed **</td>
<td>25.7</td>
<td>31%</td>
</tr>
</tbody>
</table>

* Compiler flags = -Owholeprogram_libs -Ounrollbig -O1 -Omax
** Compiler flags = -Owholeprogram_libs -Ounrollbig -O1 -OB -Omax
Introduction:
- MPC5777M Architecture Overview - memories
- Describe Tool Overview (GHS)
- Describe Basic operation for the workshop – compile code, load ELF, run, measure performance
- Describe Performance Monitor Features

Hands-on Lab: Evaluate two methods to improve performance:
1) utilize MPC5777M memory system features
2) utilize an optimizing compiler

Show results from a real world example

Summary and Conclusions
SUMMARY and CONCLUSIONS
Summary

• MPC5777M Memory architecture:
  – Flash controller contains mini-cache – provides fast access
  – IMEM and DMEM provide zero cycle access at core clock speeds
  – I-Cache and D-Cache provide fast performance, dynamically allocated cache entries

• Optimizing compiler:
  – Levels allow user to align their optimization strategy to the compiled code
  – Additional GCC flags can be used to tune performance
  – User must trade-off execution speed with code size
<table>
<thead>
<tr>
<th>Session ID</th>
<th>Title</th>
<th>When</th>
</tr>
</thead>
<tbody>
<tr>
<td>FTF-AUT-F0235</td>
<td>Overview of the eTPU Engine Control Library for Qorivva 32-bit MCUs</td>
<td>Friday 10:30 AM</td>
</tr>
<tr>
<td>FTF-AUT-F0240</td>
<td>Unleash the Power of Qorivva MPC57xx 32-bit MCUs Using the Inter-Processor Communication (IPC) Interface</td>
<td>Thursday 3:00 PM</td>
</tr>
<tr>
<td>FTF-AUT-F0338</td>
<td>Hands-On Workshop: Performance Optimization Hints and Tips for Power Architecture® (Reserved Seat Required)</td>
<td>Tuesday 1:00 PM</td>
</tr>
<tr>
<td>FTF-AUT-F0339</td>
<td>Hands-On Workshop: Low-Power Techniques for Power Architecture® (Reserved Seat Required)</td>
<td>Tuesday 3:15 PM</td>
</tr>
<tr>
<td>FTF-AUT-F0344</td>
<td>An Introduction to the MPC57xx Nexus Aurora Interface</td>
<td>Wednesday 10:30 AM</td>
</tr>
<tr>
<td>FTF-AUT-F0345</td>
<td>MPC57xx e200zx Core Differences</td>
<td>Wednesday 11:30 AM</td>
</tr>
</tbody>
</table>
Application Note Resources

- **AN4802** Qorivva MPC57xx e200zx Core Differences
- **AN4591** Lauterbach MPC57xx Nexus Trace Tools
- **AN4812** Initializing the MPC5777M Clock Generation Module and Progressive Clock Switching Feature
- **AN4341** Using the Performance Monitor Unit on the e200z760n3 Power Architecture Core

- **AN** Initializing the MPC5746M (upon request)
- **AN** Initializing the MPC5746R (upon request)
Backup Information
Agenda

• Optimization Checklist
• Targeting code to IMEM
• Using Performance Monitor Counters
• Using Nexus trace to reconstruct code execution
Agenda

• Optimization Checklist
• Targeting code to IMEM
• Using Performance Monitor Counters
• Using Nexus trace to reconstruct code execution
Checklist (1 of 2)

Configuration:

• I-Cache and D-Cache are on
• Brach Target Buffer (BTB) is on
• Stack is in local D-MEM
• code & data are targeted efficiently (use D-MEM and I-MEM for frequently used data / code if possible)
• waitstate settings for FLASH and System RAM are correct
• compiler optimization level is set correctly
Checklist (1 of 2)

Diagnostics:

• confirm execution time meets expectation
• confirm cache is used (D-Cache and I-Cache), confirm data and instructions are in cache
• confirm code profile meets expectation – no functions consuming extra time
• use Aurora trace to diagnose hard to find code operation issues
• use performance monitor registers to verify cores are operating as expected (cycle count is the most commonly used)
Agenda

- Optimization Checklist
- Targeting code to IMEM
- Using Performance Monitor Counters
- Using Nexus trace to reconstruct code execution
e200z4 Core
All Code Exec From Flash

• Flash provides 8M access space
• I-cache, D-cache provide for fast execution
• D-MEM, I-MEM allow users to target frequently accessed code for faster execution
e200z4 Core – Relocate to IMEM

- Library function myLib.c relocated to IMEM.
- A “far call” is required since we support VLE only and the address jump is greater than 24-bits.
- Far call patching supported, but it takes about 14 instructions overhead.
ez4 Core - Relocated to IMEM. and IMEM Base Address remapped to Flash

- Library function myLib.c relocated to IMEM.
- IMEM Base Address remap is enabled, points to 0x010B_0000

See Also AN 4812
What is a Section?

• To assign data to a custom program section in C code, use the `#pragma ghs section` directive. This directive takes variables and functions that the linker would otherwise assign to the specified default section (such as `.data` or `.bss`) and assigns them to a custom section instead. In other words, it maps a type of data to a custom section.

• Example:

```c
#pragma ghs section text=".vle_imem"
```
MPC5777M Linker file – a simple example.

SECTIONS
{
   .bh            : {} > bootflash
   .isrvectbl ALIGN(0x1000) : {} > int_flash /* ISRVector */
   .xptn_vectors ALIGN(0x1000) : {} > . /* Exc Vector Table */

   .init          : {} > .
   .text          : {} > .   /* BookE Code */
   .vletext       : {} > .   /* VLE Code */
   .vle_imem      : {} > int_iram   /* VLE for IMEM */

   . . .
}

freescale™  External Use | 77
Far Call Patching

- Far call: MPC57xx devices are VLE only. VLE supports 24 bit addressing only. Consequently, if code jumps from System RAM (0x4000_0000), to IMEM (0x5200_0000), a far call error is issued. There is not enough bits to represent the address. Since the MPC57xx family does not have an MMU, the addresses cannot be remapped to support the relative jumps.

- The following code is executed every time we make a far call patch. This affects the number of instructions executed, and can reduce your expected performance increase.
Far Call Patching

- Adds 14 instructions
- For example: See the table below for Real world application performance with / without far call patch

<table>
<thead>
<tr>
<th>Case</th>
<th>Instructions</th>
<th>Execution Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>No IMEM</td>
<td>1.3 Million</td>
<td>16.9</td>
</tr>
<tr>
<td>IMEM, no Base Address remap (far call patching required)</td>
<td>1.5 Million</td>
<td>15.5</td>
</tr>
<tr>
<td>IMEM + Base Address remap (no far call patch required)</td>
<td>1.4 Million</td>
<td>14.6</td>
</tr>
</tbody>
</table>
How Do We Make a Jump to IMEM w/o Using Far Call Patching?

• The IMEM controller supports a base address remap (IBAPD= 1) which can be set to eliminate the need for a far call.

• The user also programs the Base address register with the desired remap address – in this case, it will be the Flash address of our .vletext section.
## Real World Application Use Cases

<table>
<thead>
<tr>
<th>Case</th>
<th>Flash</th>
<th>IMEM*</th>
<th>IMEM* IBAPD = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Flash only</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>2 Flash + IMEM</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>3 Flash + IMEM-R</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Cache enabled for all cases
<table>
<thead>
<tr>
<th>Far call patch used?</th>
<th>Instructions executed</th>
<th>Execution Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>No IMEM</td>
<td>Not required, all code executes from Flash</td>
<td>1,348,821</td>
</tr>
<tr>
<td>IMEM: No base address remap</td>
<td>Yes</td>
<td>1,467,409</td>
</tr>
<tr>
<td>IMEM: Base address remapped</td>
<td>No</td>
<td>1,439,984</td>
</tr>
</tbody>
</table>
Agenda

• Optimization Checklist
• Targeting code to IMEM
• Using Performance Monitor Counters
• Using Nexus trace to reconstruct code execution
High Level Usage of Performance Monitor, Checking a Function

- Execute main software and subroutines
- Clear performance monitor X count register
- Configure and Enable performance monitor X
- Run subroutine myFunction()
- Exit subroutine myFunction()
- Stop performance monitor X
- Read performance monitor X count register
Show How Performance Monitor Registers Are Used

- Set PMLCa0 = 0x00010000: count processor cycles
- Set PMLCa1 = 0x00020000: count instructions executed

- Sample code:

```c
/* get software control of debug resources */
asm ("e_lis r19, 0x00000000");
asm ("mtspr DBCR0, r19");  // clear EDM bit (ext debug ctrl = 0)

/* Configure Event - PMLCa0 = PM0 Processor Cycles */
asm ("e_lis r19, 0x0001");
asm ("mtpmr 0x90, r19");

/* Clear the counter PMC0 */
asm ("e_lis r19, 0x0000");
asm ("e_or2i r19, 0x0000");
asm ("mtpmr 0x10, r19");
```
Performance Monitor Results

- Processor clocks
- Instructions executed

Execution time = \frac{\text{processor clocks}}{\text{clock speed}}

- Exec time = 0xda77ae / 200 MHz
- Exec time = 71.6 ms
Agenda

• Optimization Checklist
• Targeting code to IMEM
• Using Performance Monitor Counters
• Using Nexus trace to reconstruct code execution
Nexus Trace Overview
Multi-Core Debug Overview

- Single debug interface for run control, trace, and measurement data.
- Supports IEEE 1149.1 (JTAG) and IEEE 1149.7 (cJTAG) for run control
  - TAP sharing used to allow access to all internal clients (processor cores, timer cores, and other clients)
- Trace based on the IEEE-ISTO 5001 Nexus debug standard
  - Nexus inherently supports multi-core debug information
  - Class 2+ or Class 3+ support on most clients
  - Parallel or high-speed Serial solutions depending on requirements
IEEE-ISTO 5001 Nexus Classes (MPC57xx support)

- **Class 1** Run Time Control
  - Start/stop code execution
  - Read/Write MCU registers / memory
  - Breakpoints
  - Single step instructions
  - Read Nexus device ID

- **Class 2** Dynamic Debug
  - Real time process/task ownership tracing
  - Trigger a nexus message on an event
  - Real time, non intrusive instruction trace

- **Class 3** Data Trace
  - Real time access of Registers / memory (read/write)
  - Real Time Data Trace (writes)
  - Optional features supported:
    - Real Time Data Trace (reads)
    - Data acquisition

- **Class 4** Advanced Debug
  - Watchpoint triggered trace event
  - Message over-run Control

Optional features supported:
- Data acquisition
- Watchpoint triggered trace event
- Message over-run Control
Nexus Trace Clients

Program, Data, and Ownership trace, plus Watchpoint messaging, and Data Acquisition provide a multitude of development capabilities.

- **e200zx cores**
  Power Architecture processing unit

- **Generic Timer Module**
  Bosch Timer unit

- **Enhanced Timing Processor**
  Freescale timing processor

- **Bus Master Clients**
  Allows accesses by bus masters: EtherNet, Direct Memory Access engines, Zipwire interprocessor bus
Multi-core trace

Program, Data, and Ownership trace, plus Watchpoint messaging, and Data Acquisition provide a multitude of development capabilities

- **Code Coverage**
  - Do all lines of code get executed?
  - Are all conditional branch paths executed?

- **Execution time**
  - Task, function execution times

- **Code Profiling**
  - What functions are called the most?
  - What functions consume the most time?

- **Task analysis**
  - What tasks are run, how often.

- **Cache Coverage**
  - Analysis of program/data caching to optimize performance

- **Core loading**
  - What is the loading of each core?
  - Can programs be segregated differently to improve the balance of performance.
Diagnostics (Lauterbach example)

• Profiling:
  - Determine which calls take the most execution time. For example:
Instruction Trace: Large granularity
Instruction Trace: Filtered Granularity

- e200z7
- Nexus 1
- Nexus 3

Instruction snoop

0x0009_88E8
0x0009_88F8

0x000B_BD30
0x000B_BD78
In the MPC57xx family, the Nexus Aurora serial trace port operates at 1.25 Gbps per lane.
Reconstructed Timeline:

- Using Nexus program trace messages, along with the code symbol information, the debugger software can reconstruct the time spent in each function.
- Nexus program trace messages contain information on each branch to / from functions.
Resources

- **AN4802** Qorivva MPC57xx e200zx Core Differences
- **AN4591** Lauterbach MPC57xx Nexus Trace Tools
- **AN4812** Initializing the MPC5777M Clock Generation Module and Progressive Clock Switching Feature
- **AN4341** Using the Performance Monitor Unit on the e200z760n3 Power Architecture Core
- **AN___** Initializing the MPC5746M
- **AN___** Initializing the MPC5746R
Making the World a Smarter Place.