XDS560 Trace
Advanced Use Cases for Profiling

Daniel Rinkes
Texas Instruments
Agenda

• AET / XDS560Trace Overview
• Interrupt Profiling
• Statistical Profiling
• Thread Aware Profiling
• Thread Aware Dynamic Call Graph
Agenda

• AET / XDS560Trace Overview
• Interrupt Profiling
• Statistical Profiling
• Thread Aware Profiling
• Thread Aware Dynamic Call Graph
What is AET / XDS560 Trace

• XDS560 Trace Allows Cycle Accurate Logging of CPU Address Bus and Data Bus Activity Non-Intrusively, in Real Time

• Advanced Event Triggering (AET) is logic that allows us to smartly turn trace on in interesting locations and off in non-interesting locations so as to preserve the trace buffer for interesting data
XDS560 Trace Architecture

- CPU
- DSP
- XDS560T POD

Additional JTAG Emulation Pins

Trace & AET Jobs
Comparators
Compressor
Cycle counter

XDS560T POD
RECORDING UNIT

- Current Buffer Size: 224K
- Future: 64 MB

To/from host PC
Required Software and Hardware

- 60 Pin emulation header
- Target must support Trace (Full-Gem)
- Blackhawk USB 560
- XDS560T Trace Pod/Cable
- CCS 3.30 or higher
64x+ Device Support For Trace

<table>
<thead>
<tr>
<th>SUPPORT TRACE</th>
<th>DO NOT SUPPORT TRACE</th>
</tr>
</thead>
<tbody>
<tr>
<td>• C6455</td>
<td>• DM6443/DM6446</td>
</tr>
<tr>
<td>• C6488</td>
<td>• Any LC Device</td>
</tr>
<tr>
<td>• DM647/DM648</td>
<td></td>
</tr>
</tbody>
</table>
Agenda

• AET / XDS560Trace Overview
• Interrupt Profiling
• Statistical Profiling
• Thread Aware Profiling
• Thread Aware Dynamic Call Graph
Interrupt Profiling Overview

- Capture Program Address and Timestamp whenever the PC is within the Interrupt Vector Table
- Generate a cycle accurate picture of when each interrupt starts executing

Graphically display interrupt cycle accurate interrupt servicing frequency
Cycle Count: 102456

Trace Log

<table>
<thead>
<tr>
<th>PC</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00897C60</td>
<td>102456</td>
</tr>
<tr>
<td>0x00897C64</td>
<td>102457</td>
</tr>
<tr>
<td>0x00897C68</td>
<td>102458</td>
</tr>
<tr>
<td>0x00897C6C</td>
<td>102459</td>
</tr>
<tr>
<td>0x00897C70</td>
<td>102460</td>
</tr>
<tr>
<td>0x00897C74</td>
<td>102461</td>
</tr>
<tr>
<td>0x00897C78</td>
<td>102462</td>
</tr>
<tr>
<td>0x00897C7C</td>
<td>102463</td>
</tr>
</tbody>
</table>
Results
Agenda

• AET / XDS560Trace Overview
• Interrupt Profiling
• Statistical Profiling
• Thread Aware Profiling
• Thread Aware Dynamic Call Graph
Statistical Profiling

GOAL
Get a quick overall view of which functions in an application consume the most cycles

- Sampling every Program Counter in an application quickly consumes Trace Buffer Bandwidth, preventing analysis of the entire application
- We can eliminate this problem by only capturing a statistical sample of application execution
Statistical Profiling Overview

- The Program Address is sampled at regular intervals
- Statistical Analysis is performed on the captured samples
- As in any statistical analysis, the determinations made on the statistical sample can be related to the general population
Statistical Profiling - 2

- AET contains all of the hardware needed to capture trace samples at a specified interval
- Interval should be carefully chose so as not to coincide with a periodic function
- Application instrumentation can switch AET off in locations that are not of interest
Statistical Profiling - Results

- Comma Separated Value Format
- Sorted from most intensive functions to least
Agenda

• AET / XDS560Trace Overview
• Interrupt Profiling
• Statistical Profiling
• Thread Aware Profiling
• Thread Aware Dynamic Call Graph
Thread Aware Profiling

GOAL
Generate a cycle accurate execution graph of a Thread/Task based application
Solution

• Instrument the task/thread switch function to write the task/thread ID to a well known location (global variable)
  – Operating systems typically provide hooks to insert functions in this location

• Trace all of the writes to that location, and get a timestamp with each.
Agenda

• AET / XDS560Trace Overview
• Interrupt Profiling
• Statistical Profiling
• Thread Aware Profiling
• Thread Aware Dynamic Call Graph
Thread Aware Dynamic Call Graph

GOAL
Display a Thread Based representation of actual function execution in an application
Capturing the Data

• Thread/Task
  – Hook function writes address of task

• Function
  – Entry/Exit points instrumented with Mark 0/ Mark 1 instructions*
    • Mark 0 inlined at each function entry point
    • Mark 1 inlined at each function exit point
  – Trace captures each of these locations with timestamp
    * CGT 6.0.1 enables function hooks
Graphical Displays become impractical as the number of functions increase.

At right is a sample call graph displayed by the Guess graphing package.
GPROF Like Format

- Modeled after Unix GPROF
- Each Thread separated into it's own subsection
- Each function section contains only immediate callers and callees
### A Closer Look

**Thread:** 0x00828194

<table>
<thead>
<tr>
<th>index</th>
<th>excl_cycles</th>
<th>called</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>14491050</td>
<td>388/388</td>
<td>_DEC_tcp2PreProc [2]</td>
</tr>
<tr>
<td></td>
<td>14491050</td>
<td>388</td>
<td>_DEC_tcp2DeintUnpunctSoft3 [0]</td>
</tr>
<tr>
<td></td>
<td>32580</td>
<td>48/48</td>
<td>_dst0Isr [8]</td>
</tr>
<tr>
<td></td>
<td>12154</td>
<td>48/146</td>
<td>_edmaIsr [7]</td>
</tr>
<tr>
<td></td>
<td>9049502</td>
<td>389</td>
<td>_DEC_tcp2QuantizeSoft [1]</td>
</tr>
<tr>
<td></td>
<td>26587</td>
<td>49/146</td>
<td>_edmaIsr [7]</td>
</tr>
<tr>
<td></td>
<td>4598471</td>
<td>389</td>
<td>_DEC_tcp2PreProc [2]</td>
</tr>
<tr>
<td></td>
<td>14491050</td>
<td>388/388</td>
<td>_DEC_tcp2DeintUnpunctSoft3 [0]</td>
</tr>
<tr>
<td></td>
<td>9049502</td>
<td>389/389</td>
<td>_DEC_tcp2QuantizeSoft [1]</td>
</tr>
<tr>
<td></td>
<td>3590771</td>
<td>389/389</td>
<td>_varianceEstim [3]</td>
</tr>
<tr>
<td></td>
<td>86691</td>
<td>380/388</td>
<td>_COM_spoolPost [5]</td>
</tr>
<tr>
<td></td>
<td>12342</td>
<td>49/146</td>
<td>_edmaIsr [7]</td>
</tr>
<tr>
<td></td>
<td>3590771</td>
<td>389</td>
<td>_varianceEstim [3]</td>
</tr>
</tbody>
</table>
Future Display – Tree View

Thread: 0x828194

 Callers

_Dec_tcp2DeintUnpunctsoft3

_Callers

_Dec_tcp2QuantizeSoft

_Callers

_Dec_tcp2PreProc

_Callers

_varianceCst = 388/388

_Callers

_COM_spoolTask

_Calls = 388/388

_Calls

_SET task

_Calls

_COM_spoolPost

_Calls

_DEC_tcp2EdmaIsr

_Calls

Exclusive Cycles = 14491050

Future Display – Tree View

Technology for Innovators™
Questions?
The Results of the statistical profile tend to approximate those found using traditional profiling.
Advanced Event Triggering Overview
Use Case Examples

- **Simple**
  - Read/Write Data Location
  - Read/Write Data Range
  - Specific Value Read/Write to/from Specific Location

- **Medium**
  - Read/Write to location A, but only after executing instruction B

- **Complex**
  - Read/Write to location A, but only after executing instruction B at least 20 times

- **CPU Halt**
- **Interrupt**
- **Start/Advance Counter**
- **Stop/Reload Counter**
- **Start Trace**
- **Store Trace Sample**
- **End Trace**
- **External Trigger (EMU 0/1)**
- **State Transition**
AET Target Library

• What is it?
  – A Target Library to allow programming of AET resources from the target application

• Advantages over CCS AET Plug-in?
  – Context based control of where AET is enabled/disabled
  – Ability to reallocate AET resources on the fly
  – Some functionality not inherently contained in the plug-in

• Disadvantages
  – Some Application cycles consumed
  – Some Application footprint used
Use Case

• Consider: Thread based application crashing because of a suspected overflow of one of the task stacks.

• Would like to use AET to monitor the top of the task stacks and generate an interrupt when we get too close to the top.

• Problems
  – All of the threads are dynamically created at run time, so there’s no way to know where the top of the dynamically allocated stack will be located at load time.
  – There are numerous threads, and AET resources are consumed by ONE “Watchpoint In Range” job.
Solution

• Use the AET Target Library
• All Operating Systems (DSP/Bios, OSE, etc) have a thread switch function that occurs when switching tasks. The user can place code in this hook that gets executed every time a thread switches context.
• In this function, program AET to watch the top of the stack of the NEXT thread.
• Every subsequent time that the thread is switched, delete the old AET job, and program a new one.
Solution - 2

- What does this accomplish?
  - Reuse of resources
    - By removing the prior AET job, you are reclaiming the resources (trigger builder and comparators) that were used.
    - Additionally, AET is only watching the CURRENT stack at a given time. Thus we’re able to manage with the 4 data comparators that are available.
  - Dynamic Task Location
    - By programming AET at run time, we can know where the dynamically allocated tasks are placing their stack.
    - This method also works perfectly well for statically allocated tasks.