System-Level Cache Visibility and Optimization

SDO Applications

February, 2008
Agenda

- Cache Effect Analysis with XDS560 Trace on 64x+ Devices
- Cache Analysis with simulation tools
- System Level Cache Optimization
Agenda

- Cache Effect Analysis with XDS560 Trace on 64x+ Devices
- Cache Analysis with simulation tools
- System Level Cache Optimization
XDS560 Trace Hardware/Software Overview

- XDS560 Trace is invaluable for memory profiling & cache effect visibility

- Trace product is: -
  - Trace XDS560 Pod
  - Blackhawk USB560 emulator
  - Trace software in >= CCS 3.3
    - Recommend CCS 3.3 SR8 for latest Trace Software

- On your C64+ board you need: -
  - 60 pin header
  - Well designed signal interfaces
    - C6455DSK rev D has several ECNs to improve Trace data integrity

- Any C64+ “full-GEM” device supports Trace & AET e.g. C645x, TCI648x, DM64x
Extended XDS560 Trace Buffer Memory (EBM)

- Until recently, the XDS560 Trace Pod Only Supported 224K of compressed data
- Now, with EBM, it supports 64Mb.

<table>
<thead>
<tr>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>More data captured without any additional effort, which means more of an application can be profiled in a single run.</td>
<td>More processing time necessary to get the end result.</td>
</tr>
<tr>
<td>More time necessary just to decode the captured data into raw format</td>
<td>More time necessary just to decode the captured data into raw format</td>
</tr>
<tr>
<td>Processing data in text format is no longer practical. Files can be greater than tens of gigabytes.</td>
<td>Processing data in text format is no longer practical. Files can be greater than tens of gigabytes.</td>
</tr>
</tbody>
</table>

** EBM is not necessarily the solution to all profiling problems. Benefit still must be weighed against limitations. **
Memory/Profiling Use Cases

Use-cases
1. Event-based profiling
   • “where & how many L1P misses to ext-mem cacheable occurred in my app?”
2. Statistical profiling
   • Hotspot analysis – which functions take largest percentage of CPU load?
3. Thread-based Execution Log
   • Cycle accurate equivalent of BIOS Execution Log – shows which thread to focus on
4. Dynamic Thread-Aware Function Call-Graph profiling
   • Cycle distribution of functions in context of threads they run in

Fundamentals
• Leveraging 560Trace we have potentially zero target overhead and no real-time intrusiveness
  – No Need for large on-chip buffers or a transport mechanism to get the data off chip
• Implementation of above use-cases requires clever Advanced Event Triggering (AET) setup
  – Allows for more effective use of trace buffer
• Postprocess Trace results to achieve desired use-cases
  – Perl scripts. Leverage Visualization tools (DVT) where appropriate
Cache Event Analysis

- Trigger on precise events e.g. “L1P Read Miss, Hits External, Cacheable”
- In scenario below we capture the PC at each trigger then correlate it with function & filename.
- Highest miss-count is where to focus memory optimization efforts.
- Example below with H.264 Main Profile decode – event trigger “L1P Read Miss, Hits External, Cacheable” - moving the top “offenders” into internal memory saved ~ 10% total cycles!

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Function</td>
<td>Filename</td>
</tr>
<tr>
<td>2</td>
<td>------</td>
<td>------</td>
</tr>
<tr>
<td>3</td>
<td>H264MPVDEC_TI_slice_data_P</td>
<td>h264d_slice_p.c</td>
</tr>
<tr>
<td>4</td>
<td>H264MPVDEC_TI_calc_mwp_8x8_aflf</td>
<td>h264d_vdec_ti_mwp_l.c</td>
</tr>
<tr>
<td>5</td>
<td>H264MPVDEC_TI_sparse_idct4x4_addpred_mbaff</td>
<td>h264d_vdec_ti_sparse_idct4x4_addpred_mmbaff_p.sa</td>
</tr>
<tr>
<td>6</td>
<td>H264MPVDEC_TI_interpredict_Hp_N8x8</td>
<td>h264d_vdec_ti_interpredict_hp_n8x8_p.sa</td>
</tr>
<tr>
<td>7</td>
<td>H264MPVDEC_TI_decode_mmb_P</td>
<td>h264d_decode_mmb_p.c</td>
</tr>
<tr>
<td>8</td>
<td>H264MPVDEC_TI_aso_calc_mwp</td>
<td>h264d_aso_mwp.c</td>
</tr>
<tr>
<td>9</td>
<td>H264MPVDEC_TI_cal_vat_edge_sths</td>
<td>h264d_loopfilter.c</td>
</tr>
<tr>
<td>10</td>
<td>H264MPVDEC_TI_frame_delblk</td>
<td>h264d_loopfilter.c</td>
</tr>
<tr>
<td>11</td>
<td>H264MPVDEC_TI_interpredict_hp_N4x4</td>
<td>h264d_vdec_ti_interpredict_hp_n4x4_p.sa</td>
</tr>
<tr>
<td>12</td>
<td>H264MPVDEC_TI_residual_block_cavld</td>
<td>h264d_vdec_ti_cavldmb_p.sa</td>
</tr>
<tr>
<td>13</td>
<td>H264MPVDEC_TI_interpredict_hp_N4x4</td>
<td>h264d_vdec_ti_interpredict_hp_n4x4_p.sa</td>
</tr>
<tr>
<td>14</td>
<td>H264MPVDEC_TI_mb_header</td>
<td>h264d_mbheader.c</td>
</tr>
<tr>
<td>15</td>
<td>H264MPVDEC_TI_interpredict_hp_N4x4</td>
<td>h264d_vdec_ti_interpredict_hp_n4x4_p.sa</td>
</tr>
<tr>
<td>16</td>
<td>H264MPVDEC_TI_residual_block_cavld Chroma</td>
<td>h264d_vdec_ti_cavldmb_chroma_p.sa</td>
</tr>
<tr>
<td>17</td>
<td>H264MPVDEC_TI_slice_header</td>
<td>h264d_slice.c</td>
</tr>
<tr>
<td>18</td>
<td>H264MPVDEC_TI_recon_addr_calc_1mv</td>
<td>h264d_vdec_ti_recon_addr_calc_1.c</td>
</tr>
</tbody>
</table>
Which Events Should Be Profiled?

- There are *a lot* of events – which ones typically help most with application-level cache optimization?

- Total number of cycles should typically be close to CPU Execute Cycles + L1P Stalls + L1D Stalls + Bank/Access Conflict Stalls

<table>
<thead>
<tr>
<th>Events</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>- L1P read-miss, hits external</td>
<td>These misses are the main contributor to L1P stalls.</td>
</tr>
<tr>
<td>- L1D read miss, hits external, cacheable A/B</td>
<td>These types of misses are the two main contributors to L1D stalls. Misses to external memory hurt most so focus there first.</td>
</tr>
<tr>
<td>- L1D write miss, Write Buffer Full A/B</td>
<td></td>
</tr>
<tr>
<td>L1D non-cacheable events</td>
<td>Good way to see if you’ve forgotten to turn on MAR bits for your external memory range!</td>
</tr>
</tbody>
</table>

- Note: L1P Misses are always cached.
Unified Breakpoint Manager (UBM)

- UBM plug-in gives an interface to trigger on System Events
- In this case, we store a trace sample each time an L1D Read Miss Occurs
- UBM Available in CCS 3.3 SR8
Trace Based Statistical Profiling

- PC is sampled at some time interval
  - Value of ‘N’, sampling interval is programmable. Implemented via AET counters.
- Number of times the PC fell within a given function indicates relative amount of time spent in this function
- Results are visualized in percentage e.g. histogram, spreadsheet
- Useful for “Hot spot analysis”
  - More useful for System Profiling since not cycle-accurate
- Inherently captures memory effects precisely
  - Moving the top reported functions into fast on-chip memory can sometimes lower their % impact.

<table>
<thead>
<tr>
<th>Function name</th>
<th>File name</th>
<th>Times Encountered</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>H264MP/DEC/Ti_residual_block_caid</td>
<td>h264vdec_ti_cavidmb_p.sa</td>
<td>386</td>
<td>11.47%</td>
</tr>
<tr>
<td>H264MP/DEC/Ti_OMA_wait</td>
<td>h264vdec_qdma.c</td>
<td>386</td>
<td>6.00%</td>
</tr>
<tr>
<td>H264MP/DEC/Ti_decode_rmm_P</td>
<td>h264d_decodemmb_p.c</td>
<td>373</td>
<td>4.03%</td>
</tr>
<tr>
<td>H264MP/DEC/Ti_mb_header</td>
<td>h264d_mbheader.c</td>
<td>329</td>
<td>4.26%</td>
</tr>
<tr>
<td>H264MP/DEC/Ti_filterEdgeLuma_Str3210</td>
<td>h264d_edgeedge_genricLumaV_p.sa</td>
<td>323</td>
<td>4.18%</td>
</tr>
<tr>
<td>H264MP/DEC/Ti_aso_calc_mvp</td>
<td>h264d_aso_mvp.c</td>
<td>296</td>
<td>3.82%</td>
</tr>
<tr>
<td>H264MP/DEC/Ti_recon_mmm_P</td>
<td>h264d_reconmm_p.c</td>
<td>274</td>
<td>3.56%</td>
</tr>
<tr>
<td>H264MP/DEC/Ti_sparse_410_4_addpred_mmbf</td>
<td>h264d_tspars_addpred_mmbf_p.sa</td>
<td>229</td>
<td>2.97%</td>
</tr>
<tr>
<td>H264MP/DEC/Ti_filterEdgeHorLuma_Str3210</td>
<td>h264d_edgeedge_gerniecLumaH_p.sa</td>
<td>196</td>
<td>2.54%</td>
</tr>
<tr>
<td>H264MP/DEC/Ti_aso_read_coeff</td>
<td>h264d_aso_cavidmb.ic</td>
<td>196</td>
<td>2.53%</td>
</tr>
<tr>
<td>H264MP/DEC/Ti_calc_ext_edge_shs</td>
<td>h264d_loopfilter.c</td>
<td>186</td>
<td>2.43%</td>
</tr>
<tr>
<td>H264MP/DEC/Ti_idc14x4</td>
<td>h264d_idc14x4_p.sa</td>
<td>186</td>
<td>2.41%</td>
</tr>
<tr>
<td>H264MP/DEC/Ti_cavidMB</td>
<td>h264d_vdec_cavidmb.ic</td>
<td>183</td>
<td>2.37%</td>
</tr>
<tr>
<td>H264MP/DEC/Ti_interphighp_N8x8</td>
<td>h264d_vdec_interphighp_N8x8_p.sa</td>
<td>182</td>
<td>2.36%</td>
</tr>
<tr>
<td>H264MP/DEC/Ti_slice_data_P</td>
<td>h264d_slice_p.c</td>
<td>175</td>
<td>2.27%</td>
</tr>
<tr>
<td>H264MP/DEC/Ti_vmvdbiniK8r</td>
<td>mvmmvuiK8r</td>
<td>169</td>
<td>2.06%</td>
</tr>
</tbody>
</table>
Thread Aware Execution Log

- Show load of threads in an OS-based system [BIOS, OSE etc]
  - Helps focus optimization efforts on key threads
- Inherently captures system memory effects precisely
  - Cache Misses reflected in thread execution times
  - Moving code & data of most intensive threads into fast on-chip memory improves execution graph.
- Data Visualization Tool (Socrates)
  - zoom in/out, markers etc, Correlate thread address to thread name via symbol lookup, statistics
Statistical + Graphical View

Screen shot of a software application showing a graph and tables with data.
Thread Aware Dynamic Call Graph Profiling

• Goal: attain cycle-accurate function call-graph attributed by Thread

### Thread: tskProcess

<table>
<thead>
<tr>
<th>index</th>
<th>excl_cycles</th>
<th>called</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>51442035</td>
<td>460/460</td>
<td>_FIR_TI_filter [6]</td>
<td></td>
</tr>
<tr>
<td>51442035</td>
<td>460</td>
<td>_FIR_TI_gen [0]</td>
<td></td>
</tr>
<tr>
<td>1556097</td>
<td>460/460</td>
<td>_VOL_TI_amplify [7]</td>
<td></td>
</tr>
<tr>
<td>1556097</td>
<td>460</td>
<td>_scale [1]</td>
<td></td>
</tr>
<tr>
<td>206803</td>
<td>1</td>
<td>_thrProcessRun [2]</td>
<td></td>
</tr>
<tr>
<td>180638</td>
<td>462/462</td>
<td>_SCOM_getMsg [3]</td>
<td></td>
</tr>
<tr>
<td>142474</td>
<td>462/462</td>
<td>_SCOM_putMsg [4]</td>
<td></td>
</tr>
<tr>
<td>133975</td>
<td>460/460</td>
<td>_CHAN_execute [5]</td>
<td></td>
</tr>
<tr>
<td>37847</td>
<td>460/460</td>
<td>_UTL_stsStopFunc [12]</td>
<td></td>
</tr>
<tr>
<td>29969</td>
<td>461/461</td>
<td>_UTL_stsStartFunc [15]</td>
<td></td>
</tr>
<tr>
<td>26375</td>
<td>231/231</td>
<td>_UTL_stsPeriodFunc [16]</td>
<td></td>
</tr>
<tr>
<td>18275</td>
<td>231/231</td>
<td>_checkMsg [17]</td>
<td></td>
</tr>
<tr>
<td>1616</td>
<td>4/4</td>
<td>_SCOM_open [18]</td>
<td></td>
</tr>
</tbody>
</table>

### Thread: tskTxJoin

<table>
<thead>
<tr>
<th>index</th>
<th>excl_cycles</th>
<th>called</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1496824</td>
<td>1</td>
<td>_thrTxJoinRun [23]</td>
<td></td>
</tr>
<tr>
<td>132940</td>
<td>231/231</td>
<td>_SCOM_getMsg [3]</td>
<td></td>
</tr>
<tr>
<td>27600</td>
<td>230/230</td>
<td>_SCOM_putMsg [4]</td>
<td></td>
</tr>
<tr>
<td>1060</td>
<td>2/2</td>
<td>_SCOM_open [18]</td>
<td></td>
</tr>
</tbody>
</table>

• Contribution of each function in the context of calling thread.
• Inherently captures memory effects precisely
  - Can see which thread & function context to focus on
# Interpreting The Results

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>0</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>1</td>
<td>9049502</td>
<td>389/389</td>
<td>DEC_tcp2PreProc [2]</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></td>
</tr>
<tr>
<td>2</td>
<td>4598471</td>
<td>389/389</td>
<td>COM_spoolTsk [4]</td>
</tr>
<tr>
<td></td>
<td>4598471</td>
<td>389</td>
<td>DEC_tcp2PreProc [2]</td>
</tr>
<tr>
<td></td>
<td>14491050</td>
<td>380/388</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>3</td>
<td>2590771</td>
<td>389/389</td>
<td>DEC_tcp2PreProc [2]</td>
</tr>
<tr>
<td></td>
<td>3590771</td>
<td>389</td>
<td>varianceEstim [3]</td>
</tr>
</tbody>
</table>
How Does It Work?

- Compiler enables implementation of inline “function hooks” at the entry and exit of each function.
- Each “function hook” contains an instruction that tells trace to capture it, along with the timestamp.
- Thread Switches are marked with a write to a global variable. These writes are also traced.
- Post Processing Script recreates the entire function call graph.
## Pros and Cons of Profiling with XDS560 Trace

<table>
<thead>
<tr>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Zero / very-low intrusiveness</td>
<td>• only available on limited set of devices. NOT available on DM6446, 643x etc.</td>
</tr>
<tr>
<td>• Inherently captures memory effects precisely</td>
<td>• Need to build 60 pin header on to your board</td>
</tr>
<tr>
<td>• Powerful set of Memory Profiling use-cases already available</td>
<td>• Buffer Size Trade Off</td>
</tr>
<tr>
<td></td>
<td>– Too Large – Processing Time increases</td>
</tr>
<tr>
<td></td>
<td>– Too Small – Not Enough data for application visibility</td>
</tr>
<tr>
<td></td>
<td>• Smart Triggering with buffer size selection allows the correct mix</td>
</tr>
</tbody>
</table>
Agenda

• Cache Effect Analysis with XDS560 Trace on 64x+ Devices
• Cache Analysis with simulation tools
• System Level Cache Optimization
Basics

- Simulators exist for various TI chips
  - Examples
    - C641x, C671x simulator.
    - C6455
  - Two important variants
    - Device cycle accurate (C641x, C671x, C6455)
      - Accurate cycle profiling, runs slower.
    - Device Functional simulator (C641x, C671x)
      - Accurate event profiling, no system level cycle profiling, runs faster.
  - Good option for C64+ chips with only “mid-GEM” Emulation support (i.e. no AET/Trace)
    - DM643x, DM644x, DM646x, C642x, OMAP2430 etc
  - DM643x/C6424 Memory Optimization simulation support will be available in upcoming CCS 3.3 Service Pack
What Simulator to pick for memory optimization

- Device Functional Simulator runs faster and gives event counts
- Verify the improvements on device cycle accurate simulator by seeing improvements in overall cycles.
- Choices
  - If simulation speed is not a concern
    - Pick device Cycle accurate simulator
      - Note: For C64x+, all simulators are cycle accurate
  - If speed is important for overall profile flow
    - Use both. Do optimization iterations on functional, and validate them on the cycle accurate simulator, and if possible on the EVM/DSK.
Tools available on simulator for memory optimization

• Profile your code to find hotspots
  – CCS Profiler
    • Advantages
      – Function and range profiling. Large number of profile events available.
    • Disadvantage
      – Does not work well for threaded applications
  – Exclusive Multi Event Profiler
    • Advantages
      – Output available in a Microsoft excel sheet.
      – Works well for threaded applications.
    • Disadvantages
      – Profiles a maximum of only 12 events at a time.
      – Changing the default set of events is a bit tricky.

• Simulator Analysis
Using the Exclusive Profiler tool

1. Load Program
2. Profile Setup
3. Check This
4. Enable Profile, Run Application and Disable Profile
5. View the Data
## Exclusive Profiler Tool

<table>
<thead>
<tr>
<th></th>
<th>Function</th>
<th>cycle.Total</th>
<th>cycle.CPU</th>
<th>L1P.miss.summary</th>
<th>L1P.miss.summary_rate %</th>
<th>L1d</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>main_</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>mat_oprn.</td>
<td>884992</td>
<td>573440</td>
<td>49411</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>mat_xpose_oprn</td>
<td>284837</td>
<td>146876</td>
<td>293</td>
<td>50%</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Others</td>
<td>353</td>
<td>283</td>
<td>29</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Total</td>
<td>1170215</td>
<td>722621</td>
<td>98853</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Large differences between Cycle.Total and Cycle.CPU indicate potential system level effects like unoptimized cache.
# Exclusive Profiler Tool

<table>
<thead>
<tr>
<th></th>
<th>Function</th>
<th>File</th>
<th>Line no.</th>
<th>Size (bytes)</th>
<th>Start address (hex)</th>
<th>#times called</th>
<th>%coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>addi</td>
<td>instr.c</td>
<td>28</td>
<td>140</td>
<td>0x0000011d0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>add</td>
<td>instr.c</td>
<td>18</td>
<td>144</td>
<td>0x000001140</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>4</td>
<td>decode</td>
<td>decode.c</td>
<td>16</td>
<td>2080</td>
<td>0x000000920</td>
<td>2</td>
<td>69</td>
</tr>
<tr>
<td>5</td>
<td>divi</td>
<td>instr.c</td>
<td>91</td>
<td>188</td>
<td>0x000001598</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>get_index</td>
<td>utils.c</td>
<td>46</td>
<td>43</td>
<td>0x000079fc</td>
<td>18</td>
<td>100</td>
</tr>
<tr>
<td>7</td>
<td>init globals</td>
<td>main.c</td>
<td>17</td>
<td>92</td>
<td>0x00008ca0</td>
<td>21</td>
<td>100</td>
</tr>
</tbody>
</table>

Placing the code in internal vs. External Memory.
- The decision is a combination of
  - Frequency of calls
  - Code size
  - Coverage.
Profile Events on Simulator

• How optimized is your code for system (cache effects, L2 EMIF) vs. CPU core.
  – Measure the difference of cycle.CPU and cycle.Total.
  – If the difference is small, then optimize the algorithm.
  – If the difference is large, focus on optimizing the cache usage and the DMAs and other system level effects as applicable.

• Cache related events
  – Conflict Miss events.
    • Are some lines in cache getting thrashed.
  – % of data cache access becoming misses.
    • L1D Miss vs. total L1D access
  – % Distribution of L2 misses.
    • Events at L2 cache filtered by source/L1 or L2
    • L2 conflict misses.
    • The above two give a clue for the L2 behavior and who is causing the L2 misses. The COST of L2 read miss is particularly high.
  – % of L1D Misses hit in L2 vs. Going to External
    • L2.cache.miss.data.write
    • L2.cache.hit.data.write
    • L2.cache.miss.data.read
    • L2.cache.hit.data.read

• CPU spending idle time
  – CPU NOP and CPU idle events.
Exclusive Profiler Tool Limitations

- Max of 12 events can be profiled at a time.
  - Default events can be changed from:
    `<CCS_INSTALL_DIR>\plugins\generic\activities\coveragec6x.ini`.
  - These events should not be:
    - Non leaf events.
    - Summary events.
    - Non Cache events.
- Gives only Exclusive profile. Does not give inclusive profile data.
Getting Around Simulator Limitations (One Example)

• Simulating Peripherals not simple
  – Replace a peripheral with a DMA from/to external memory.
    • Link a few video frames in external memory
    • Transfer using DMA to Internal
    • Process frame
    • Transfer back to external memory
  – Correctly simulates cache behavior (assuming simulator accurately models cache coherence operations)
Agenda

- Cache Effect Analysis with XDS560 Trace on 64x+ Devices
- Cache Analysis with simulation tools
- System Level Cache Optimization
System-level Cache Optimization Guidelines

• Previous slides described techniques to profile cache effects
• This section describes how to perform cache optimizations given the profile data.

• Note
  – Does NOT cover cache coherency – see References instead
  – Does NOT cover low-level algorithm techniques e.g. cross-path stalls, bank conflicts, touch loops etc
    • Assume the algorithm developer has already considered these
Application Level Optimizations (1)

• Signal Processing Code
  – Control and data flow of DSP processing are well understood
  – More careful optimization possible
  ➢ Use IDMA/EDMA for streaming data into L1, L2 SRAM to achieve best performance
  • Allocation in L1 SRAM eliminates cache effects altogether
  • L1 SRAM typically very small hence use for only most critical code/data
    – Must also keep decent L1 cache size to ensure best performance of code/data in L2 & external memory
  • L2 typically larger but clocked at CPU/2
    – Cache coherence is automatically maintained (as opposed to external memory)
Application Level Optimizations (2)

- General Purpose Code
  - Straight-line code, conditional branching
  - Not much parallelism, execution largely unpredictable
  - Use L2 Cache (configurable between 32K → 256K)
    - Leaving L1/L2SRAM for performance-critical signal processing loops
  - Example: C6000 TCP/IP Stack Benchmark Application – 1 GHz C6455
    - Conclusion: ‘working set’ is > 32K hence need ≥ 64K L2 cache size for best performance

<table>
<thead>
<tr>
<th>L2 Cache Size</th>
<th>Frame Rate</th>
<th>Frame Size (bytes)</th>
<th>Throughput (Mbits/s)</th>
<th>CPU Load</th>
</tr>
</thead>
<tbody>
<tr>
<td>256K L2 Cache</td>
<td>Max</td>
<td>8192</td>
<td>93</td>
<td>17%</td>
</tr>
<tr>
<td>128K L2 Cache</td>
<td>Max</td>
<td>8192</td>
<td>93</td>
<td>17%</td>
</tr>
<tr>
<td>64K L2 Cache</td>
<td>Max</td>
<td>8192</td>
<td>93</td>
<td>21%</td>
</tr>
<tr>
<td>32K L2 Cache</td>
<td>Max</td>
<td>8192</td>
<td>93</td>
<td>38%</td>
</tr>
</tbody>
</table>
Avoiding L1P Conflict Misses

- Allocate functions contiguously in memory. Key since L1P is direct-mapped.

```
SECTIONS {
    GROUP {
        .text:function_1
        .text:function_2
    } > SRAM
    ...
}
```
Avoiding L1D Conflict Misses

- L1D – 2 way set-associative cache, 64-byte line size
- Above example exceeds number of ways, causes conflict misses
- Allocate arrays contiguously in memory to resolve
- Align arrays at cache-line boundary to maximize cache-line reuse

```
short in1 [N]; short other1 [N];
short in2 [N]; short other2 [N];
short w1 [N]; short other3 [N];
short w2 [N];
```

```
r1 = dotprod(in1, w1, N);
r2 = dotprod(in2, w2, N);
r3 = dotprod(in1, w2, N);
r4 = dotprod(in2, w1, N);
```
Avoiding L1D Capacity Misses

Example: Arrays exceed L1D capacity

```
for (i=0; i<4; i++)
{
    o = i * N/4;
    r1 += dotprod(in1+o, w+o, N/4);
    r2 += dotprod(in2+o, w+o, N/4);
    r3 += dotprod(in3+o, w+o, N/4);
    r4 += dotprod(in4+o, w+o, N/4);
}
```

- Technique is called ‘blocking’
  - Split up arrays into smaller portions that fit into cache
  - Above: ½ cache for in[], ½ cache for w[]
Avoiding Eviction by Interrupts

• In larger system, your core signal processing code may be (unexpectedly) evicted by e.g. a timer interrupt
• For truly cache-critical-performance code consider
  – Freezing cache e.g. in a BIOS program
    • BCACHE_setMode(BCACHE_L1P, BCACHE_FREEZE)
    • Forcefully retains the code in cache
  – Disabling pre-emption
    • TSK_disable(), SWI_disable(), HWI_disable()

• NOTE : disabling interrupts for long periods of time may adversely affect system functionality
Summary

• Cache optimization is a frequently requested topic

• First step is profiling – some techniques include: -
  – 560 Trace: powerful, cycle-accurate, but only available on certain devices
  – Simulation ATK: thread-aware profiling. Select up to 12 events

• Next step is optimization – application level techniques include: -
  – Careful choice of what goes in L1, L2 SRAM and L1, L2 cache sizes
  – Understanding mix of Signal Processing v. GPP-style code
  – Allocating functions contiguously in memory to avoid L1P conflicts
  – Allocating buffers contiguously in memory to avoid L1D conflicts
  – ‘Blocking’ – working on smaller buffers to fit within L1D cache
  – Forcefully retaining critical code in cache
References

• **Articles / Presentations**
  – “Optimizing Cache Performance on TMS320C6000 DSP Architectures”, O Sohm, TI Developer’s Conference 2004 [link]
  – “Simulating RF3 to Leverage Code Tuning Capabilities”, V Wan, P Lal, July ‘06 [link]
  – “Debugging Cache Coherence and Optimizing Cache Performance with New TMS320C64x™ DSP Tools”, O Sohm, TI Developer’s Conference 2006 [link]
  – “What should I know about cache coherence?”, Davinci Mediawiki, ongoing [link]
  – “TMS320C64x+ DSP Cache User’s Guide”, spru862a, Oct ’06 [link]

• **Products**
  – CCStudio v3.3 with 560Trace & Simulation ATK [link]
  – Trace-based Profiling use-cases [link]