Pipeline Stall Analysis

{{#switchcategory:MSP430=<McuHitboxHeader/>|C2000=<McuHitboxHeader/>|Stellaris=<McuHitboxHeader/>|TMS570=<McuHitboxHeader/>|MCU=<McuHitboxHeader/>|MAVRK=<MAVRKHitboxHeader/>|<HitboxHeader/>}}

Contents

Pipeline Stall Analysis

Overview

Pipeline Stall Analysis is a method of using XDS560 Trace to detect Program Address in an application that are repeatedly susceptible to pipeline stalls. This type of analysis provides only indications of where the stalls are occurring. Additional steps need to be taken to diagnose the root cause of the stall.

Methodology

Pipeline Stall Analysis data is captured with a simple PC/Timing Trace Data Capture. The captured data is then analyzed to find all Program Addresses in the execution trace where a stall was encountered, and report how many times each PC was executed when a stall occurred, what the average number of stall cycles were when a stall occurred, and what the maximum number of stall cycles was for any given single execution of an address.

Useful Tip

When tracing in this manner, we are not using any type of stall signals to detect a pipeline stall. Trace knows how many cycles every instruction will take to execute under ideal conditions. When it sees an instruction execute in more than the expected number of cycles, it reports those extra cycles as a Pipeline Stall. However, Trace has no concept of what might have caused that stall, so it is up to the developer to do more research in order to try and eliminate the stall.

The goal here is to identify locations that get executed many times AND have a high average number of stall cycles. We then want to further analyze those locations and see if we can reduce or eliminate the stalls at those locations. In this manner, we get the largest performance increase for our effort if we eliminate these types of stalls.

The types of stalls that will be detected here typically have a root cause related to either code/data placement or instruction ordering. Stalls that are identified in this manner are tightly correlated to the program address where they are located. This could potentially be dues to a cache conflict if an instruction that shares the same cache line always gets called shortly before this instruction. This could potentially be a case of poor instruction scheduling causing contention for a resource (typically, the compiler should avoid these cases. It might also point to a case where large amounts of data that exceed the size of the data cache are accessed in a ping-pong fashion.

Example

Consider an application with a function foo() that always calls another function bar(). If foo() and bar() happen to get linked at two locations in external memory that share the same cache lines, this can have a disastrous effect on the performance of an application. When foo is executed the first time, it is fetched from external memory and stored into cache. As it executes, it needs to call bar(), so bar() is fetched from external memory into cache, inadvertently evicting foo in the process. The execution of foo() never sees the benefits of cache, because whenever it gets called it is assuredly not already in cache because it was evicted by bar during it's previous call.

The solution in this case would be to group foo() and bar() together in an output section so that they are linked close enough together that they won't share the same cache line. The Pipeline Stall Analysis scenario will give a quick indication that this type of phenomenon is occurring.

Limitations

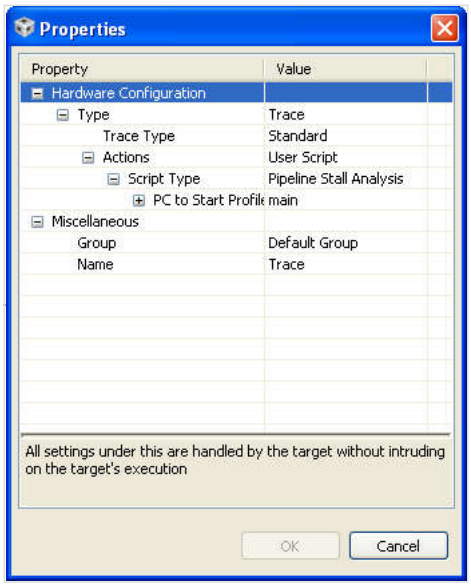
- The script does not analyze any instructions that do not have stalls associated with them. So, the instruction at location 0x80000000 might get executed 1000 times in the trace stream, but if on one of those executions there is a pipeline stall of 20 cycles, and on the other 999 there are no stalls, the script will report that address as having an average stall of 20 cycles. In actuality, this represents an *average* stall of .02 cycles per execution. This script focuses on determining the addresses where the most cycles are lost throughout the entire application.

Capturing Pipeline Stall Analysis Data

Configure trace in the Trace Control menu as desired. Typically, "Stop on Buffer Full" mode is used with Pipeline Stall Analysis, but it can also be used with "Circular Buffer" mode. If using an XDS560T, configure the desired trace buffer size. A larger buffer will capture much more data, but the data will take a longer time to process. A smaller buffer won't get nearly as much of the application, but will be post processed very quickly.

The Unified Breakpoint Manager (UBM) Plugin in Code Composer Studio can be easily configured to capture Pipeline Stall Analysis data

- From the Breakpoint window, create a new Trace job.
- Edit the properties of the job, and select Trace Type->Standard, Actions->User Script, and Script Type->Pipeline Stall Analysis as shown in the image.
- In the "PC to Start Profile" enter the PC where you want Trace to start capturing data. You can use a symbolic address here also, so if you want to start at the symbol "main", just enter **main**.
- Click OK to save the configuration and ensure that the job is enabled in the breakpoint window.
- Run the application. You should see trace data being captured in the Trace Display Window.



Processing Pipeline Stall Analysis Data

We will use the Stand Alone trace decoder to get the trace data for our script. The Stand Alone decoder is shipped with CCSv4 and is located at the following location <CCS4_INSTALL_DIR>\ccsv4\emulation\analysis\bin\td.exe. It can be called from the command line. Use the command td.exe -help to get details on the switches to be passed to the stand alone decoder. CCSv4 will create a file called XDS560_RecTraceData.bin in the <CCS4_Install_Dir>\ccsv4\emulation\analysis\bin\logs directory automatically whenever trace is captured. This is a copy of the raw data trace file. We will pass this file to the trace decoder to be processed.

Generating the func_info file

The first step in processing the trace data is to create a func_info.csv file from your .out file using the utility ofd6x.exe. This is a utility that is shipped with the code generation tools. The file that we need to generate will contain a list of all of the functions and filenames, along with the start, end, and length of each function. The utility ofd.exe is also shipped with the trace script package. It is located in the utils directory. Create the func_info file with the following command line

```
ofd6x --func_info <.out file name> > <.csv file name>
```

The <.out file name> parameter specifies the location and name of the .out file. The <csv file name> specifies the name of the file to be generated. You can choose whatever name you wish, but it is recommended to use the .csv extension. This file only needs to be regenerated when the .out file changes. One option is to have CCS execute this command as a post-build step every time you build your application so you will always have an up to date func_info file.

Decoder Prerequisites

In order for the Pipeline Stall Analysis application to be able to process the data, the following is required of the output of td.exe.

- The following fields must be included in the output (Default is All fields, which is fine)
  - Program Address
  - Cycles
  - Trace Status
- The timestamp must be in **Absolute** mode (-timestamp abs ***NOT THE DEFAULT***)
- The format must be in CSV_NO_TPOS_QUOTE form (-format CSV_NO_TPOS_QUOTE **This is the default**)

Command Line

One of the following commands can be used to process the trace data. Note that full or relative paths to each file supplied must be provided if all files are not in the current directory.

td.exe -bin XDS560_RecTraceData.bin -app <out file> <trace_decoder_options> | trace_pipeline_stall_analysis.exe -f <.csv file name> [-p <pipeline stall threshold>] [--noheader]

or

td.exe -bin XDS560_RecTraceData.bin -app <out file> <trace_decoder_options> | perl trace_pipeline_stall_analysis.pl -f <.csv file name> [-p <pipeline stall threshold>] [—noheader]

Specifying the pipeline stall threshold is optional, but this will filter the data significantly. Without it, there will be an entry in the output for every instruction that ever exhibits even a single stall. Using this threshold value will limit the output to instructions that have an *average* stall count greater than or equal to the threshold.

Sample Results Data

The results data is output through stdout in comma separated value form. A header line is prepended to the output, detailing the contents of each field. It can be supressed by using the —no_header option with the script.

```
PC,Access Count,Average Stall Cycles,Maximum Stall Cycles,Function Name,File
--,------------,--------------------,--------------------,-------------,----
0x800000c0,1,6,6,"ENC_cheInit","ENC_che.c"
0x800005bc,1,6,6,"SCH_init","C:/test/ExamplePackages/demos/event_trace/EA_c6488_tracews/Source/sch/src/SCH_init.c"
```

```
0x800005ec,1,6,6,"SCH_hwiInit","C:/test/ExamplePackages/demos/event_trace/EA_c6488_tracews/Source/sch/src/SCH_init.c"
0x800005f4,1,6,6,"SCH_hwiInit","C:/test/ExamplePackages/demos/event_trace/EA_c6488_tracews/Source/sch/src/SCH_init.c"
0x8000076c,1,6,6,"SCH_timerInit","C:/test/ExamplePackages/demos/event_trace/EA_c6488_tracews/Source/sch/src/SCH_timer.c"
0x80000cb4,1,6,6,"IRE_init","IRE_ire.c"
0x80000d20,1,5,5,"IRE_init","IRE_ire.c"
0x80000d50,1,5,5,"IRE_init","IRE_ire.c"
0x80001064,1,5,5,"DEC_vcpInit","DEC_VCP.c"
0x80001442,1,6,6,"DEC_vcpInit","DEC_VCP.c"
0x80001ad0,1,5,5,"DEC_vcpInit","DEC_VCP.c"
0x80001d56,1,5,5,"DEC_vcpInit","DEC_VCP.c"
0x80001e2c,1,5,5,"DEC_vcpInit","DEC_VCP.c"
0x80001fda,1,5,5,"DEC_vcpInit","DEC_VCP.c"
0x8000232a,5,6,6,"COM_spoolInit","COM_spool.c"
0x008563c0,8,6,6,"PDTCH_convenc3h","SIG_convEncPDTCH3h.c"
0x00862180,8,6,6,"ENC_PDTCH_mcs8","ENC_PDTCH_mcs8.c"
0x008621cc,8,6,6,"ENC_PDTCH_mcs8","ENC_PDTCH_mcs8.c"
0x0087c90e,8,4,4,"SCH_blockFill","C:/test/ExamplePackages/demos/event_trace/EA_c6488_tracews/Source/sch/src/SCH_blocks.c"
0x0087caac,8,4,6,"SCH_blockFill","C:/test/ExamplePackages/demos/event_trace/EA_c6488_tracews/Source/sch/src/SCH_blocks.c"
0x0087d070,8,4,6,"SCH_blockGenerate","C:/test/ExamplePackages/demos/event_trace/EA_c6488_tracews/Source/sch/src/SCH_blocks.c"
0x0087e5ae,1,6,6,"edmaInit","C:/test/ExamplePackages/demos/event_trace/EA_c6488_tracews/test/tools/edmaIsr.c"
0x0088036c,6,6,6,"TSK_create","src/tsk/tsk_crea.c"
0x00880a7a,1,5,5,"MEM_define","src/mem/mem_defi.c"
0x008824f4,1,5,5,"TSK_exit","src/tsk/tsk_exit.c"
0x00882a38,1,6,6,"TSK_startup","src/tsk/tsk_stup.c"
0x00884868,12,6,6,"COM_byte2bit","COM_c6x.c"
0x008889d4,1,6,6,"SCH_schedulerTsk","C:/test/ExamplePackages/demos/event_trace/EA_c6488_tracews/Source/sch/src/SCH_scheduler.c"
0x008889f0,1,6,6,"SCH_schedulerTsk","C:/test/ExamplePackages/demos/event_trace/EA_c6488_tracews/Source/sch/src/SCH_scheduler.c"
0x00888ba0,4,6,6,"SCH_schedulerTsk","C:/test/ExamplePackages/demos/event_trace/EA_c6488_tracews/Source/sch/src/SCH_scheduler.c"
0x00888bf6,4,4,6,"SCH_schedulerTsk","C:/test/ExamplePackages/demos/event_trace/EA_c6488_tracews/Source/sch/src/SCH_scheduler.c"
0x00889094,1,6,6,"SCH_schedulerTsk","C:/test/ExamplePackages/demos/event_trace/EA_c6488_tracews/Source/sch/src/SCH_scheduler.c"
0x0088942c,4,6,6,"SCH_schedulerTsk","C:/test/ExamplePackages/demos/event_trace/EA_c6488_tracews/Source/sch/src/SCH_scheduler.c"
0x0088a050,1,6,6,"COM_taskBenchUpdate","COM_bench_tsk.c"
0x0088cf40,7,5,5,"CACHE_waitInternal","csl_cacheWait.c"
0x0088d090,7,6,6,"CACHE_waitInternal","csl_cacheWait.c"
0x0088daac,1,6,6,"testSchTsk","../main.c"
0x0088daba,1,6,6,"testSchTsk","../main.c"
0x0088dad8,1,6,6,"testSchTsk","../main.c"
0x0088db04,1,6,6,"testSchTsk","../main.c"
0x0088db8e,1,6,6,"testSchTsk","../main.c"
0x0088dba8,1,6,6,"testSchTsk","../main.c"
0x0088e894,7,4,4,"CACHE_invAllL1p","csl_cacheL1.c"
0x0089baa8,1,6,6,"RTDX_Buffer_ReadCB","src/buffer/buffer2.c"
0x0089ee20,6,6,6,"SCH_msLoop","C:/test/ExamplePackages/demos/event_trace/EA_c6488_tracews/Source/sch/src/SCH_msLoop.c"
```

This data shows the set of instructions that have an average stall cycle of 4 cycles or greater.

The results can be piped to a file or to another script for further processing.

Trace Script Download

The script package can be downloaded at the following location https://www-a.ti.com/downloads/sds_support/applications_packages/trace_csv_scripts/index.htm

Note that in order to use the stand alone trace decoder, you must have version 3.0.0 of the scripting package or later.

Trace Script Frequently Asked Questions

Generic Scripting FAQS

- Q: Why do I sometimes see "UNKNOWN", "UNKNOWN" in the output for functions/filenames
  - A: The function/filename symbols are determined from the output of the ofd6x.exe (Object File Dump utility), which generates a list of functions and filenames from the .out file, along with their staring and ending addresses. If "UNKNOWN" values are showing in the output, it's because trace captured program execution that had a program address outside the ranges specified by the OFD utility. Common causes might be code that has been dynamically loaded/allocated which wouldn't have associated information in the .out file. You can usually determine the exact cause by looking in the file generated by ofd6x.exe and the .map file and determining why the offending program address is not defined.

Pipeline Stall Analysis Specific FAQS

{{ 1. switchcategory:MultiCore= ▪ For technical support on MultiCore devices, please post your questions in the C6000 MultiCore Forum ▪ For questions related to the BIOS MultiCore SDK (MCSDK), please use the BIOS Forum Please post only comments related to the article **Pipeline Stall Analysis** here.	Keystone= ▪ For technical support on MultiCore devices, please post your questions in the C6000 MultiCore Forum ▪ For questions related to the BIOS MultiCore SDK (MCSDK), please use the BIOS Forum	C2000=*For technical support on the C2000 please post your questions on The C2000 Forum. Please post only comments about the article **Pipeline**	DaVinci=*For technical support on DaVincoplease post your questions on The DaVinci Forum. Please post only comments about the article **Pipeline Stall Analysis** here.*	MSP430=*For technical support on MSP430 please post your questions on The MSP430 Forum. Please post only comments about the article **Pipeline Stall***	OMAP35x=*For technical support on OMAP please post your questions on The OMAP Forum. Please post only comments about the article **Pipeline Stall Analysis** here.*	OMAPL1=*For technical support on OMAP please post your questions on The OMAP Forum. Please post only comments about the article **Pipeline Stall***	MAVRK=*For technical support on MAVRK please post your questions on The MAVRK Toolbox Forum. Please post only comments about the article*	*For technical s please post yo questions at http://e2e.ti.cor Please post on comments abo article **Pipeline Analysis** here.* }}

Please post only comments related to the article **Pipeline Stall Analysis** here.

Stall Analysis *here.*

Analysis *here.*

Analysis *here.*

Pipeline Stall Analysis *here.*

Links

Amplifiers & Linear
Audio
Broadband RF/IF & Digital Radio
Clocks & Timers
Data Converters
DLP & MEMS
High-Reliability
Interface
Logic
Power Management
Processors

- ARM Processors
- Digital Signal Processors (DSP)
- Microcontrollers (MCU)
- OMAP Applications Processors

Switches & Multiplexers
Temperature Sensors & Control ICs
Wireless Connectivity

{{#switchcategory:MSP430=<McuHitboxFooter/>|C2000=<McuHitboxFooter/>|Stellaris=<McuHitboxFooter/>|TMS570=<McuHitboxFooter/>|MCU=<McuHitboxFooter/>|MAVRK=<MAVRKHitboxFooter/>|<HitboxFooter/>}}