ClearSpeed Visual Profiler

CSX System - Example 5

Understanding the performance of a simple application using overlapped I/O transfers

Advance
Accelerate
Optimize

ClearSpeed Visual Profiler

CSX System - Example 5

Introduction

This final example in the CSX system profiling section brings together everything learnt so far into a more complicated example. The concept of asynchronous I/O between the mono and poly processors is introduced and the visual profiler is used to demonstrate the performance benefits that this brings. We use all the facilities of the CSX700 monitor port in this example and also the various options in the debugger for dumping the trace data into different file types. This example also demonstrates how you can use the visual profiler to understand the differences between slightly different pieces of code and analyse the performance differences. There are two Cn source files used for this example, one which works completely sequentially in terms of compute and data transfer and one which overlaps them. A trace session is run for both examples and then the results compared. The user is expected to have been through the visual profiler GUI overview workshop before attempting this example.

Running the example

The Cn source files and csgdb scripts for this example can be found in the following directory

```
csvprof_examples/csx_system/example5
```

This example requires the use of the ClearSpeed SDK in order to assemble the example executable file.

The command lines for generating the workshop file is as follows

```
cscn –o example5_sync.csx –g sinp_sync.cn
cscn –o example5_async.csx –g sinp_async.cn
```

This will generate two executable files `example5_sync.csx` and `example5_async.csx`.

There are also two csgdb scripts for this example, one for each Cn file allowing the user to collect trace non-interactively.

The contents of the first script file (example5_sync.gdb) are as follows.

```
connect
load ./example5_sync.csx
cstrace event all on
tbreak _start
```


Advance
Accelerate
Optimize

```
c
cstrace enable
c
cstrace dump example5_sync
cstrace dump branch example5_sync
q
```

The commands in this file are very similar to the other examples and the user should now be familiar with the steps taken to connect to the CSX700 processor to enable and capture trace via csgdb. This script file is slightly different from the others in the fact that it loads the executable file example5_sync.csx inside this file. The other difference is that all the monitor port events are enabled in the file from one line in the script. As with a previous example the script dumps out trace files in both the standard and branch trace formats for use with the visual profiler.

The second script (example5_async.gdb) is as follows.

```
connect
load ./example5_async.csx
cstrace event all on
tbreak _start
c
cstrace enable
c
cstrace dump example5_async
cstrace dump branch example5_async
q
```

As you can see this script file is identical to the previous one with just the executable file and the name of the trace files differing.

The following commands can now be used to generate the trace data.

```
csgdb –x example5_sync.gdb
csgdb –x example5_async.gdb
```

1. Starting the csvprof GUI

The csvprof GUI is started from the command line using the following command. The example trace file is passed on the command line.

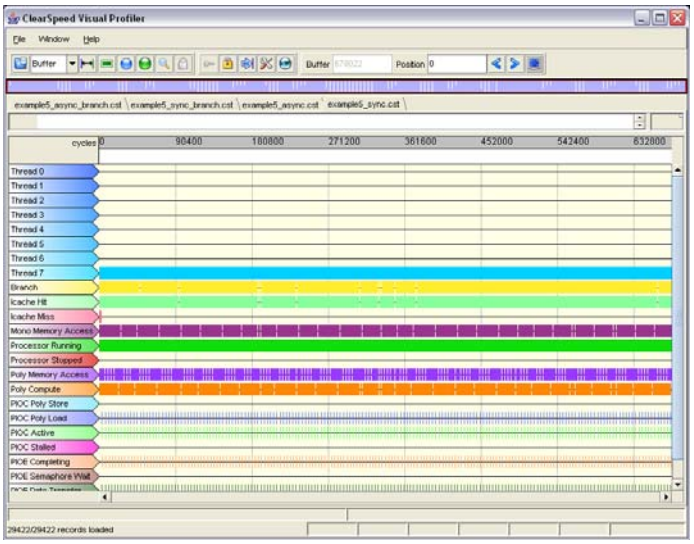
```
csvprof *.cst
```

This command will load up all 4 trace files related to this example.

Start up the visual profiler with the above command line.

The GUI should start up and appear as below.

Advance
Accelerate
Optimize

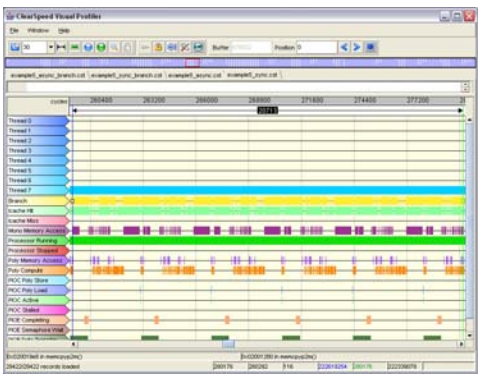


When the GUI starts you can see that there are 4 tabs active in the main window each relating to an individual trace file loaded on the command line. In this example we have loaded 4 files, 2 for each of the executables run.

On start up the view is zoomed to display as much of the trace data as possible and in this case we see the whole file. To analyze this particular example we want to zoom in to the trace data.

 # Set the global markers option and then set the left and right markers in the current trace view in the middle of the trace with around 20000 cycles between the left and right marker. Press fit to markers and then cycle through the tabs and press the fit to markers button for each.

The tabs for example5_sync.cst and example5_async.cst should look as below.



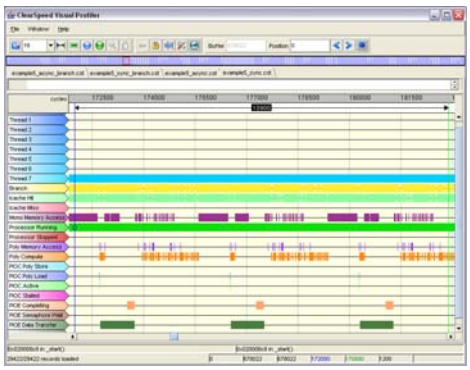
The synchronous data transfer code is shown on the left and the asynchronous on the right. The first thing to note is that the asynchronous version is using multiple threads as is seen by the switching activity between threads 0 and 7. As the files are zoomed to the same scale it is also clear that the frequency of the poly compute events show that the compute is occurring with less gaps. This gives an indication that the asynchronous version is

Advance
Accelerate
Optimize

performing better. As the I/O is overlapped with the compute in the asynchronous example it is possible to execute more compute cycles in the same time. We can see the I/O and compute overlap in the two versions from the visual profiler.

 # Drag the vertical scroll bar down to the bottom to reveal the PIOE Data Transfer events and zoom in around 3 of these events in the example5_sync.cst trace file tab by setting markers around them and pressing fit to markers in all the tabs.

The trace data view should now look as follows.



From these screens we can see the following.

1. In the synchronous version on the left the PIOE Data Transfer event does not overlap the call to sinp() whilst in the async version the I/O does indeed overlap the start of the code.
2. The synchronous version of memcpyp2m() maintains cache coherency so has an extra overhead. This can been seem by the long Mono Memory Access event in the trace.
3. The gaps between the compute are much smaller in the async version meaning that more poly instructions are being issued in the same time frame.

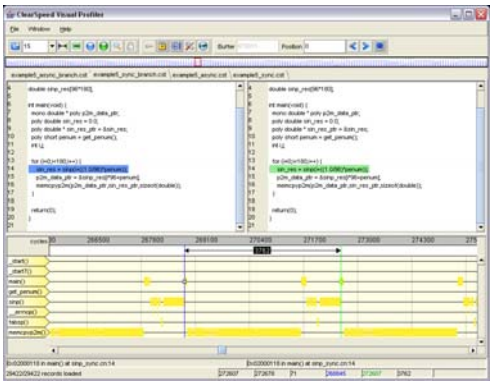
We can now use the branch trace data files which are loaded in the visual profiler to get a measurement for exactly how long a single loop iteration takes in both the synchronous and asynchronous versions. This will allow us to benchmark the inner loop and get a feel for the performance differences. The branch trace view provides a much better idea of the way the code is executing and so provides an easy way to find points to measure between.

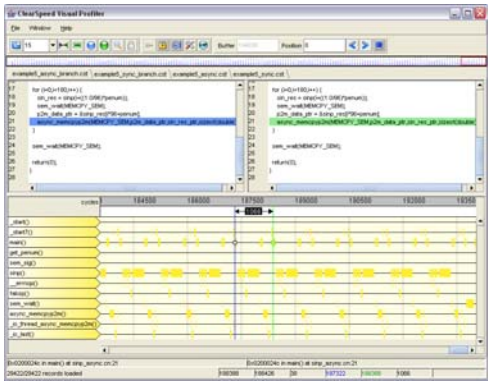
 # Set the lock to event and the snap to event options from the control panel and de-select the global markers option. Select the synchronous branch trace file and set the left marker at the end of the call to main prior to calling sinp(). Set the right marker at the same place but one loop iteration on. Note the cycle time and repeat the same process of setting the markers at the same points in the asynchronous branch

Advance
Accelerate
Optimize

trace file. You may want to adjust the source windows down in these tabs to see where the markers are being set.

The two files should look as follows.





From these views of the trace data we can clearly see that by overlapping the I/O we have gained around a 3x performance increase. The branch trace view makes it much clearer to visualize the difference in the code flow and you can see in the synchronous version just how long it spends in the memcpy() call.

This is the end of the example using the visual profiler to look at complete application performance. Feel free to experiment with the GUI and then use the exit option on the file menu to finish.

Conclusion

In this example we have put the knowledge gained from the previous workshops together to demonstrate how a user can start to analyze performance for a whole application. We have also shown how the visual profiler shows the overlap of compute and I/O to the user which is important for balancing application performance. This simple example shows the user how you would begin to look at the various aspects of application performance in order to optimize code. This workshop concludes the CSX700 system trace analysis section and has provided a user with the basics required to use the visual profiler tool and csgdb to optimize their application code.