ClearSpeed Visual Profiler

CSX System - Example 3

Visually inspecting PE array trace events generated from the CSX700 monitor port

Advance
Accelerate
Optimize

ClearSpeed Visual Profiler

CSX System - Example 3

Introduction

The CSX700 monitor port has the ability to visualize events generated from the poly array. This example focuses on the poly compute and poly load/store events within the PE array. These events relate to when a PE is performing compute and loading or storing to its internal memory. These events are captured using the CSX700 monitor port and then post processed by csgdb into information that can be understood by the visual profiler. The code for this example is provided in Cn with some simple poly functions which will cause the events described above to be displayed. This is the first example in this section to introduce the use of the branch specific trace file which provides timing information visually in relation to the functions called in the code. This example as the others in this section assumes the user is familiar with the features of the visual profiler from the GUI overview workshop.

Running the example

The Cn source file and csgdb script file used for this workshop can be found in the directory

```
csvprof_examples/csx_system/example3
```

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

The command line for generating the workshop file is as follows

```
cscn –o example3.csx –g poly.cn
```

This will generate the executable file `example3.csx`

There is also a csgdb script for this example to allow commands to be given to the debugger in a non interactive mode.

The contents of the script file are as follows.

```
connect
cstrace event mono on
cstrace event thread off
cstrace event polymem on
cstrace event polycomp on
tbreak _start
r
cstrace enable
```


Advance
Accelerate
Optimize

```
c
cstrace dump example3
cstrace dump branch example3
q
```

As with the previous examples a csgdb script is used to configure and capture the trace data. The events enabled are for the mono processor with the hardware thread trace disabled and then the events for the poly load / store and poly compute activity. This introduces the capture of events relating to poly code execution. The next sequence of commands sets a temporary breakpoint at _start, runs to that point, enables the trace and then continues until the program terminates. There are two dump commands in this script file and they relate to the type of trace file being generated. The debugger is capable of generating the trace in the standard format where the processor events are displayed in the GUI or it can output data in the form where the functions in the user code are displayed in the GUI with processor events related to the functions. This second type of file allows a much better view of how the program functions relate to each other. Once these two files have been output the script quits the debugger.

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

```
csgdb –x example3.gdb example3.csx
```

For this example note the cycle time printed out when the example is run, it should be in the order of ~8000 cycles.

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. As this example generates two trace files, one in standard format and the other function specific, we pass both to csvprof on start up. Notice how in this example we are passing both files that were generated on the command line.

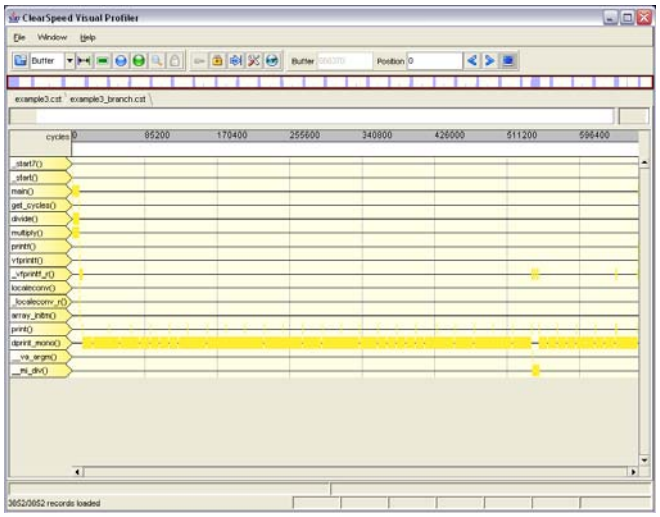
```
csvprof example3.cst example3_branch.cst
```

The second file `example3_branch.cst` contains the trace data in a format relating the branch trace data from the monitor port to functions within the source code. Rather than having monitor port events down the left hand side of the GUI there are functions. This allows a much better understanding of code execution flow and makes timing particular sequences of functions much easier.

Start up the visual profiler with the above command line.

The GUI should start up and appear as below.

Advance
Accelerate
Optimize



As you can see there are now two tabs available within the GUI (one for each file) and the current open view is the last file passed on the command line which in this instance in the branch trace specific file. From this view you can see the visual layout of the branch trace file format.

Selecting the other tab for `example3.cst` will bring up the familiar monitor port trace file used in the previous examples.

 # Select the other tab in the visual profiler and switch to view example3.cst.

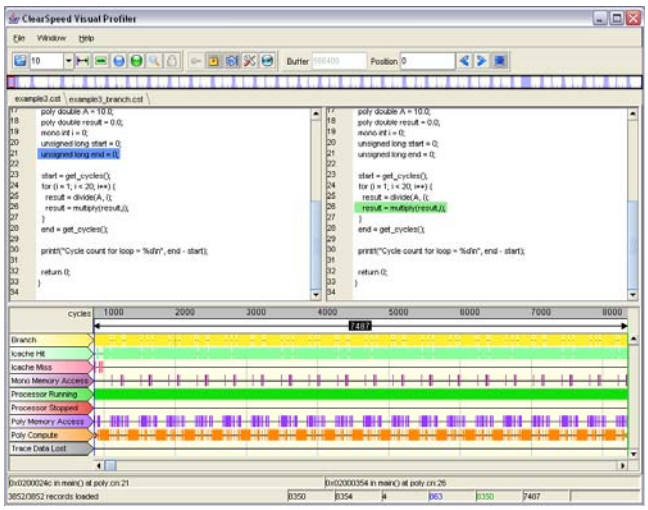
The GUI should now look as follows.

In this example we are interested in looking at the new events for PE microcode active and PE LD/ST active. These tell us when compute and memory access is happening within the poly array. The code we are interested in in this example is only where the microcode is active. If we zoom in around the orange section in the trace we can get a better view of the microcode activity.

Advance
Accelerate
Optimize

 # Switch on the lock cursors and snap to events options from the control panel. Adjust the source code window so that the trace takes up only the space required. Set the left and right markers around the start and end of the poly compute event and press fit to markers.

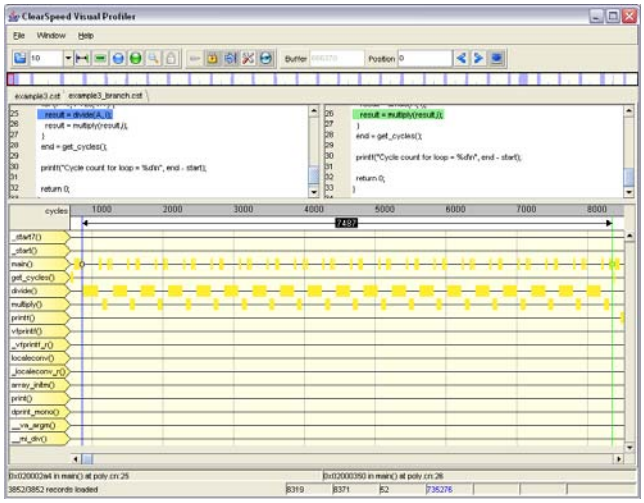
The GUI should now look as below.



From this trace you can see when the microcode and LD/ST activity is occurring. You will notice when selecting microcode events that the source code line does not always match up with the code which operates on the PE array. This illustrates the latency between poly instruction issue from the mono processor and poly execution.

 # Set the global markers option, switch to the other tab, adjust the source code window and press fit to markers.

The following should now be displayed in the GUI.



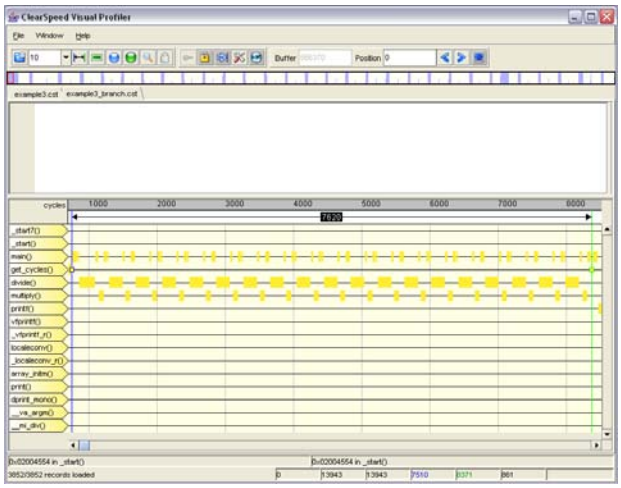
Advance
Accelerate
Optimize

The global markers option allows markers to be set across two trace files and tie them together. This allows interesting sections of trace to be analyzed across separate trace files. Only files that are based off the same data make sense to analyze in this way due to the cycle times for events having to match up.

We can now use the visual profiler to accurately measure the cycle time of the set of loop iterations and compare it to the time taken from `get_cycles()` in the test code.

 # Set the zoom level to 10 (if it is not already), place the left button marker at the end of the first call to get_cycles() and the right button marker at the start of the second call.

The GUI should now display the cycle time for the set of loops and look as follows.



As you can see the cycle time is just as accurate as using `get_cycles()` in the code without the overhead of the function call when you compare the time with the value printed out when the test was run.

Feel free to experiment and when finished exit the GUI from the file menu.

Conclusion

In this example events related to the poly processor have been captured along with events from the mono processor. Along with this we have demonstrated the different styles of program flow analysis that can be done by switching the trace file type being output to the branch trace mode. Using this branch trace mode it is much simpler to locate loops in user code and do performance analysis of code sections by simply selecting the start and end of functions from the trace data. As with previous examples the debugger has been used to enable and capture trace data.

Advance
Accelerate
Optimize