Benchmarking OCPI PCIe Data Throughput on the ML605

Version 1.0

Revision History

Revision	Description of Change	Date
1.0.0-alpha	Initial document creation	02/22/2017

Table of Contents

List of Figures

List of Tables

1 References

This document assumes a general understanding of FPGA development tools and workflow.

Table 1: References

Title	Published By	Link

[[PageOutline]]

2 Benchmarking the PCIe Tx Data Rate

This exercise seeks to enumerate the maximum PCIe rate attainable running within the OCPI framework, and compare the empirical value to that advertised by PCIe.

3 Theoretcal Data Rates

The following describes the methodology for achieving this on a generic platform.

The PCIe transmission protocol has three layers. From the bottom up, they are:

- Phisical Layer (PL)
- Data Link Layer (DLL)
- Transmission Link Layer (TLL)

3.1 PCIe Theoretical Max

Calculating the Theoretical Maximum

Board documentation should provide the theoretical maximum speed of the PCIe interface. If you're unsure as to the generation of your PCIe, you may find clues in the files created by coregen.

As an example, the ml605 kit currently has PCIe v1, x4. So four (32-bit) lanes at 250MB/s per lane = 4*250MB = 1GB/s or 8Gb/s.

Note that this maximum already takes into account the overhead due to 8b/10b data encoding over the physical layer.

3.2 PCIe message protocol overhead

Find overhead created by each layer of PCIe message protocol

3.2.1 Find the maximum allowed TLP length

using lspci as sudo, and with the -vv option will give the maximum allowable payload, for each interface. "'It is important to note that the actual employed maximum payload is limited by the smallest listed allowed, even if that is not the peripheral you are using."'

```
$sudo lspci -vv
 07:00.0 USB controller: NEC Corporation uPD720200 USB 3.0 Host Controller (rev 04) (prog-if
 Capabilities: [a0] Express (v2) Endpoint, MSI 00
        .
        .
        DevCap: MaxPayload 128 bytes, PhantFunc 0, Latency L0s unlimited, L1 unlimited

 08:00.0 RAM memory: Xilinx Corporation Device 4243 (rev 02)
        .
        .
        DevCap: MaxPayload 512 bytes, PhantFunc 0, Latency L0s <64ns, L1 unlimited
```

The above code block shows the limiting PCIe segmentation is 128 bytes.

3.2.2 Subtract TLL protocol overhead per packet as defined by transmitting hardware

The TLL overhead per packet for PCIE gen 1 consists of

- A 12-16-byte header (12 bytes 32-bit addressing, and 16 bytes for 64-bit addressing)
- An optional 4-byte ECRC

3.2.3 Subtract DLL overhead as defined by recipient hardware

The DLL overhead per TLL packet consists of

- 1 start byte
- A 2-byte sequence ID
- A 4-byte LCRC
- 1 end byte

Using the max payload size found in step 1, and the overhead specified in 2 and 3, we can calculate the theoretical maximum throughput.

$$\frac{(16B + 8B)}{(16B + 8B + 128B)} = about 15.8\%$$

3.3 OCPI Overhead

Calculate additional overhead caused by OCPI RDMA vmsg which includes (per message, defined as between start of message (som) and end of message (eom)

Note actual data in message can be zero-size.

3.3.1 Caluclate size of OCDP buffer

To achieve minimum overhead percentage, use the largest message where entire message+overhead will fit into the FPGA buffer used. In our test case, the BRAM used in the OCDP is 32768 bytes. The BRAM consists of 2 buffers, so each buffer is 16384 bytes.

If you already have an app up and running, you can use ocpirun -l 8 ¡app¿ to get the OCDP (hardware) buffer size:

```
$ ocpihdl -d 8 -v -x get
HDL Device: 'PCI:0000:08:00.0' is platform 'ml605' part 'xc6vlx240t' and UUID 'a8a14a66-7c89-
Platform configuration workers are:
  Instance p/ml605 of platform worker ml605 (spec ocpi.platform) with index 0
  .
  .
  .
Container workers are:
  Instance c/pcie_ocdp0 of interconnect worker ocdp (spec ocpi.devices.ocdp) with index 5
  0         nLocalBuffers: 0x2
  1        nRemoteBuffers: 0x2
  2       localBufferBase: 0x0
  3     localMetadataBase: 0x7fe0
  4       localBufferSize: 0x800
  5     localMetadataSize: 0x10
```

```
 6          nRemoteDone: <unreadable>
 7           rsvdregion: <unreadable>
 8               nReady: 0x0
 9             foodFace: 0xf00dface
10                debug: 0x0
11          memoryBytes: 0x8000
 .
 .
 .
```

Line 11 shows the buffer size set for this assembly is 0x8000, or 16352 bytes.

3.3.2 Calculate the overhead due to OCPI messaging

Each OCPI message has the following. Dwords are 32-bit words.

- 4 dwords metadata
- N dwords data (N can be 0)
- 2 dwords flag

Therefore, the overhead due to OCPI in this case is

$$\frac{24}{24+16352} = about 1.5\%$$

The overall theoretical maximum PCIe data throughput can now be calculated

Maximum Data Throughput = Max Rate * (PCIe data throughput) * (OCPI data througput)

$$8Gb/s * \frac{1-(16B+8B)}{16B+8B+128B} * 1 - \frac{24B}{24B+16352B} = 6.727Gb/s = 841MB/s$$

4 Observed Data Rates

4.1 Full Hardware -> Software Application

To determine the actual PCIe rate, set up an application that uses two workers, one firmware and one software, which will communicate over PCIe. In the test case below characterizes a PCIe interface on an ml605 board. The firmware (hdl) worker runs on a Virtex 6, and the software (rcc) worker is on a Linux machine with an x86_64 architecture.

In the initial test case, the firmware worker, written in vhdl, contains a counter. The value of the counter is then transmitted over PCIe to a file_write software worker. There are various switches in the firmware that can vary the rate and conditions under which the counter value progresses.

TODO: – attach workers, application and assembly–

Initially, write to a readable file for verification that the count progresses linearly, and there are no gaps. Once data is verified, it is best to have the file_write worker point to /dev/null to minimize the processing lag.

Here is a graph showing observed data rates writing to three different locations:

[[Image(PercentWordsWritten_vs_ClkDivision.gif)]]

* looking at the graph, we see that at 125MHz, with no clock division, the effective data rate is just above 40

4.2 NFT

NFT is an ocpi program that will connect and run hardware workers, and will run data to the PCIe interface without necessitating a software worker to receive the data. Using nft with the workers mentioned above could yield a faster data rate than the traditional application/assembly message.

4.2.1 Modifying the NFT code

You will have to modify the nft c code to fit your desired data flow as follows:

1. Change the defined workers to match the workers in your data flow. If you have previously run your application without nft, you can find the correct indecies to put in the parentheses by running ocpihdl get 8. 8 is the typical device assignment given to the PCIe:

```
HDL Device: 'PCI:0000:08:00.0' is platform 'ml605' part 'xc6vlx240t' and UUID '381f2148-7
Platform configuration workers are:
  Instance p/ml605 of platform worker ml605 (spec ocpi.platform) with index 0
Container workers are:
  Instance c/pcie\_ocdp0 of interconnect worker ocdp (spec ocpi.devices.ocdp) with index
  Instance c/pcie\_sma0 of adapter worker sma (spec ocpi.sma) with index 3
Application workers are:
  Instance a/multirate\_counter of normal worker multirate\_counter (spec benchmark.multi
```

   Above we see that the ocdp has index 2, the sma has index 3, and the multirate_counter has index 1. Now we can modify the workers in nft as below:

```
//#define WORKER_DP0 (2)
//#define WORKER_DP1 (4)
//#define WORKER_SMA0 (3)
//#define WORKER_BIAS (1)
//#define WORKER_SMA1 (5)
#define WORKER_DP0 (2)
#define WORKER_SMA0 (3)
#define WORKER_MULTIRATE_COUNTER (1)
#define OCDP_OFFSET_DP0 (32*1024)
```

2. Next change the names of your volatiles to reflect the new worker names:

```
//  volatile OcdpProperties *dp0Props, *dp1Props;
//  volatile uint32_t *sma0Props, *sma1Props, *biasProps;
//  volatile OccpWorkerRegisters *dp0, *dp1, *sma0, *sma1, *bias;
  volatile OcdpProperties *dp0Props; //*dp1Props;
  volatile uint32\_t *sma0Props, *multirateCounterProps; //*sma1Props, *biasProps;
  volatile OccpWorkerRegisters *dp0, *sma0, *multirateCounter; //**dp1, *sma0, *sma1, *bias;
```

3. Remove function calls associated with unused workers, and add corresponding function calls for new workers:

```
//dp1Props = (OcdpProperties*)occp->config[WORKER_DP1];

//sma1Props = (uint32_t*)occp->config[WORKER_SMA1];
//biasProps = (uint32_t*)occp->config[WORKER_BIAS];
multirateCounterProps = (uint32_t*)occp->config[WORKER_MULTIRATE_COUNTER];

//  dp1 = &occp->worker[WORKER_DP1].control,
//sma1 = &occp->worker[WORKER_SMA1].control,
//bias = &occp->worker[WORKER_MULTIRATE_COUNTER].control;
multirateCounter = &occp->worker[WORKER_MULTIRATE_COUNTER].control;

//reset(sma1,  0);
//reset(bias,  0);
reset(multirateCounter,  0);
//reset(dp1,  0);

//init(sma1);
//init(bias);
//init(dp1);
```

```
//*sma0Props = 1; // WMI input to WSI output
//*biasProps = 0; // leave data unchanged as it passes through
//*sma1Props = 2; // WSI input to WMI output
  *sma0Props = 2; //1; // WMI input to WSI output
  *multirateCounterProps = 0; // leave data unchanged as it passes through


//  setupStream(&fromCpu, dp0Props, false,
//              nCpuBufs, nFpgaBufs, bufSize, cpuBase, dmaBase, &dmaOffset, ramp);

  setupStream(&toCpu, dp0Props, true,
               nCpuBufs, nFpgaBufs, bufSize, cpuBase, dmaBase, &dmaOffset, ramp);

//  setupStream(&toCpu, dp1Props, true,
//              nCpuBufs, nFpgaBufs, bufSize, cpuBase, dmaBase, &dmaOffset, ramp);

//start(dp1);
start(multirateCounter);
//start(bias);
//start(sma1);
```

4. Lastly, since our data flows only from the FPGA to the CPU, we can remove any code inferring data flow in the other direction.

```
//#if 0
//     for (n = 1000000000; n && !(tcons = fromCpu.flags[fromCpu.bufIdx]); n--) {
//     }
//     if (!n) {
//       fprintf(stderr, "Timed out waiting for buffer from cpu to fpga\n");
//       return 1;
//     }
//     checkStream(&fromCpu, NULL, NULL);
//#endif
```

4.2.2 Allocating Memory

We are almost ready to run NFT. If you try to run it now, you will likely get the message

```
You must set the OCPI_DMA_MEMORY variable before running this program
```

In order to find the proper memory size, do the following:

```
#sudo cat /etc/grub.conf

title CentOS (2.6.32-504.1.3.el6.x86_64)
        root (hd0,4)
        kernel /vmlinuz-2.6.32-504.1.3.el6.x86_64 ro root=/dev/mapper/VolGroup-lv_root rd_NO_L
SYSFONT=latarcyrheb-sun16 crashkernel=128M rd_LVM_LV=VolGroup/
```

We see that the operating system has 512M of memory at location 7F790000. To set OCPI_DMA_MEMORY, write the following:

```
export OCPI_DMA_MEMORY=512M\$0x7F790000
```

4.2.3 Running NFT

Now, to run NFT:

```
sudo -E `which nft` -m 1024 -t -s -v 0000:08:00.0 > foo
```

- sudo -E allows you to run as a superuser, while preserving your environment variables

- -m defines the number of frames for the program to process. The frame size is set by the unsigned variable bufSize. Make sure that bufSize matches or exceeds the set size of the fpga buffer.
- -t displays metrics at the end of each program run including processing speed.
- -s runs the program in a single thread
- -v prints the direction of each frame (to cpu or from cpu), the opcode, and the buffer index

Below is a sample output

```
$ sudo -E `which nft` -m10 -c -t -s -v 0000:08:00.0 > foo
BufSize=16384, CpuBufs 200 FpgaBufs 2 Ramp 0
delta ticks min 0 max 0 avg 0
now delta is: 0ns
res: 1
to cpu stream: d 7f790000 m 7fab0000 f 7fab0c80
Running single threaded
Nanoseconds:    Size         Pull        Push       Total    Processing
to cpu done 1 0
Measure:       16384   1279424260       14865  2016524255   737085130
to cpu done 2 1
Measure:       16384   1279424260       14930  2016923360   737484170
to cpu done 3 2
Measure:       16384   1279424260       14905  2017242585   737803420
to cpu done 4 3
Measure:       16384   1279424260       14930  2017561815   738122624
to cpu done 5 4
Measure:       16384   1279424260       14850  2017960805   738521694
to cpu done 6 5
Measure:       16384   1279424260       14850  2018280079   738840969
to cpu done 7 6
Measure:       16384   1279424260       14850  2018599329   739160219
to cpu done 8 7
Measure:       16384   1279424260       15125  2018998614   739559229
to cpu done 9 8
Measure:       16384   1279424260       14935  2019317864   739878668
to cpu done 10 9
Measure:       16384   1279424260       14935  2019637094   740197898
Bytes 161920, Time delta = 3565, 45.419355 MBytes/seconds, Message size 16384

$ sudo -E `which nft` -m10 -c -t -s 0000:08:00.0 > foo
BufSize=16384, CpuBufs 200 FpgaBufs 2 Ramp 0
delta ticks min 0 max 0 avg 0
now delta is: 0ns
res: 1
Running single threaded
Nanoseconds:    Size         Pull        Push       Total    Processing
Measure:       16384   1140274596       14850   509339577  3664017427
Measure:       16384   1140274596       14850   509499132  3664176982
Measure:       16384   1140274596       14825   509658662  3664336537
Measure:       16384   1140274596       14825   509818187  3664496062
Measure:       16384   1140274596       14825   509977741  3664655617
Measure:       16384   1140274596       14825   510137296  3664815172
Measure:       16384   1140274596       14800   510296821  3664974721
Measure:       16384   1140274596       14850   510456401  3665134251
Measure:       16384   1140274596       14805   510615931  3665293827
Measure:       16384   1140274596  4294902386   510775531  3665533141
Bytes 161920, Time delta = 1749, 92.578616 MBytes/seconds, Message size 16384
```

These two runs show data speeds of 45.4 and 92.6MB/s. The slower run is on par with the data rate observed using the full OpenCPI with the counter-to-file_write application. The faster run offers almost a two-fold improvement.

5 Further Exploration

There is a possibility that clock bubbles inserted between n-1 message's eom and n message's som. If you suspect this behavior (perhaps based on chipscope output) here are some methods to find bubbles:

5.1 Monitor Worker

A monitor entity can be inserted between two workers, then connect output of monitor to capture. monitor can be programmed to count bubbles per message boundary, and write to capture worker.

Currently the monitor worker is not on the OCPI git hub. Email TODO: contact info

5.2 ocpihdl commands

ocpihdl -v get, run after a ocpihdl run can dump can tell number of busy signals observed in the sma worker within an execution of an application. Compare busy signal counts between workers to see where data is getting held up.

to allow OCPI to log busy signals, there is a debug switch in the worker that needs to be from the default "off", to "on". The easiest way to do this, currently, is to modify sma.v, and rebuild. See below for the modification.

```
wire     hasDebugLogic = 1'b1; //ocpi_debug;
```