CLAPPS Tutorial

Geraldo Francisco de Oliveira

September 29, 2017

1 Pre-Requisites

The CLAPPS simulator was developed using the SystemC language in its 2.3.1. version. Therefore, it is expected that the user has already sucessful installed the SystemC API. In order to use the available Makefile provided alongside the CLAPPS framework, the SystemC library must be installed in:

/usr/local/systemc-2.3.1/lib-linux64

Besides that, to compile CLAPPS, it is required to be installed C++ version 11. The IDE used to create CLAPPS was Eclipse Mars.2.

2 Download and Compile

To download CLAPPS, one must clone the following git director:

```
git clone https://github.com/clapps-hmc/clapps.git
```

Now, to compile CLAPPS, just go to the Debug folder and type *make all*. It will generate the executable file, called *clapps*, at the root of the working directory.

3 Trace generator

Since our simulator is trace-based, we provided some scripts that can produce valid trace files to feed the simulator. There are four distinct trace scripts available. The *trace-generator-seq.py* script generates READ/WRITE requests that accesses memory sequentially. The *trace-generator-pim-seq.py* script also generates a trace file that access memory sequentially, but creates native PIM HMC requests. On the other hand, both *trace-generator-rand.py* and *trace-generator-pim-rand.py* scripts generate READ/WRITE and PIM HMC requests, respectively, that access memory randomly. The trace file generated has the memory address, HMC command, cub_id, and in the case of a WRITE request, the input data. All python scripts can be founded in the folder *scripts*. To see the parameters required for each python script, just execute *python script_name.py*.

If the user decides to evaluate the memory execution of a real benchmark, he or she needs to convert the benchmark's memory access pattern to the simulator's trace format. Each line of the trace file has the HMC opcode, the memory address, the number of data FLITs in the request, and the list of data itself. For example, in the following line of a trace file,

0001001 0000000000000000000000000000000000 2 7 9

the first seven bits (0001001) indicate the operation is a WRITE request of 32 bytes, the following 34 bits indicate the target memory address, then the third portion indicates that the list of input data has two elements, and then 7 and 9 are the input data elements themselves.

4 Defines

The defines configuration file, in *src/common/defines.h*, specifies some parameterizable HMC values. The user can modify the number of Vaults, the number of links, the row buffer size, the TSV width, the number of memory controllers, the DRAM's timing parameters, and the Vault and Transceiver clock rate in the defines file. It is important to point out that the user must not generate a request HMC command (READ/WRITE) bigger than the *MAX_BLOCK_SIZE* parameter (default 32B). When modifying this parameters, the code must be recompilated.

5 Simulation Core

There are three important fiels that are the core of the CLAPPS simulator. First, the *hmc-testbench* file drives and monitors the *hmc-device* module. It reads the trace file produced by the trace generator scripts, and sends one HMC instruction in the trace file to the simulator at each transceiver's clock cycle. The *traffic-monitor* gathers run-time statistics. The *hmc-device* is the implementation of the HMC architecture.

6 Running

Figure 1 illustrates the simulating steps that CLAPPS goes through during execution. Once the simulator has been compiled, just run:

./clapps TRACE_NAME

If TRACE_NAME was not defined, the hmc_testbench will look for *trace.txt* in the root of the working directory. The simulating process will procede as following.

In ❶, the user generates the trace file that will be used during simulation. The trace file can be produced using any of the available trace generator scripts.

In ❷, the user can modify the standard HMC structure by changing the define file. Then, he or she needs to compile the simulator. Since SystemC is based on C++, the only library the user needs to provide and link during compilation is the SystemC 2.3.1. one. When the compiling and linking process has finished, the user can run the simulator. ❸ describes the run-time process. The *hmc-testbench* will read the trace file, driving the *hmc-device* with all requests in the trace file. The requests are sent to the *hmc-device* one per clock. Besides that, the *hmc-testbench* will monitor the output port of the *hmc-device*, writing all response packets generated by the latter into the

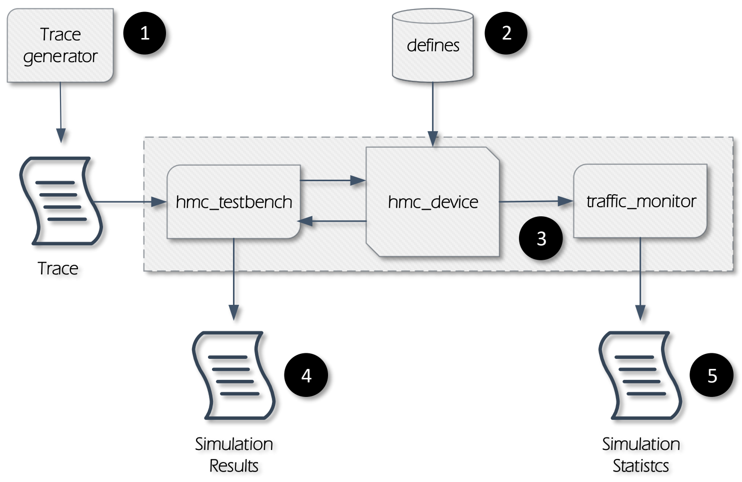

Figure 1: Simulation steps our mechanism travels through execution.

simulation results file 4. To check the correctness of the simulation, the user can compare the simulation result file with the gold file produced by each one of the four trace generator scripts. Finally, at the end of the simulation, the *traffic-monitor* writes the produced statistics to the simulation statistics file 5. This file is divided into per Vault and per Link statistics. In both cases, it is shown the total number of bytes and written, the total time (in ns) the *hmc-device* took to perform all requests, the obtained individual and global bandwidth (in GB/s), and the number of packets produced.

7 PIM Interface

The primary goal of this work is to provide a simple yet concise infrastructure that allows new PIM architecture exploration. To do so, we have implemented the HMC design as previously described, and studied a way to insert a custom processing module into HMC without having to worry about HMC organization. We came to a simple PIM Interface, placed into the *Vault Controller*, that receives custom PIM instructions, and also provides a mechanism to perform read/write operations.

To implement the PIM Interface, we decided not to modify dramatically the HMC data-path and its communication protocol. However, we still had to perform four minor modifications to the HMC design. First, we have included a new opcode command to the set of HMC command operations. The opcode included was the PIM_INSTRUCTION. All custom PIM instructions will make use of this same opcode. A single PIM request is seen by our simulator similarly to a 16 bytes write request. All information related to the custom PIM instruction will be placed in the data sections of the request packet. Therefore, the user does not need to worry about violating already reserved opcodes or being limited by the number of free available HMC opcodes when de-

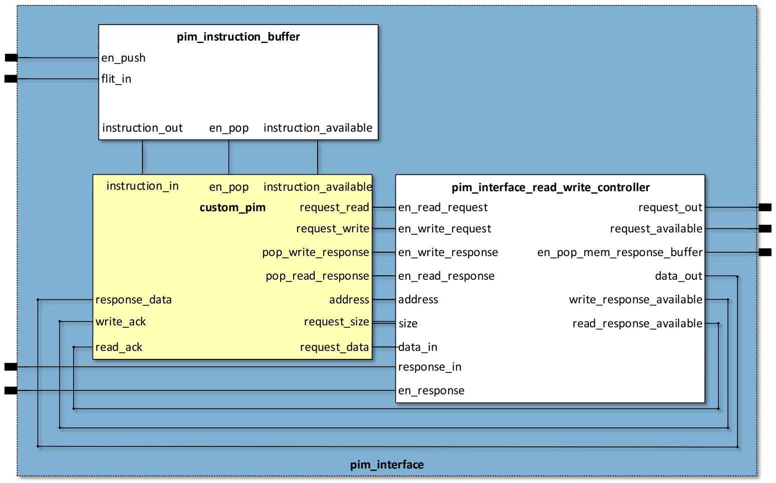

Figure 2: Block diagram of the proposed PIM Interface.

signing his or her instruction set. Second, we have included the already mentioned *traffic_request_controller* module. This module is required to allow a request flow to come from both the host and from the PIM Interface. Third, we have made use of a reserved bit from the request header field (bit 23), to indicate to the *request_controller* whether the current READ/WRITE request should return to the host or the PIM Interface. Finally, we have included a path between the request decoder and the PIM Interface, therefore FLITs related to a PIM request can be sent to the PIM Interface.

Figure 2 depicts the proposed PIM user interface. The module is composed of the *pim_instruction_buffer* and the *pim_interface_read_write_controller*. The process of performing custom PIM operations works as follows. First, when issuing a new request to the *Transceiver* module, the user must specify the PIM_INSTRUCTION opcode, the address of the request, and then a 16 bytes of data that represents their custom instruction. The *packet_generator* will encapsulate this request the same way it makes a write request. After that, the produced FLITs will travel through the same path as a standard request until arriving at the *request_decoder*. At this point, when the *request_decoder* de-codifies the command portion of the request, it will not recognize the opcode, and then will send the upcoming FLITs to the *pim_instruction_buffer* at the PIM Interface. The *pim_instruction_buffer* will then remove additional payloads of the FLITs (as header and tail fields), and will raise the *instruction_available* output signal to notify the *custom_pim* module that there is a new instruction currently available to it. The *custom_pim* does not need to have any information about how the packet was encapsulated since the upcoming PIM instructions have only information related to its own instruction set.

When the *custom_pim* instruction decides to perform a READ or WRITE operation to the HMC module, it sets the *request_read* or *request_write* signal, informing to the *pim_interface_read_write_controller* that a new HMC request must be generated. The *custom_pim* also provides the address it wants to access, the size of the request, and data values in the case of a WRITE operation. The controller will then encapsulate the PIM request into FLITs, creating a header and tail field in accord with the HMC protocol, and then transmitting the generated request back to the *Vault Controller*. Also, it will set bit 23 of the header field to one, indicating that the response package must return to the PIM Interface. The request will be sent to the *traffic_request_monitor*, and when appropriated, it will be sent to the *request_decoder*. Once at the *Vault Controller*, the request will follow the same path as any standard host request. However, once the request goes through the *response_control*, the module will check that the *rsp_destination* signal is set to one (recall this signal stores the value of bit 23 of the header), and then all response FLITs will be sent back to the PIM Interface. Finally, when the PIM Interface receives the response packets from the *Vault Controller*, it will notify the *custom_pim* module that its READ/WRITE request is completed by raising the *write_ack* or *read_ack* signals. In the case of a READ request, the PIM Interface removes the header and tail payloads of the response package, and send the requested data to the *custom_pim*.