PacketNoc - How To

Fabio Arlati - arlati.fabio@gmail.com
Antonio Miele - miele@elet.polimi.it

First draft: July 2010
Revision: October 2010

Contents

What is this document?

This document is an hands-on guide to the use of the packet NOC component defined in ReSP. The instantiation, connection and manipulation of this component will be described in the next few sections with the aid of some pictures.

1 Definition

This NOC model is a standard TLM component (which could be attached to other elements in ReSP), but is internally built as a common SystemC component that represents a set of packet-exchanging components. In particular, the internal architecture of the NOC model is based upon three different types of components:

1. Master Processing Element (MPE), which represents the interface module between a connected master and the NOC. It is devoted to the packeting the TLM transactions coming from the attached TLM initiator;

2. Switch, which receives, memorizes into a buffer, and forwards the packets according to a given route;

3. Slave Processing Element (SPE), which represent the interface module between a connected slave and the NOC. It is devoted to unpacket TLM transactions, forward them to the attached TLM target, and send back a packeted answer to the requesting MPE.

For every TLM transaction, the appropriate MPE starts a new session (composed by one or more packets). This session is concluded only when the entire answer (composed, in turn, by one or more packets) is received from the SPE. If the answer is not received before a fixed timeout, the session is restarted, and a new series of requesting packets is created by the MPE. Until a session has not been successfully completed, the TLM transaction will not be concluded.

2 Instantiation

A simple example of how a packet NOC can be instantiated is contained in the following ReSP architecture file:

`resp-sim/architectures/test/test_packetNoc.py`

As shown in Figure 1 at lines at lines 132-133, the NOC is instantiated starting from a small set of parameters:

- A name for the component (*noc* in the example).
- The path to an `xml` file describing the NOC (contained in the NOC_DESCRIPTION variable of the example).
- The period of the internal clock of the NOC (calculated starting from the NOC_FREQUENCY variable of the example).

The following `xml` file describes the NOC of the considered example:

`resp-sim/architectures/test/test_packetNoc.xml`

```
132     clockNoc = scwrapper.sc_time(float(1000)/float(NOC_FREQUENCY), scwrapper.SC_NS)
133     noc = packetNoc32.packetNoc('noc',NOC_DESCRIPTION,clockNoc)
134     connectPorts(noc, noc.initiatorSocket, mem1, mem1.targetSocket)
135     connectPorts(noc, noc.initiatorSocket, mem2, mem2.targetSocket)
136     # Add memory mapping
137     noc.addBinding(5,0x0,memorySize1-1)
138     noc.addBinding(6,memorySize1,memorySize1+memorySize2-1)

160         connectPorts(dataCaches[i], dataCaches[i].initSocket, noc, noc.targetSocket)

175         connectPorts(instrCaches[i], instrCaches[i].initSocket, noc, noc.targetSocket)
```

Figure 1: The instantiation of NOC example

Figure 2 shows a fragment of the XML descriptor. The XML descriptor has to contain a number of MPEs equal to the number of initiator components we want to connect; similarly, it has to contain a number of SPEs equal to the number of target components we want to connect. In the considered, the architecture is composed by 4 master components (2 processors, which, according to the Harvard architecture, have both a data and an instruction port) and 2 slave components (2 simple memories). Thus, inside the NOC XML descriptor we need to declare 4 MPEs and 2 SPEs; moreover, every MPEs and SPEs require a unique ID as an attribute. Between these elements we are free to declare an undefined number of switches with any kind of topology: even the switches require a unique ID attribute. The connections can be declared through the *connect* tag embedded inside an element, specifying the target ID; connections are unidirectional, thus for a bidirectional path it is necessary to specify another *connect* tag for the inverse direction. The number of connections is limited in the following ways:

- Processing elements are allowed to have only one connection.
- Switches can have up to *PORTS* connections, where *PORTS* is a constant defined in `/resp-sim/components/interconnection/packetNoc/nocCommon.hpp`. Moreover, at most a MPE or a SPE can be connected to each switch. The standard value of *PORTS* is 5, if you want to modify it, you should recompile the tool (use `./waf` in the main folder).

Starting from this information the entire NOC is initialized, the SystemC connections between the components are automatically performed, and the routing tables of the switches are automatically computed by applying the Dijkstra algorithm to an ad-hoc Boost Graph. Figure 3 shows the graph obtained by reading the file in our example.

```
 1 <?xml version="1.0" ?>
 2 <noc>
 3   <mpe id="1">
 4     <connect dst_id="7"/>
 5   </mpe>
12   <mpe id="4">
13     <connect dst_id="8"/>
14   </mpe>
15   <spe id="5">
16     <connect dst_id="9"/>
17   </spe>
26   <switch id="8">
27     <connect dst_id="3"/>
28     <connect dst_id="4"/>
29     <connect dst_id="9"/>
30   </switch>
31   <switch id="9">
32     <connect dst_id="5"/>
33     <connect dst_id="6"/>
34     <connect dst_id="7"/>
35     <connect dst_id="8"/>
36   </switch>
37 </noc>
```

Figure 2: The XML descriptor of NOC example

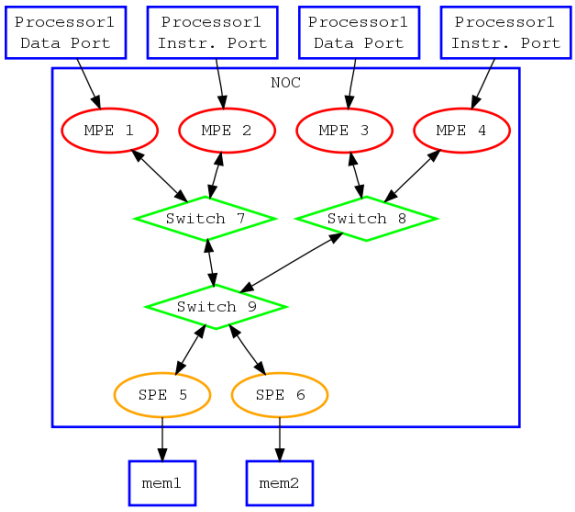

Figure 3: The connection graph of the NOC example

After the instantiation, the NOC is connected to the initiator components (the masters, usually the processors and their caches), at lines 160 and 175 and to the target components (the slaves, usually the memories) at lines 134-135. Keep in mind that the NOC externally provides only two multi passthrough TLM ports (one for targets and one for initiators); then, internally the various initiator and target components are binded to the appropriate MPE or SPE according to a positional order of the components' connection and MPEs/SPEs' declaration. For instance, first connected initiator port, that is the data port of the first processor is connected to MPE 1 that is the first MPE defined in the XML file, and so on; similarly, *mem1* is binded to SPE 5 and *mem2* is binded to SPE 6.

Finally, we have to define the map the various target components in the address space. it is performed at lines 137-138 where the first parameter represents the ID of the SPE of the target component we want to map.

3 Interaction

The NOC provides a complete set of functions to monitor and modify its behavior at run-time:

printBindings() prints on the standard output the address bindings of the NOC.

printGRT() prints on the standard output all the Routing Tables of the NOC switches.

printStats() prints on the standard output all the current statistics of the NOC elements (flits received, sent and dropped).

getActivePorts(unsigned int switchId) returns the IDs of the active ports in a given switch.

getBufferCurrentFreeSlots(unsigned int switchId, unsigned int portId) returns the number of free slots in a specific buffer of a switch.

getFlitsIn(unsigned int elId, unsigned int portId) returns the number of flits received by a particular element on a specific port. In case the element is a MPE or a SPE, portId is ignored.

getFlitsOut(unsigned int elId, unsigned int portId) returns the number of flits sent by a particular element on a specific port. In case the element is a MPE or a SPE, portId is ignored.

getTimeouts(unsigned int masterId) returns the number of sessions timed out in a particular master element.

getAllTimeouts() returns the number of sessions timed out in all master elements.

getBufferSize(unsigned int switchId, unsigned int portId) returns the size of a specific buffer of a given switch.

getDropped(unsigned int switchId, unsigned int portId) returns the number of flits dropped by a particular switch on a specific port.

getAllDropped() returns the number of flits dropped by all the switches.

changeTimeout(unsigned int masterId, sc_time tO) changes the timeout value of a particular master element. The standard timeout is 100 ns, and is hardcoded in `/resp-sim/components/interconnection/packetNoc/nocCommon.hpp`. If you wish to change the standard timeout, you should reissue `./waf` in the main folder.

changeAllTimeouts(sc_time tO) changes the timeout value of every master element.

changeBufferSize(unsigned int switchId, unsigned int portId, unsigned int size) changes the size of specified buffer of a particular switch. The standard buffer size if 1, and is hardcoded in `/resp-sim/components/interconnection/packetNoc/nocCommon.hpp`. If you wish to change the standard buffer size, you should reissue `./waf` in the main folder.

changeAllBufferSizes(unsigned int size) changes the buffer size of every switch.

changePath(unsigned int switchId, unsigned int destinationId, unsigned int nextHop) changes the forwarding path for a given destination in a particular switch. The `nextHop` parameter requires the ID of a switch (or an element) directly attached to the switch under analysis. The forwarding paths are initialized by the Dijkstra shortest path algorithm applied to the network graph when the NOC is constructed.

getSwitchIds() returns the list of the IDs of all the switches instantiated in the NOC.

An example of how these functions are used can be found in `resp-sim/architectures/test/test_packetNoc.py` at line 223, where all the timeouts of the master elements are set to 1 µs. This operation is necessary in our example, because whenever one of the caches requires to load an entire block of memory, a huge amount of packets is generated during the session; in order to allow the completion of these sessions, a greater timeout is required in the master elements, otherwise every session will reach the standard timeout of 100 ns. It is strongly recommended to use this approach to change the standard parameters of the NOC, instead of modifying the constants in `nocCommon.hpp`.

It is worth noting that the described interface can be easily enhanced in order to expose further methods for accessing and modifying the NOC characteristics.

4 Automatic Interaction

It is possible to instantiate a particular component, named *governor*, which automatically executes a specified routine every specified period of time. This governor is directly attached to the packet NOC, thus it is possible to perform an automated interaction using the functions specified in Section 3.

The governor can be instantiated as shown in `resp-sim/architectures/test/test_packetNoc2.py` at line 102. It requires three parameters:

- A name for the component (*governor* in the example).
- The NOC object previously instantiated (*noc* in the example).
- The period of the internal clock of the governor (1 ms in the example).

The behavior of the governor is simple: at every clock cycle (i.e. every ms), a routine is executed: this routine is named `activity` and is specified in `resp/components/interconnect/packetNoc/governor/packetNocGovernor.cpp`. The user can customize the routine simply by re-implementing the function body. remember that, after the routine has been changed, a recompilation should be issued by launching `./waf` in the main folder.

In the example proposed in `resp-sim/architectures/test/test_packetNoc2.py`, the `activity` routine implements a simple resource manager able to dynamically reconfigure the sizes of the buffers inside each switch according to the information about the traffic in the NOC. In particular, the memory slots are distributed among the buffers of each switch according to the packet drop statistics; in this way, the size of the buffers on on the direction the switch is experiencing a higher traffic are increased with the free slots of the buffers on the other directions.

The governor is initialized at lines 94-104 of the architecture file by specifying a set of parameters:

BUFFER_SIZE the initial buffer size;

INIT_THRESHOLD the initial value of the parameter driving the buffer size changing;

MIN_THRESHOLD the minimum threshold for the parameter driving the buffer size changing;

MAX_THRESHOLD the maximum threshold for the parameter driving the buffer size changing;

MULT_FACTOR a multiplication factor used on packet drop;

RED_FACTOR a reduction factor used when no packet drop occurs on a specific buffer

Given these parameters, the *activity* algorithm works as following: a *drop_index* parameter is assigned to every buffers of each switch and it is initialized with *INIT_THRESHOLD* value. At each execution of the *activity* routine, each buffer of each switch is checked to detect if any drop occurred. For each occurred drop, *drop_index* is multiplied by MULT_FACTOR; otherwise, it is subtracted by RED_FACTOR. Then, if *drop_index* of a buffer overcomes the *MAX_THRESHOLD*, its size is increased by one by removing a slot from another buffer of the same switch with the minimum *drop_index* (it should have at least a free slot); after, the *drop_index*s of both the buffers are re-initialized. Similarly, if a buffer *drop_index* becomes lower than *MIN_THRESHOLD* and it has a free slot, the slot is reassigned to another buffer of the same switch with the highest *drop_index* and the two

parameters are re-initialized. In this way, it is possible to note that when accurately tuning the parameters, there is a reduction in the packets' drop and consequently in the overal execution time. Another example of use of the *governor* is shown in the architecture `resp-sim/architectures/test/test_packetNoc3.py`.