Seventh Framework Program –Theme 3.6: Computing Systems

Grant Agreement n° 248 776

D1.3

"3D Interconnects - Modelling & Characterisation"

WP1: "From 3D Opportunities to 3D Manycore Architectures"

Editor: Martino RUGGIERO (UNIBO)

Date: Friday, 08 April 2011

1	CEA (coord.)	Commissariat à l'énergie atomique et aux énergies alternatives – Laboratoire des technologies de l'information.
2	UJF	Université Joseph Fourier, Grenoble I – VERIMAG
3	ETHZ	Eidgenassische Technische Hoschschule Zürich
4	UNIBO	Universtità di Bologna
5	STM	STmicroelectronics
6	EPFL	École polytechnique fédérale de Lausanne

Version: v1.2 – Public

Versions of the Document

Version	Date	Author	Comment
v0.1	2011-03-09	Martino RUGGIERO	First draft.
v1.0	2011-03-15	Daniele BORTOLOTTI	Second draft.
v1.1	2011-03-30	Martino RUGGIERO	Final version.
v1.2	2011-03-31	Martino RUGGIERO	Public version.

Contributors

Daniele BORTOLOTTI (UNIBO), Martino RUGGIERO (UNIBO),

Andrea MARONGIU (UNIBO), Luca BENINI (UNIBO).

Table of Contents

Table of Figures

1 Introduction

Modern multiprocessor embedded systems are heterogeneous and very complex platforms. The interconnection architecture plays a key role in these systems, being responsible not only for communication between the various elements but also for the management of fundamental mechanisms in a multiprocessor environment, such as shared resources utilization and performance scalability.

There are several approaches for interconnection fabric design: the classical shared bus structures are flanked by a crossbar, ring, and the emergence paradigm of Network-on-Chip [2]. Each design approach has advantages and drawbacks in terms of complexity and performance. For example, the shared bus is the easiest to design, but quickly becomes limiting in terms of bandwidth and latency, when scaling to a high number of bus masters or cores. The crossbar allows communication between several IP modules, but increasing of area makes this approach often not feasible. The Network-on-Chip paradigm instead provides good scalability with regards to the number of processors, providing at the same time high bandwidth, modularity and a high degree of customization; all at the expense of design complexity.

The main platform architecture developed in PRO3D project is depicted in Figure 1. It can be considered a derivation of the P2012 platform template, described in D7.1. Several computation tiles are connected via a mesh NoC.

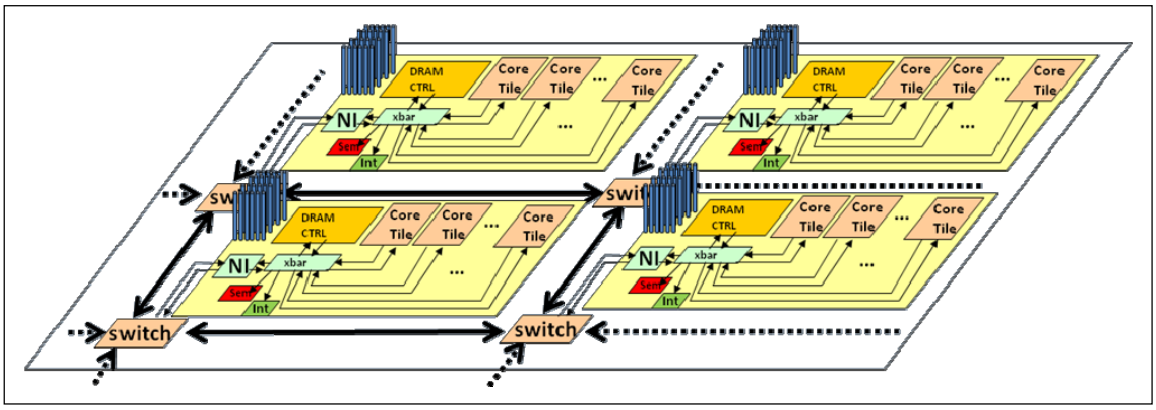


Figure 1: PRO3D Target Architecture

The main computation tile is described in Figure 2. The computation tiles are supposed to be homogeneous and consist of a DRAM memory controller, a network interface, a crossbar, a set of devices for efficient intra-tile synchronization and a variable number of core tiles.

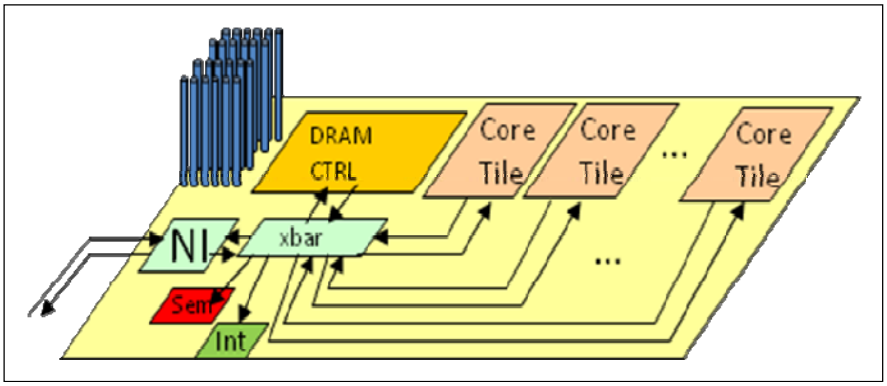


Figure 2: Computation Tile

In the context of PRO3D project, MPARM [1] virtual platform will be used as main tool for design space explorations. MPARM is a virtual SoC platform based on the SystemC simulation kernel, which could be used to model both HW and SW of complex systems. It has been developed by UNIBO for the past 7 years and has been shared with more than 20 research partners, who have substantially contributed to its evolution. The classical system architecture simulated by the default MPARM distribution is represented by a homogeneous multicore system based on shared bus communication. During PRO3D project, MPARM will be enhanced with several HW parametric models of the main micro-architectural components of a 3D integrated inter-connect and memory hierarchy. The new 3D models will be highly parametric and customizable.

The present deliverable reports the work done in WP1, which is focused in deploying the modelling infrastructure to simulate thermal and functional features of the PRO3D stacks under dynamic working conditions, i.e., during the execution of real parallel application execution.

The following sections describe our modelling approaches and provide the technical details on the implementation work that has been conducted for developing and characterizing the interconnection model in the proposed PRO3D platform. More in details, Section 2 describes the implementation of:

- The crossbar module within a computational tile (Section 2.1);
- The cycle accurate Network-on-Chip model (Section 2.2);
- The overall tile model (Section 2.3);

Section 3 discusses the profiling and characterization of the target communication architecture, while Section 4 motivates and describes the porting of a flit-level Network-on-Chip model. Finally, Section 5 highlights the main conclusions.

2 Communication architecture modelling

In this section basic ideas and implementation details for the modelling of the crossbar are discussed, also the porting of a SystemC [3, 4] model of Network-on-Chip in MPARM virtual platform is explained.

The various components of a single cluster (also called TILE) are briefly described as well as how the address space is managed, both locally (inside a single cluster) and globally (a clustered architecture interconnected by a NoC).

2.1 Crossbar

2.1.1 SWARM Communication Protocol

The processing modules of the system are represented by cycle accurate models of cached ARM cores. The module is internally composed of the ARM CPU, the first-level cache and peripherals (UART, timer, interrupt controller). It was derived from the open source cycle accurate SWARM (software ARM) simulator encapsulated in a SystemC wrapper. The insertion of an external (C++) ISS is subject to the necessity of interfacing it with the SystemC environment (for example, accesses to memories and interrupt requests must be trapped and translated to SystemC signals). Another important issue is synchronization, the ISS, typically written to be run as a single unit, must be capable of being synchronized with the multiprocessing environment (i.e. there must be a way to start and stop it maintaining cycle-accuracy). Last, as a further requirement, the ISS must be capable of being multi-instantiable (for example it must be a C++ class), since there will be one instance of the module for each simulated processor.

The SWARM simulator is entirely written in C++. It emulates an ARM CPU and is structured as a C++ class which communicates with the external world using a Cycle function, which executes a clock cycle of the core, and set of variables in very close relation to the corresponding pins of a real hardware ARM core. Along with the CPU, a set of peripherals is emulated (timers, interrupt controller, UART) to provide support for an Operating System running on the simulator.

The cycle-level accuracy of the SWARM simulator simplifies the synchronization with the SystemC environment (i.e. the wrapper module), especially in a multiprocessor scenario, since the control is returned to the main system simulator synchronizer (SystemC) at every clock cycle. The interesting thing about ISS wrapping is that with relatively little effort, other processor simulators can be embedded in our multiprocessor simulation back-bone. Provided they are written in C/C++, their access requests to the system bus need to be trapped, so to be able to make the communication extrinsic and generate the cycle accurate bus signals in compliance with the communication architecture protocol. Moreover, the need for a synchronization between simulation time and ISS simulated time arises only when the ISS to be embedded has a coarse time resolution, i.e. when it does not simulate each individual processor clock cycle. Finally, the wrapping methodology determines negligible communication overhead between the ISS and the SystemC simulation engine, because the

ISS does not run as a separate thread and consequent communication primitives are not required, that would otherwise become the bottleneck with respect to the simulation speed.

We want here to describe the interface of the SWARM core encapsulated in a SystemC module (wrapper), it is important to describe the mechanism of master / slave communication because this has been the starting point of our work. A description of the internal behaviour of the core can be found in [5]. As shown in Figure 3, the interface of the wrapper consists of three interfaces: request (output), ready and extint (inputs) and a bidirectional interface called "PINOUT". A header file defines the core structure of the communication interface, the structure is shown below:

```
struct PINOUT
{
      bool nreset;
      uint32_t address;
      uint32_t data;
      bool rw;
      bool fiq;
      bool irq;
      bool benable;
      uint32_t bw;
      uint32_t burst;
};
```

Where the main fields are:

- `address` : 32-bit field for addresses;
- `data` : 32-bit field for data;
- `rw` : boolean value for READ (rw = 0) and WRITE (rw = 1) operations;
- `burst` : 32-bit field (only 5 used) for burst size;
- `benable`: bus enable, boolean value for bus interaction (asserted to communicate with the external world);
- `bw`: 2-bit field to set WORD size ('0' = 4B, '1' = 1B, '2' = 2B, '3' = UNDEF);

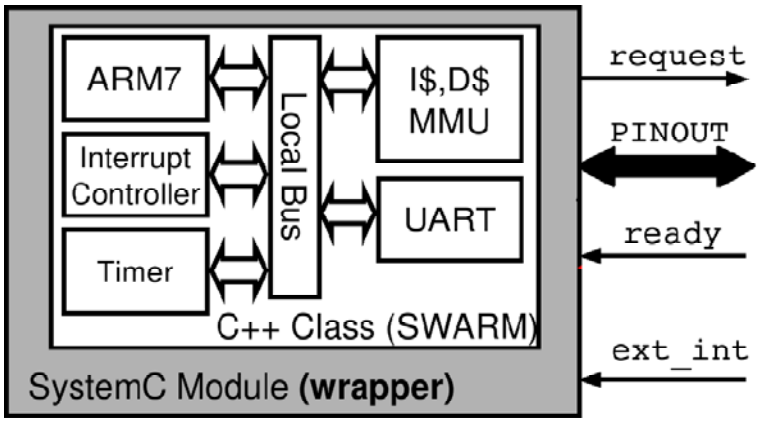


Figure 3: SWARM Interface

In Figure 4 are shown some transactions between a SWARM processor and a PINOUT slave, connected through proper signals.

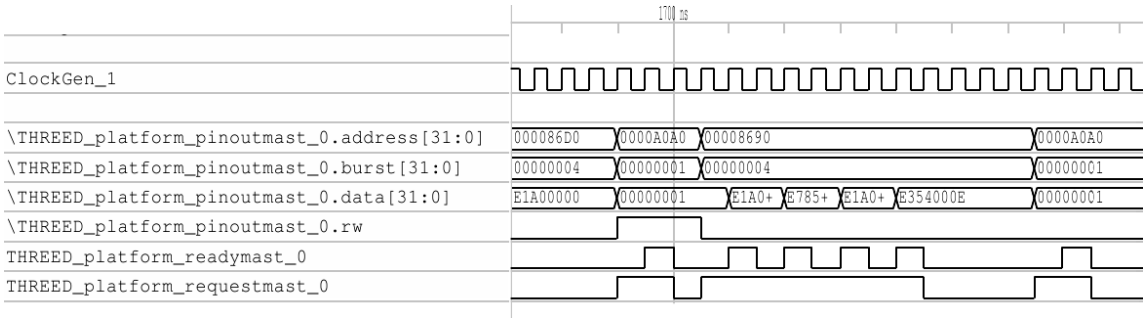


Figure 4 : SWARM memory transactions

Presented here is a sequence of possible operations when a SWARM core accesses memory: SINGLE WRITE, BURST READ (with a maximum burst of 4) and SINGLE READ. The study of the communication protocol between core and slave has been the basis for the implementation of the crossbar, which is described in the next section.

2.1.2 THREED crossbar module

The crossbar has been designed as a highly flexible module, capable of connecting an arbitrary number of masters and slaves, one may think of this component as a multiplexer N:M, where N cores can be interfaced to M slaves. The module THREED-xbar can be considered as a full crossbar, allowing communication between any master and any slave.

Interface

In order to instantiate a dynamic module, the gates to interface THREED-xbar are declared as pointers:

```
sc_in<bool> clock;

sc_in<bool> *request_from_core;
sc_out<bool> *ready_to_core;
sc_inout<PINOUT> *pinout_core;

sc_out<bool> *request_to_slave;
sc_in<bool> *ready_from_slave;
sc_inout<PINOUT> *pinout_slave;
```

As one may see, in addition to the system clock, there are symmetrical ports, a master side and a slave side. The creation of the actual number of ports is dynamically determined when the constructor of the class THREED-xbar (SystemC modules are members of `sc_module` class) is invoked in the act of creating an instance.

Several parameters are passed to the THREED-xbar, among them the most relevant are:

- `nm`: name of the module (mandatory);
- `ID`: id of the module;
- `num_cores`: number of master ports;
- `num_slaves`: number of slave ports;
- `delay`: latency introduced by the module (expressed as number of clock cycles, minimum is 1).

These parameters can be set for a single instance, offering a fine-grained control.

Methods, signals and internal variables

The crossbar implements 3 different methods, each one associated with a sensitivity list, which determines the behaviour of the module:

```
//this is the module constructor
...
public:
void req_polling(int index);
void arbiter();
void fsm(int index);
...
```

The method `req_polling()` is in charge of polling master ports to check whether they have raised the request signal (`request_from_core`) or not. The method `fsm()`interfaces with a given slave port, it implements a simple state machine to handle the communication protocol between masters and slaves. The real functionality of the crossbar is within the `arbiter()` method, this determines the slave port a master wants to access after operating address decoding, it is also responsible for conflicts management deploying different scheduling policies.

The module uses different signals and internal variables, for the communication between the various methods and to implement their functionalities.

```
sc_signal<bool> *go_fsm;
sc_signal<bool> *served;
enum ctrl_state { IDLE=0, READ=1, WRITE=2 };
ctrl_state *cs;
unsigned char *idc_fsm;
unsigned char *next_to_serve;
bool *req;
bool *busy_fsm;
```

In a similar way as described above for the interfaces, their creation is dynamic and is done in the process of creating an instance:

```
go_fsm = new sc_signal<bool> [num_slaves];
cs = new ctrl_state [num_slaves];
```

```
idc_fsm = new unsigned char [num_slaves];
next_to_serve = new unsigned char [num_slaves];
req = new bool [num_cores];
served = new sc_signal<bool> [num_cores];
busy_fsm = new bool [num_slaves];
```

The functionality of the methods, and their interaction with signals and variables, will be detailed in the next section.

Figure 5 shows a block diagram describing the internal structure of the module, follows a description of the various methods explaining the functional behaviour of the crossbar.

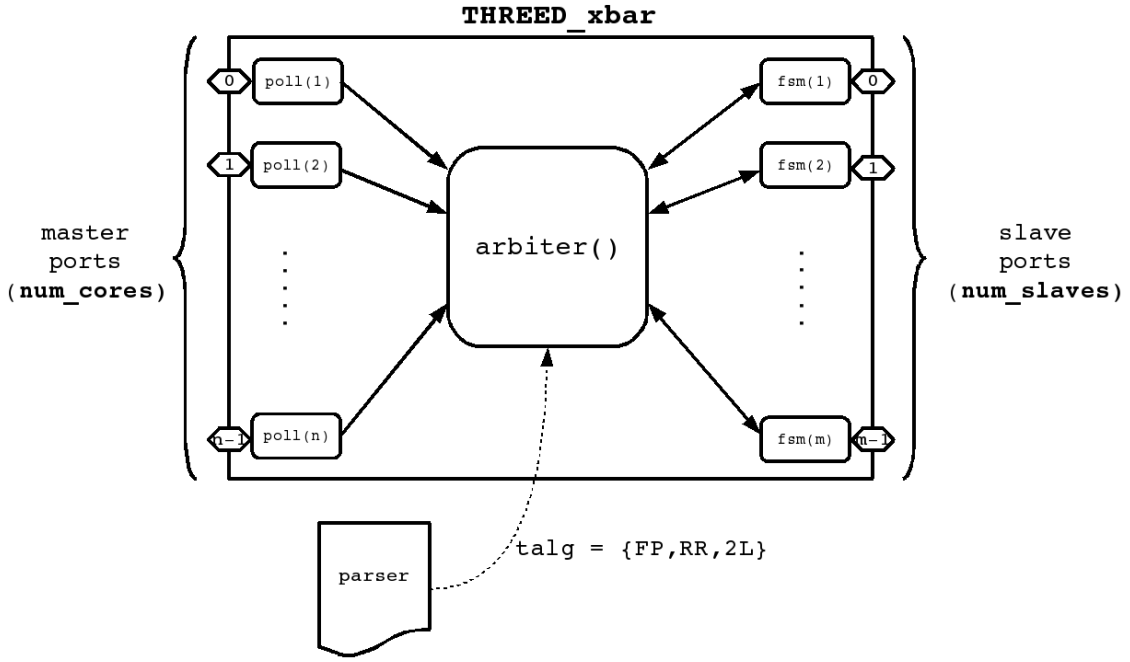


Figure 5: Internal structure of crossbar module

- **`req polling(int index)`**: the polling methods are employed to check, at every clock cycle, if the master port identified by `index` has the request signal logically high or low. If there is a request by a core, the corresponding variable `req[index]` is set to '1', the process of polling is then suspended until such variable is set to '0' by the `arbiter()` , this is done once the core has been served.
- **`fsm(int index)`**: This method handles the flow of data between a slave port (identified by `index`) and a master port, following a decision of the arbiter. For the management of read and write transactions, this component is modelled by a simple state machine, whose possible states are described by data type `ctrl_state` and variables `cs[i]` for the current state:

```
enum ctrl_state { IDLE=0, READ=1, WRITE=2 };
```

```
ctrl_state *cs;
```

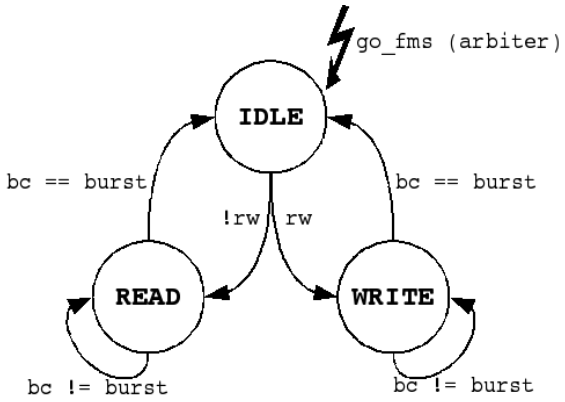


Figure 6: State diagram for fsm() method

Figure 6 shows state diagram for `fsm()` method. The arbiter, after carrying out an address decoding and checking that the i-th state machine is free (`busy_fsm[i] != 1`), sets the index of the master port to serve (`idc_fsm[i]`) and activates the state machine with the signal `go_fsm[i]`. In this manner, a particular state machine is servicing a core, when the transaction will be over, it will be released.

When the signal `go_fsm[i]` is set by the arbiter, the state machine is activated from idle, when waking up, it reads the pinout interface of the core to be served (`pinout_core[idc_fsm[i]]`). Depending on the `rw` field, the next state will be READ or WRITE, and after having replicated core's pinout field at the slave port (`pinout_slave[i]`), asserts the request to the slave device by raising the signal `request_to_slave_[i]`. When in READ state, the `fsm[i]` is waiting for a number of data transactions equal to the burst field (read at the master port to be served), at this stage there is a sequence of handshakes between fsm and slave device, according to the asynchronous request / ready protocol. Any received data is immediately made available at the master port, with no additional latency.

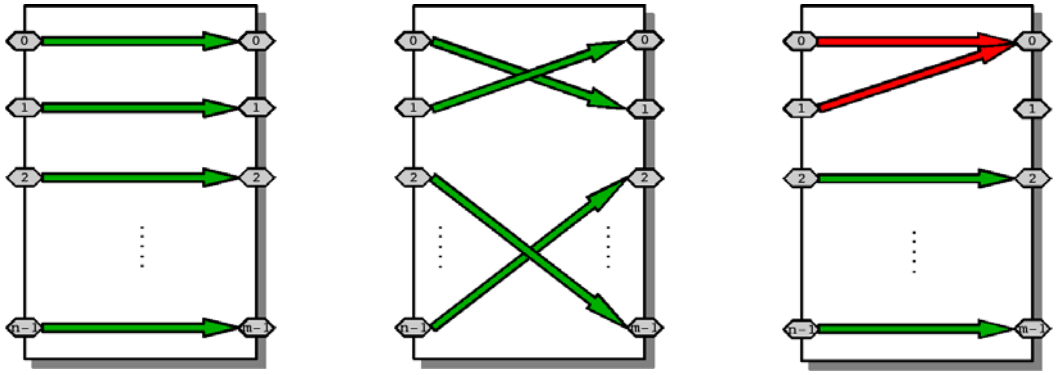
In case `rw` is equal to 1, the state machine moves to the WRITE state where, with a similar mechanism, waits for the slave to confirm the write has took place setting the `ready_to_core` signal to '1'.

Whether you are in READ or WRITE state, at the end of the transaction (transaction counter bc is equal to burst) the state machine declares itself free (`busy_fsm[i] = 0`), states the core has been served (`served[idc_fsm[i]] = 0`), lower `go_fsm[i]` and returns to IDLE.

- **`arbiter()`**: this is the method responsible of controlling the crossbar, it is activated at every positive edge of the clock. When activated, the arbiter checks for requests from the cores by looking at the array of registers written by the polling method. If

there is a pending request (`req[i] = 1`) and that core has not been served (`served [i] = 0`), the boolean table of requests (`req_table`) is filled out, where each slave is associated with the master which is requiring access, this mechanism is based on the fact that each slave port (and the device connected to it) is logically mapped into a range of addresses. When the crossbar is instantiated, a particular method executed once by the constructor, `LoadSlaveTable()`, is responsible to dynamically load a table, where the range of addresses associated with each slave port is defined. In this way, the arbiter can determine which slave a master wants to access, operating an address decoding.

Once the table of requests has been filled, the arbiter has a complete map of what masters want to do. If there is no contention, the crossbar is designed to have a state machine active for each slave port, ensuring that there are more active communication channels in parallel (as shown in Figure 7)



(a) parallel (b) crossed (c) conflict

Figure 7: Parallel communication and conflicts

When, in the same clock cycle, two or more masters are requesting access to the same slave port there is a contention of the resource (as in Figure 4.c). We have implemented three different scheduling algorithms for the management of conflicts:

- **Fixed Priority**: the priority is inversely proportional to master port's ID, there is no pre-emption, and a core with lower ID is always privileged.
- **Round Robin**: the allocation of the shared resource is circular. Once a core has been granted the access, is then pre-empted. A subsequent request to the same slave is queued, and treated as the lowest priority request.
- **2 Levels**: combines the two algorithms above on two levels of priority, in case of conflict higher priority is given to the core with lower ID, while on the other non-conflicting cores a round robin is used.

Latency

The crossbar is a synchronous module, but the abstraction level to which is modelled does not take into account the tight timing typical of a pure RTL description. Overall, the default latency introduced by the crossbar is a clock cycle, in order to provide flexibility a `delay` parameter is passed to the constructor of the class, allowing to vary the crossing time.

2.2 Network-on-Chip

In this section we describe the implementation of the NoC models in MPARM; follows a description of the bridges, fundamental modules in order to enable communication between clusters.

2.2.1 NoC Mesh in MPARM

We created different SystemC NoC architectures in many different platforms. All the platforms have a mesh topology but they differ, however, in the number of elements that are able to interconnect. With mechanisms similar to those described in the previous section, it is possible to instantiate the `noc_2x2_platform`, `noc_3x3_platform` and `noc_4x4_platform`.

Each of these platforms hierarchically instantiates all the components of a Network-on-Chip (switch, link, network interface, etc.) and has two network interfaces for each node that can connect. In Figure 8: 2x2 NoC interconnection is shown a schematic view of the 2x2 architecture, a module like this is only an interconnection system, without any active modules (masters) or passive (slaves) generating traffic.

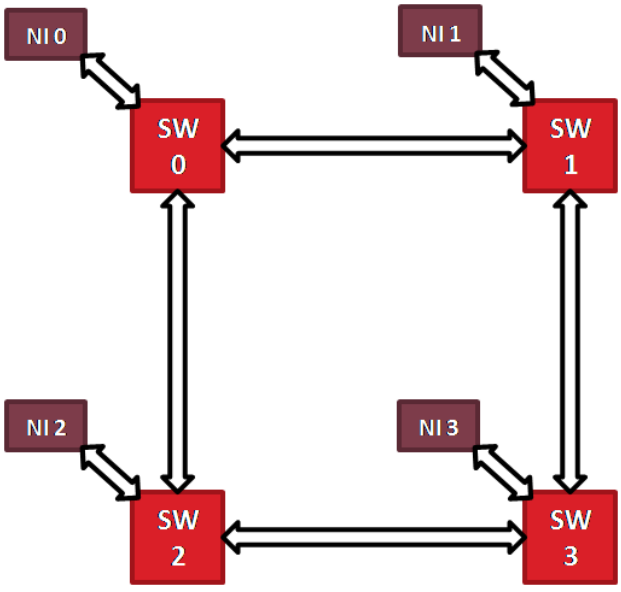


Figure 8: 2x2 NoC interconnection

The description of the internal structure is beyond the scope of this work; however it is important to explain the mechanism of addressing in the network. Each node is connected to the network via a Network Interface (NI), INITIATOR for masters (cores or traffic generators) and a NI TARGET to communicate with slave devices (typically memories), both interfaces have a standard interface OCP [7], as shown in Figure 9.

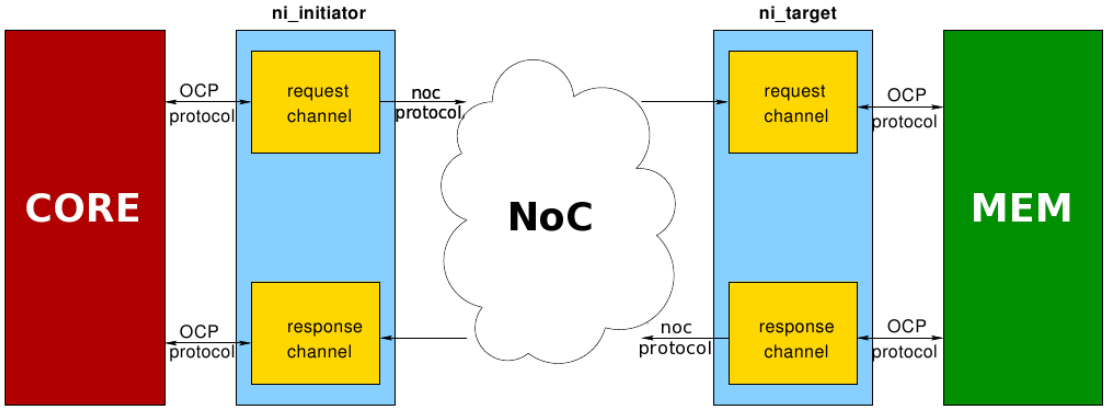


Figure 9: Network Interfaces and NoC abstraction

When a master connected to the NI INITIATOR inject traffic in the network, the NI is responsible for creating the packet to be sent by converting the OCP protocol to the network protocol. A generic transaction is physically split into several blocks (flit), which logically can be of three types: header, payload and tail. In particular, the header (whose structure is outlined in Figure 10) contains the information necessary for the proper routing of the packet; both in request and response phase, and this information are calculated according to the most significant byte (MSB) of master's OCP address field (`Maddr`).

Routing	Source	Address	Command	Burst Control

Figure 10: NoC flit fields

Referring to the architecture depicted in Figure 8, in order to drive traffic node 2 (NI 2) a given core needs to use addresses between `0x20000000` and `0x2FFFFFFF`. It is therefore the first hexadecimal digit to discriminate the range of addresses in which a node is mapped.

2.2.2 Bridges

In this section we present the modules responsible for communication between the cluster and network-on-Chip. As will be seen in the next section about the TILE, local (within a tile) communication within this, core and slave devices communicate via the pinout interface, the Network Interface (NI) instead have a standard OCP interface that uses different signals. To allow the sending and receiving packets on the NoC, we have implemented two modules that operate protocol conversion.

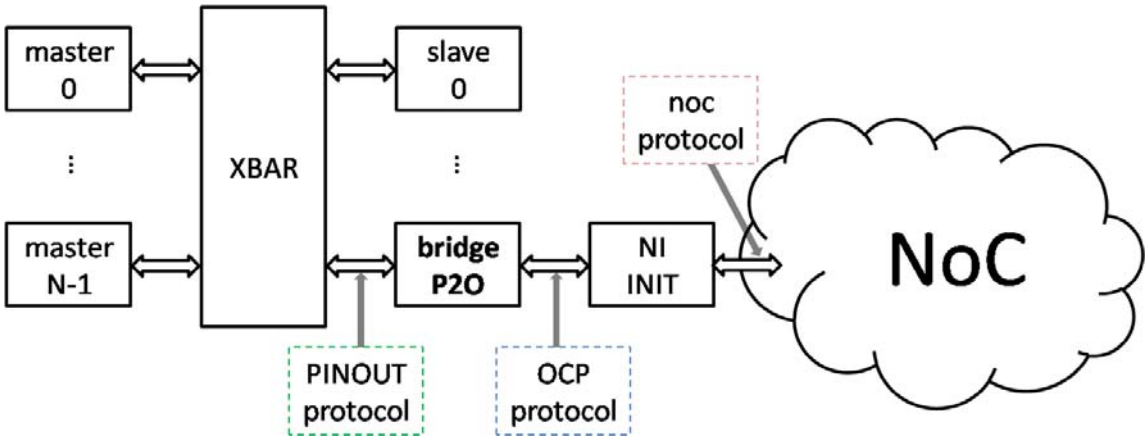


Figure 11: Role of bridge PINOUT / OCP

Bridge OCP / PINOUT

This component is responsible for converting an OCP interface type in an interface PINOUT to emulate the communications protocol processor SWARM. This component is in fact designed to serve as a bridge between a master port and a TARGET NI's crossbar (as illustrated in Figure 11). In this way, packets addressed to the network are translated and presented to the crossbar with the same modes of a local master.

2.3 The TILE module

The TILE module, as already mentioned in the introduction consists of a computational node, organized in clusters, inside we have an arbitrary number of SWARM cores, different kind of memories (private, scratchpad, shared-3D), a crossbar for local communication and two bridges working as interface for the Network-on-Chip. A simplified scheme of this module is given in Figure 12.

This section describes the modelling of slave devices within a single tile, THREED-tile SystemC module is described and a description about the address space management is given.

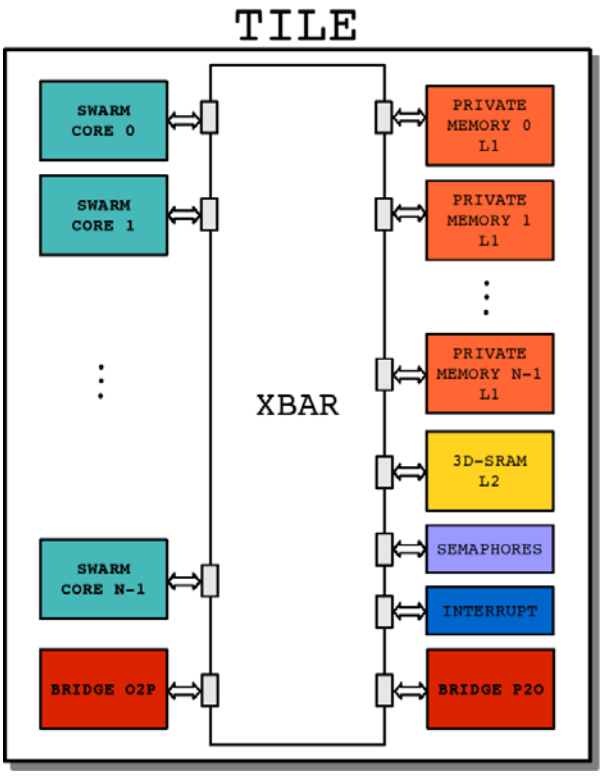


Figure 12: Internal structure of THREED-tile module

2.3.1 Slave Devices

In this section we present the internal behaviour of the slave devices that can be instantiated within a tile, here is a listing of them:

- Private memories;
- 3D-SRAM memories;
- Scratchpad memories;
- Semaphore devices;
- Interrupt devices;

Interface and common behaviour

Each one of the available slave devices listed shares a common interface, except the interrupt device as will be explained later, able to handle request/ready protocol:

```
sc_in<bool> clock;
sc_in<bool> reset;
sc_in<bool> request_from_wrapper;
sc_out<bool> ready_to_wrapper;
sc_inout<PINOUT> pinout;
```

Communication with the outside world is enabled by ports symmetrical with respect to SWARM core ports, in this way there is not an unnecessary protocol translation, granting a quick access.
To ensure the configurability of the modules, when an instance is invoked some key parameters can be passed to the constructor in order to characterize the memory such as:

- `ID`: global ID number;
- `START_ADDRESS`: starting logical address of the device;
- `TARGET_MEM_SIZE`: ending logical address of the mapping range;
- `MEM_IN_WS`: access latency, expressed in number of clock cycles;
- `MEM_BB_WS`: latency between every burst beat (number of clock cycles).

All slave devices are internally implemented as memory modules with different behaviours in case of reads and writes, determining how they operate. Inner memory functionality is implemented within the C++ class `mem_class`, already used in MPARM, offering `mem_class::Read` and `mem_class::Write` methods, address checking, word size handling (WORD, HALFWORD, BYTE) granting proper values are written and read.

Private memories

The private memories have been designed as fast SRAM on-chip, with small size and low latency (typically a few cycles), in the hierarchical multi-level memory system, these are first level (L1) memories. When creating every private memory module, the `mem_class` is responsible of uploading the code with a bitwise copy of the binary, enabling each processor to fetch its code.

3D shared memories

The 3D memories have been designed as fast SRAMs arranged on a layer different from the core's one (see Figure 13), of bigger size than the private memories and a higher level (at least L2) in the hierarchy. The configurability offered to this module allows user to control the size and access time of in terms of latency.

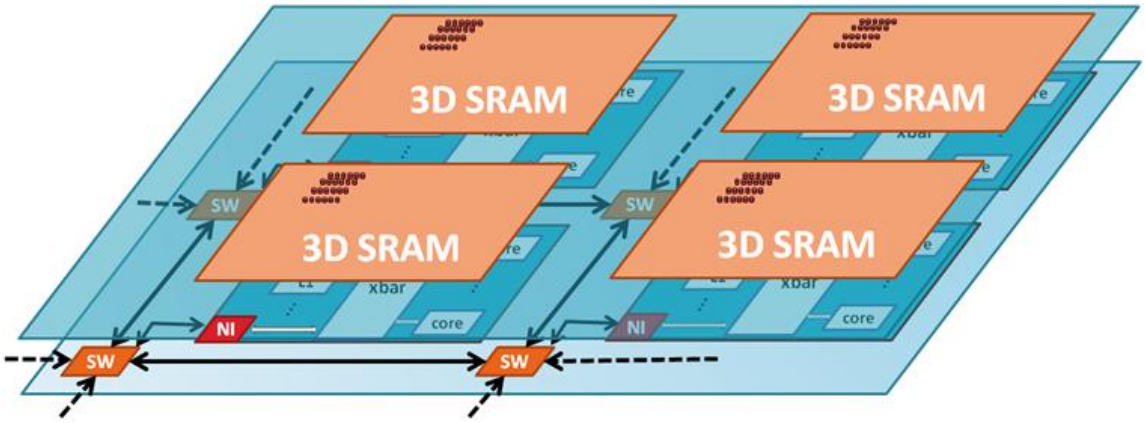


Figure 13: Layered architecture

Scratchpad

Scratchpad memories are used to simplify the cache coherence management in multiprocessor systems. In a multiprocessor system, cache management becomes a pressing problem having to ensure they do not contain obsolete data, so each time a processor accesses data that is also present in the cache of another processor, the data block in the cache memory needs to be invalidated, and this leads to a high dataflow between the processors, slowing down the system. A scratchpad memory on the other hand, is directly addressable so if a processor wants to access certain data in a scratchpad of another processor may do it with no need for invalidation. This simplifies hardware and reduces the number of synchronization messages between the processors. Just like cache memories, scratchpads are based on the principle of locality, and thus if a processor copies the data that uses frequently in its scratchpad can significantly reduce the accesses to slow second or third level memory, without the need to use a cache. The address space reserved for a scratchpad memory is partitioned: a portion is dedicated to the memory itself and a smaller part to the access queue.

Semaphores

The presence of multiple cores in the same cluster brings the need for synchronization mechanisms in accessing shared resources, otherwise it is not possible to coordinate the activities of processors and ensure proper execution. These mechanisms are exploited at the software level; however, to provide such abstractions is required to model hardware behaviour.

The semaphore consists of a series of registers, accessible through the crossbar as a generic slave, associating a single register to a shared data structure. By using a mechanism such as a hardware “test & set”, we are able to coordinate access: if the reading at a particular address returns the value '0 ', the resource if free and accessible and the semaphore automatically locks it, if it returns a different value, typically '1 ', access is not granted.

Interrupt

Each SWARM core of a single tile is sensitive to a configurable number of interrupts. The THREED-interrupt module is responsible of inter-processor communication through the appropriate signals `ext_int`: an access to a particular address in this module involves sending interrupt to other processors. The interface of this module is the same as the other slave devices, except for the signals that connect this device to the cores:

```
* sc_inout <bool> extinct;
```

For this device user can set the size (which depends on the number of cores and the number of interrupts that can handle), to which logical address is mapped. Only write operations are possible on this device; the mechanism for sending interrupts is shown below:

```
void Write(uint32_t addr, uint32_t data, uint8_t bw)
{
      intno = ((addr - START_ADDRESS) / 4) - 1;
      if (intno < 0 || intno >= num_cores)
      {
            exit(1);
      }
      wait();
      extint[intno].write(true);
      wait();
      extint[intno].write(false);
}
```

To understand how the overall mechanism works, Figure 14 shows the situation where the interrupt device is mapped at address 0x01000000 and core 0 issues a write at address 0x0100000A.

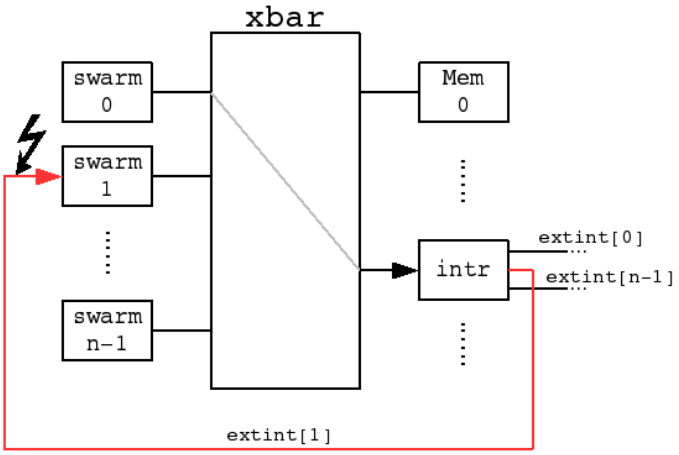


Figure 14: Interrupt mechanism

2.3.2 Local Address Space

Address space management, i.e. how the various components are addressable by the processors, is a key part of the overall structure. The SWARM core has a 32-bit address field, it is then capable to address 2^{32} memory locations equal to 4GiB. A single cluster is associated with an address space (TAS - Tile Address Space) equal to 0x10000000 (256 MiB) and, as already described, each slave port of the crossbar is mapped in a range of addresses, so the organization of local space depends on the elements that are instantiated within a cluster.

Figure 15 represents the address space of a tile; the first hexadecimal digit of the address identifies the tile, so the figure regards the tile with TID equal to 0.

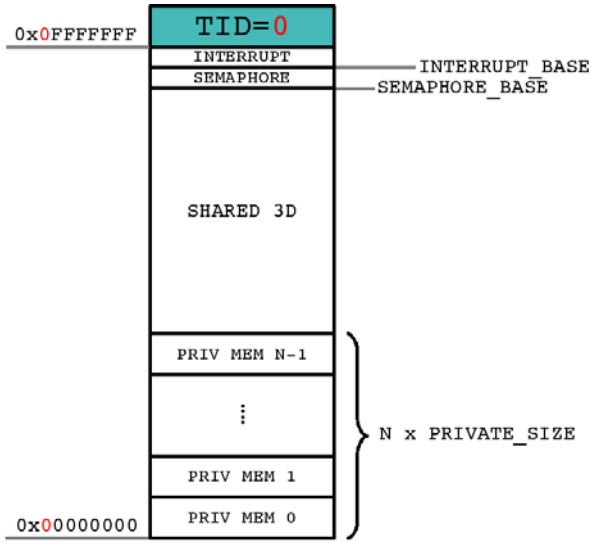


Figure 15: TILE 0 Address space

The size of some devices, such as private memories, semaphores and interrupts is static and is defined in a configuration file. The size of the 3D shared memory is instead calculated at runtime and fills a portion of the address space left empty by other elements. The default configuration includes:

```
TILE_SPACING 0x10000000 (256 MB)
TILE_PRIVATE_SIZE 0x01000000 (16 MB)
TILE_SEM_BASE 0x0FF00000 (255 MB)
TILE_SEM_SIZE 0x00008000 (32 KB)
TILE_INT_BASE 0x0FF09000 (255MB + 36K)
TILE_INT_SIZE 0x00000200 (512 B)
TILE_CORESLAVE_BASE 0x0FF10000 (255MB + 64K)
TILE_CORESLAVE_SPACING 0x0001E000 (120 KB)
TILE_CORESLAVE_SIZE 0x00015000 (84 KB)
TILE_SCRATCH_SIZE 0x00010000 (64 KB)
TILE_QUEUE_SIZE 0x00004000 (16 KB)
```

Figure 12 does not show scratchpad memories, for the sake of simplicity, and because these are optional, they can be enabled passing a parameter (CORESLAVE) to the simulator. In case user wants to organize a cluster as a single 3D memory module, passing `L3_MEM_ONLY = true` to the constructor of the tile, the entire space of 256 MB is automatically dedicated to the shared-3D. Despite the size of the various components of a tile is defined statically, you can vary the size in a simple manner in order to study the behaviour of the system varying these parameters.

As you can see, the P2O bridge is not mapped in any address range, this choice was dictated by the fact that the single tile is identified by the first digit hexadecimal address field, so an access to a non-local address is directly conveyed to the bridge by the crossbar.

2.4 Global Address Space

This section presents an example of an architecture that can be instantiated and the communication mechanism among the tiles. Figure 16 shows a platform where the interconnection system is a 3x3 NoC and each network interface is connected to a tile, identified by a TID (Tile ID). Every single tile can be configured for all parameters described above.

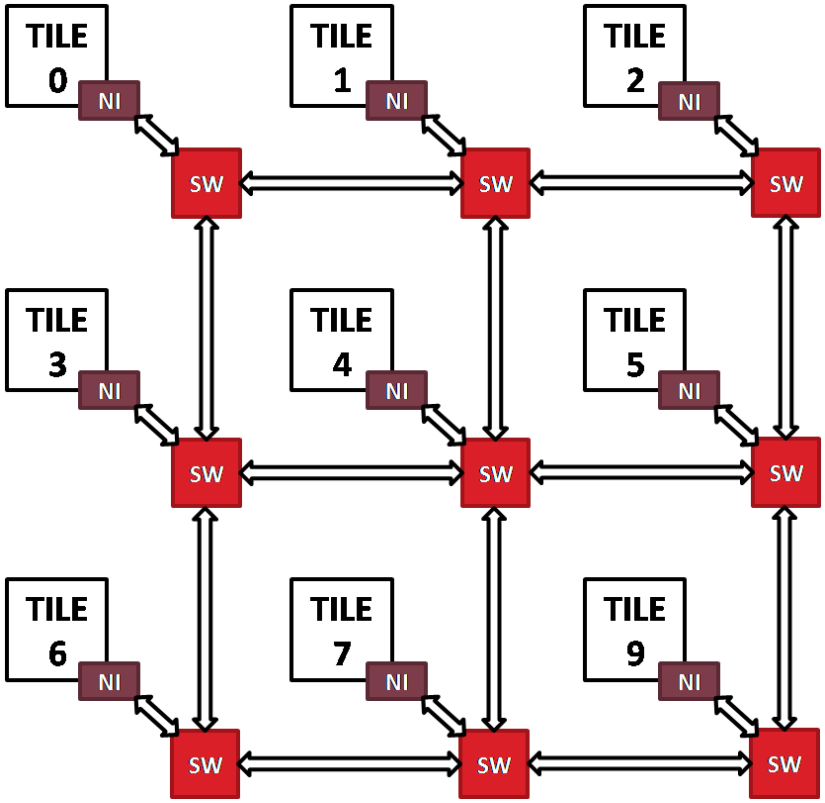
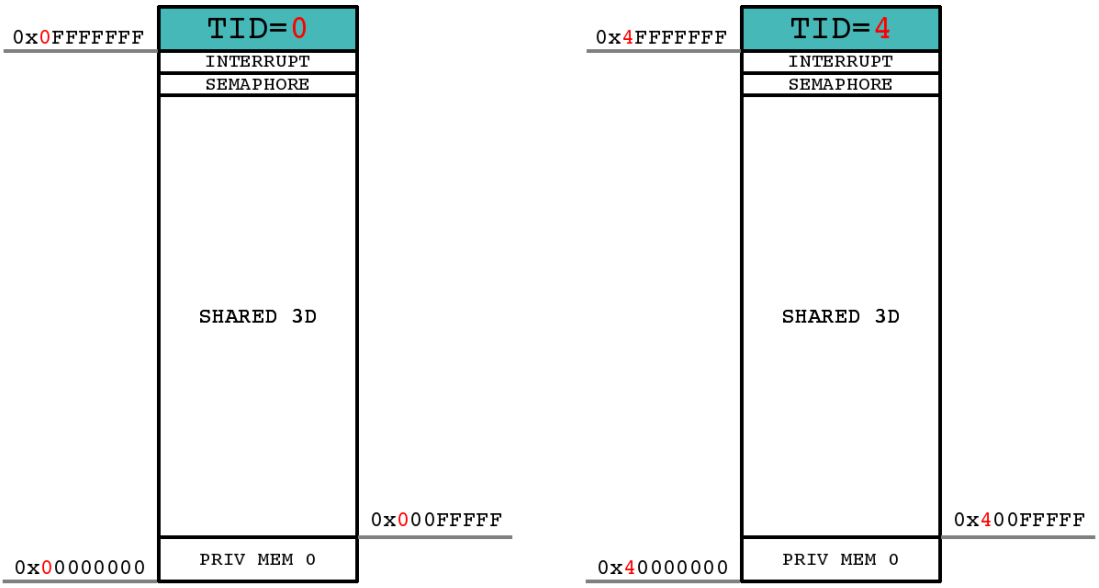


Figure 16: Architecture composed of 9 tiles interconnected by a 3x3 mesh NoC

As already outlined, the first hexadecimal digit of the address identifies the individual TILE, while the other bytes are meaningful locally and interpreted by the crossbar to allow communication between different local devices. For example, Figure 17 shows two possible organizations of the address space for the tile with TID 0 and 4.



(a) TAS 0 (b) TAS 4

Figure 17: Address Spaces for Tile 0 and 4

With this configuration, if a processor inside tile 0 wants to access the 3D shared memory of tile 4, for example by reading at 0x4A00000, the crossbar inside tile 0 will connect the processor and the network interface through the P2O bridge, while the crossbar inside tile 4 will receive a read request from its O2P bridge at 0x4A000000. This global address management can ensure both local (**intra-tile**) and remote (**inter-tile**) communication.

3 Communication architecture characterization

3.1 Introduction

This section discusses the simulations conducted to test and analyze the behaviour of the modelled interconnection matrix using the MPARM simulator. During the modelling phase, for each component were carried out tests to validate the targeted behaviour, once verified the functional correctness of the whole platform, clustered architecture were simulated. Particular attention was paid to the properties of system interconnect scalability, by studying the behaviour of NoC and crossbar, varying key parameters such as:

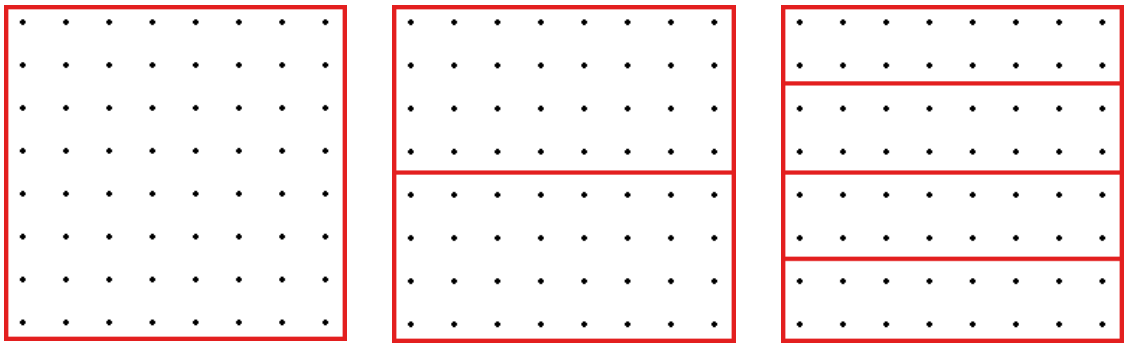
- Number of processors within a single tile;
- Access policies to shared resources (fixed priority, round robin and 2 levels priority);
- Access latency to 3D shared SRAM memory;
- Number of "active" tiles in the platform.

3.2 Experimental Results

3.2.1 Crossbar Scaling - no contention

The first scenario consists of a workload on the system that has no synchronization point and therefore does not imply any kind of communication between processors.

This is obtained by performing a matrix multiplication, distributing the workload among processors independently; to do this each core operates exclusively in its own private memory. As the number of cores involved in the matrix product increases, the portion of the computation (chunk) for each one is gradually reduced, Figure 18 summarizes the portion of the matrix related to each processor in the case of 1, 2 and 4 active cores.



(a) (b) (c)

Figure 18: Workload distribution among cores in case of 1, 2 or 4 active cores

A good empirical compromise between simulation time and granularity in the distribution of the workload, leads us to choose matrices of size 32x32, we could then operate in parallel up to 32 cores. Since both the input matrices and the output one are allocated in each core's private memory, there is no situation of contention and the crossbar is able to offer a dedicated channel to each of them.

The vertical axis of the chart of Figure 19 shows the relative execution time (normalized to the case of a single processor that takes care of the whole matrix multiplication), while on the horizontal axis varies the number of active processors. The two trends shown here represent the simulated case (real) and the ideal case (ideal), with a trend like 1/N. As you can see the difference from the ideal trend is negligible, due to edge effects, with a maximum difference in relative execution time of 0.07% (8 cores situation).

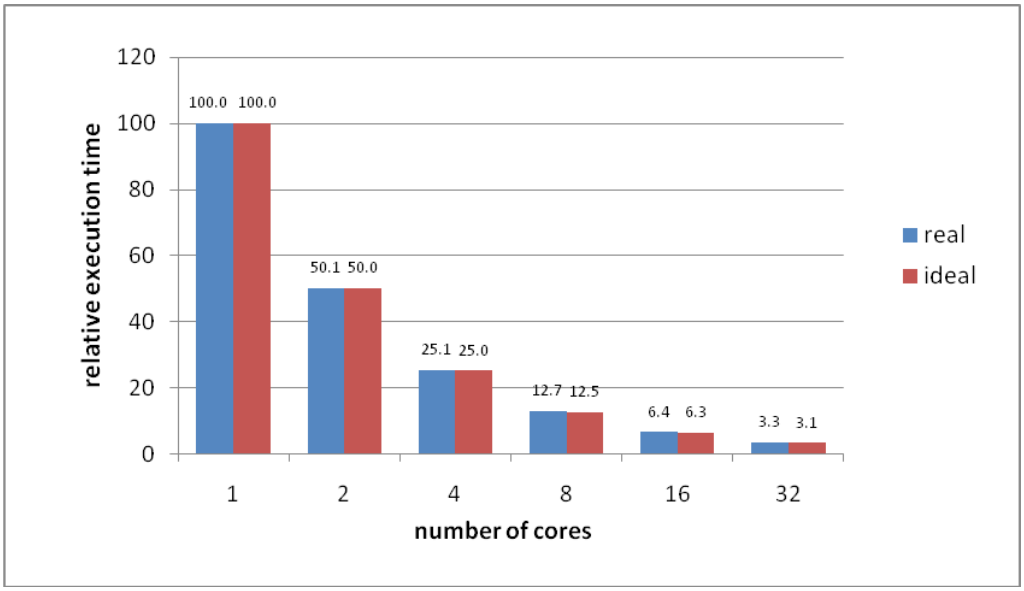


Figure 19: Crossbar scaling – no contention

This test was mainly aimed at verifying the behaviour of the crossbar in the situation of a large number of processors with independent communication; the results show that the desired behaviour is obtained.

3.2.2 Crossbar Scaling - local contention

The goal of this setup is to study the behaviour of the crossbar in case of contention for shared resources. To carry out this analysis, a benchmark consisting of a matrix multiplication has been used, where input matrices are in private memory, and the output one is located in the shared memory (3D). Unlike the previous test, here we do not want to study the properties of parallelism in the workload, because the load is constant for each processor, keeping always 8 active cores, each one responsible of 1/8 of the total load. In this situation there is no traffic on the NoC because all the processors are within a single cluster.

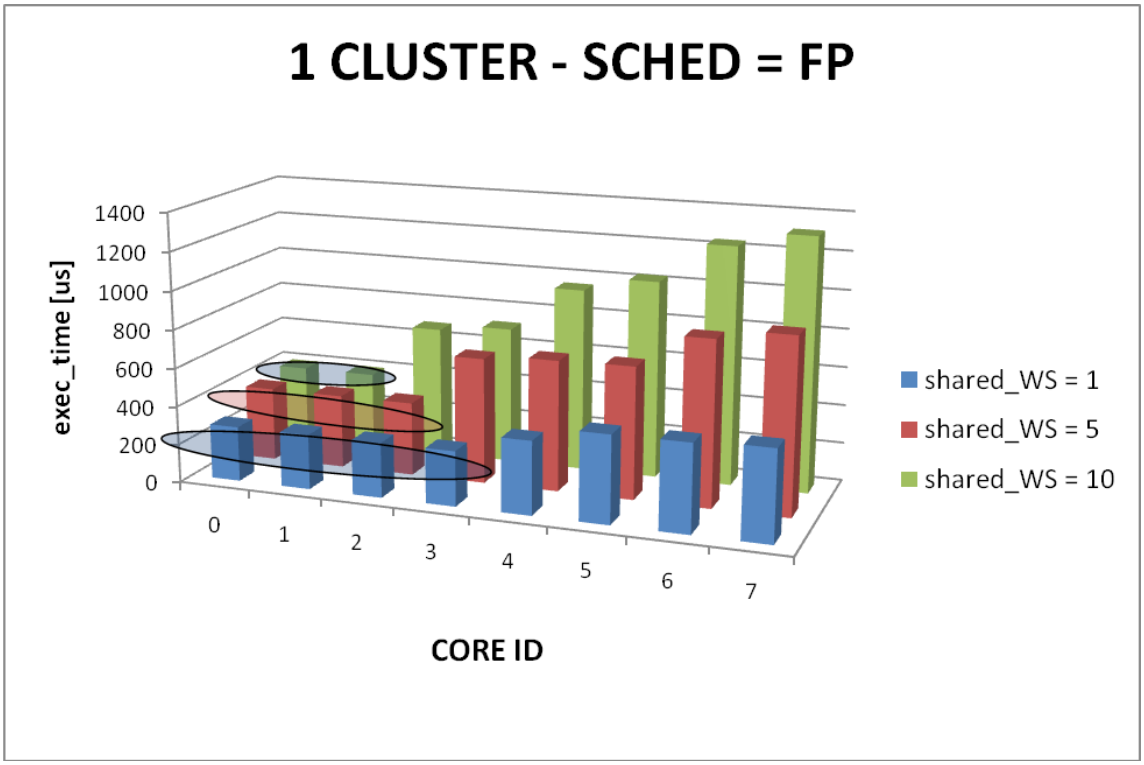


Figure 20: 1 Cluster - Fixed Priority

In Figure 20 the horizontal axis shows the ID of the 8 active processors, the vertical axis shows the time (absolute) to complete the calculation. On the third axis varies the access latency of shared memory in terms of number of cycles (shared_WS). The cores with higher priority (lowest ID), because of the scheduling FP, are privileged in access to shared memory, this trend is maintained even with increasing memory latency, however, a slower memory reduces the number of processors that can be served in succession.

Whereas in the case of a very fast shared 3D memory (shared_WS = 1), when a processor has accessed the memory, the computation right after the access is long enough to allow 3 more processors to be served, before the first one issues another request. As shown in Figure 18 when dealing with slower memories, the ratio between computation and communication is reduced and less processors can have comparable accesses (in terms of cycles) to the shared memory. It is easily pointed out that the crossbar, considering a fast shared memory, scales well up to 4 processors, when instead it has to do with a slower memory it becomes a bottleneck for the whole system.

With the same configuration we studied the behavior of the crossbar with other scheduling algorithms. Figure 21 and Figure 22 show the results for round robin and two levels priority.

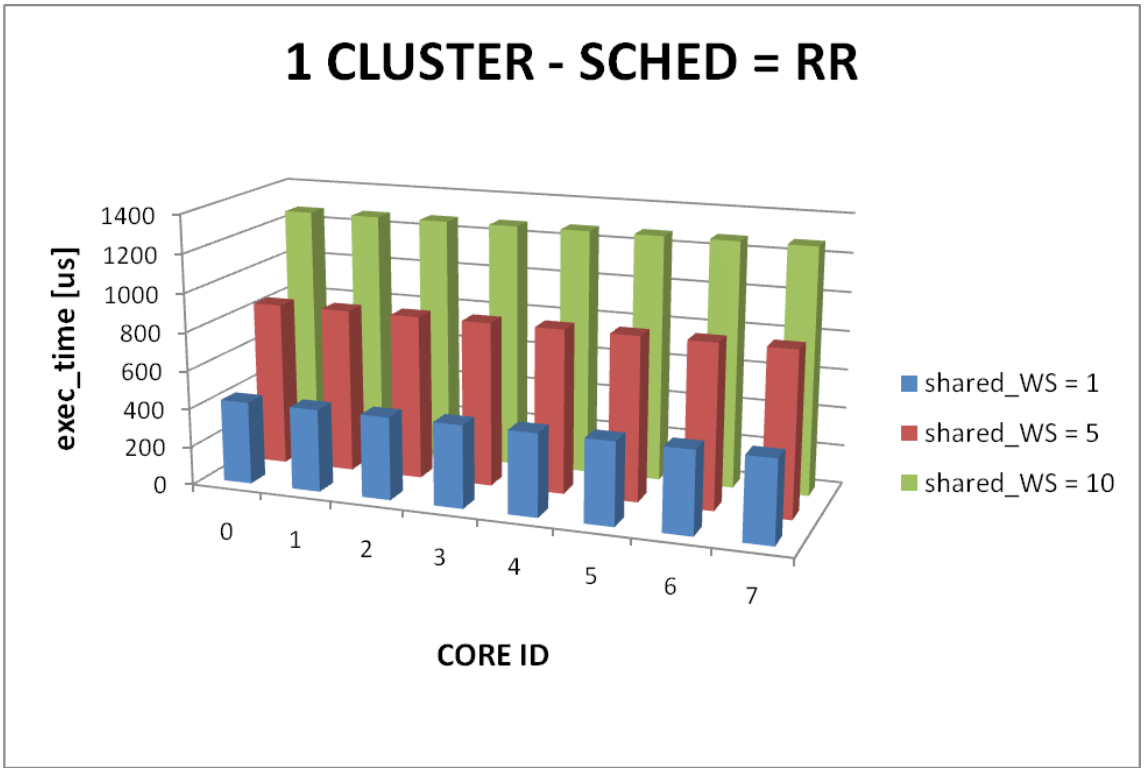


Figure 21: 1 Cluster – RR

As you can see the effect of round robin is to equalize the various execution times, the circular priority of this algorithm ensures that all processors are treated the same way when accessing the shared resources. A slower memory, in this case, affects equally every core increasing proportionally the execution times, without any particular impact. Comparing the simulation times shown, in the case of a round robin scheduling, execution time of the various processors is not the average of the execution times in the case of fixed priority. As a matter of fact, the round robin equalizes the maximum time spent waiting, distributing it evenly over the cores.

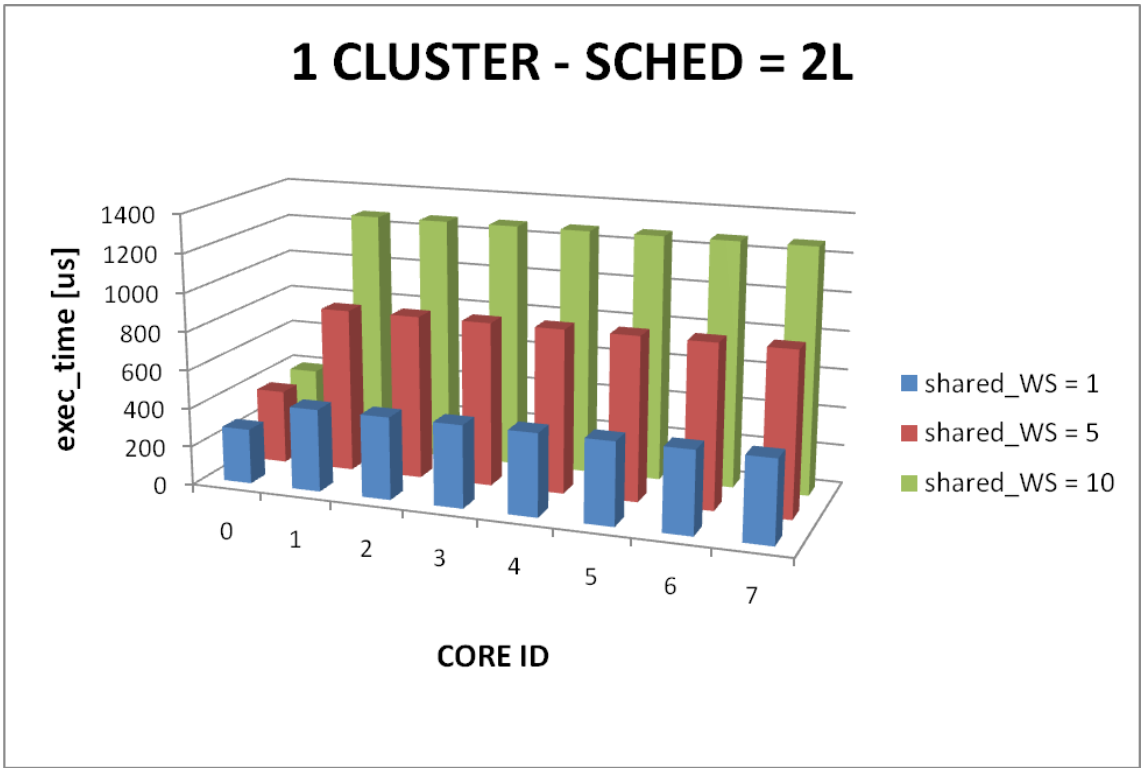


Figure 22: 1 Cluster – 2 levels scheduling

Observing Figure 22, we can see that two levels scheduling actually combines the other two algorithms, the core with lowest ID maintains the same advantages in terms of access privilege as in the fixed priority case, other cores have lower priority and identical to each other. As for round robin the shared memory latency will affect only the total run time.

The scheduling on two levels may have an intuitive application for systems organized in clusters, considering a multiprocessor node with a core operating as a local master and controls the other cores in the cluster, in a case such as this a two levels priority suits perfectly.

3.2.3 Crossbar and NoC Scaling – distributed contention

2 TILES

In order to study the behaviour of crossbar and tile together, we kept the same benchmark used in the previous test, but the 8 processors are distributed in two separate tile. Our configuration sets input matrices in private memory, the output matrix, instead, will be located in shared memory inside tile 0. In this case, the 4 processors of tile 0 access the shared memory locally, contending the access through the crossbar, the other 4 processors of tile 1 have contention on two levels, the port for the NoC (via the P2O bridge) and then access to shared memory inside tile 0. With this configuration, the contention becomes distributed, involving different components of the system (both crossbar and NoC).

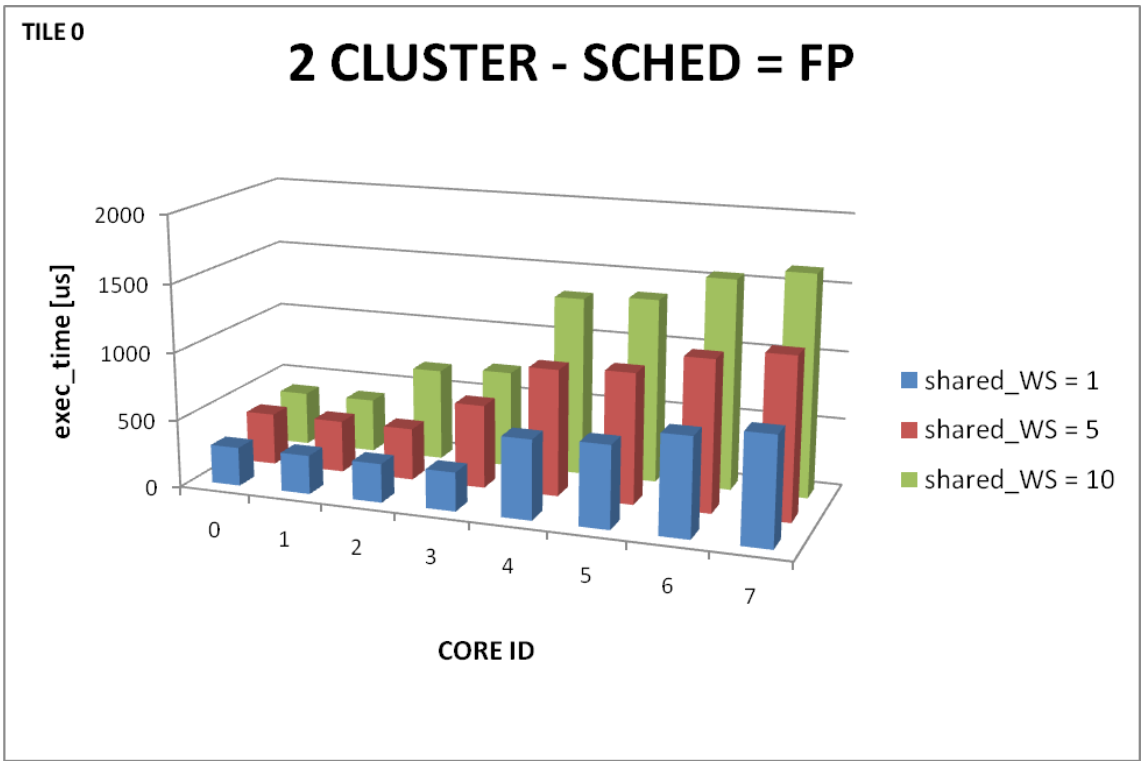


Figure 23: 2 Clusters – Fixed Priority

Looking at the chart in Figure 23, we can see the same behaviour already outlined in the previous analysis: for the 4 local processors (core 0, 1, 2, 3) as the latency of shared memory increases, the ability to serialize access is reduced. Looking at the execution times for cores 5, 6, 7, 8 (located inside tile 1) you can see how it impacts the latency of the NoC, in fact, their distribution in another tile results in a substantial increase in execution time compared to the previous case. This trend is maintained even with increasing memory latency.

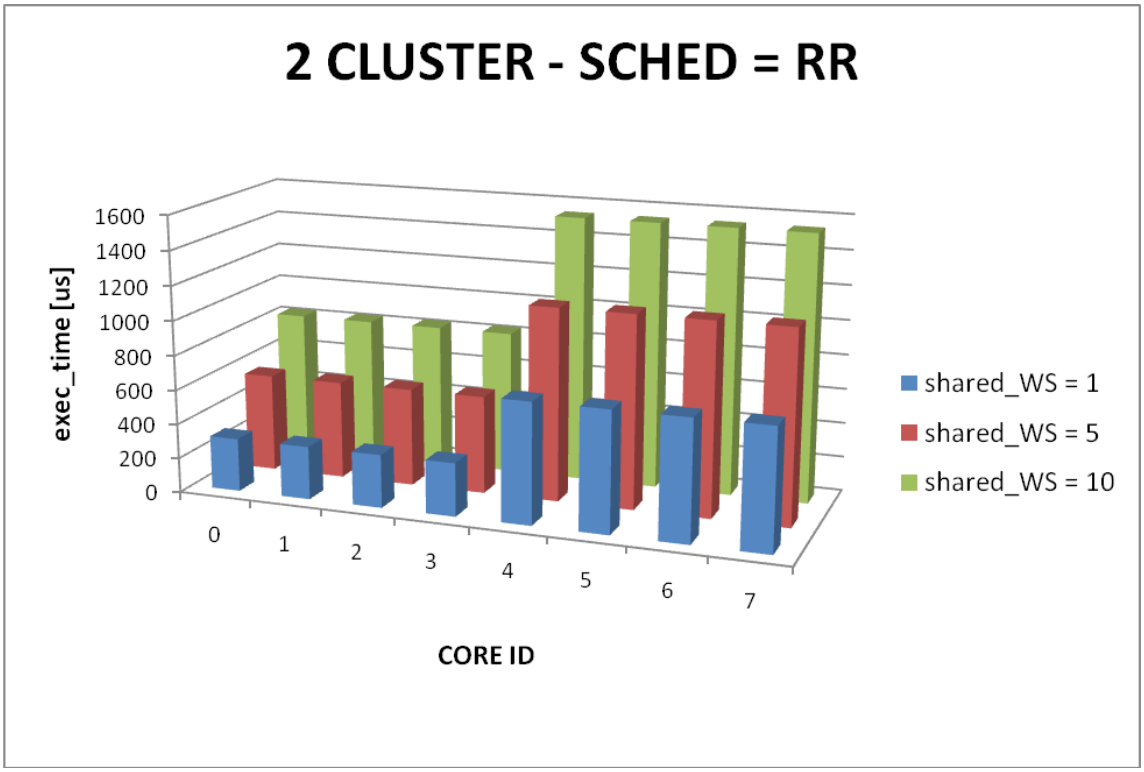


Figure 22: 2 Clusters – Round Robin

Considering round robin, as shown in Figure 22, it is clear the impact of contention for Network-on-Chip: the execution of processors inside tile 1 takes almost twice the time (+182% with WS = 10) than local processors.

4 TILES

In this subsection we discuss the results of an organization of 8 cores into 4 tiles, each tile has 2 processors. All processors have input matrices in their private memory; the output matrix is in the shared memory of tile 0 leading to NoC traffic. In this setup, the 2 processors inside tile 0 access the shared memory locally, contending for access through the crossbar, while the other processors must contend for access to both their NoC port (through the P2O bridge) and shared memory inside tile 0. As usual, the parameters involved are shared resources scheduling and shared memory latency.

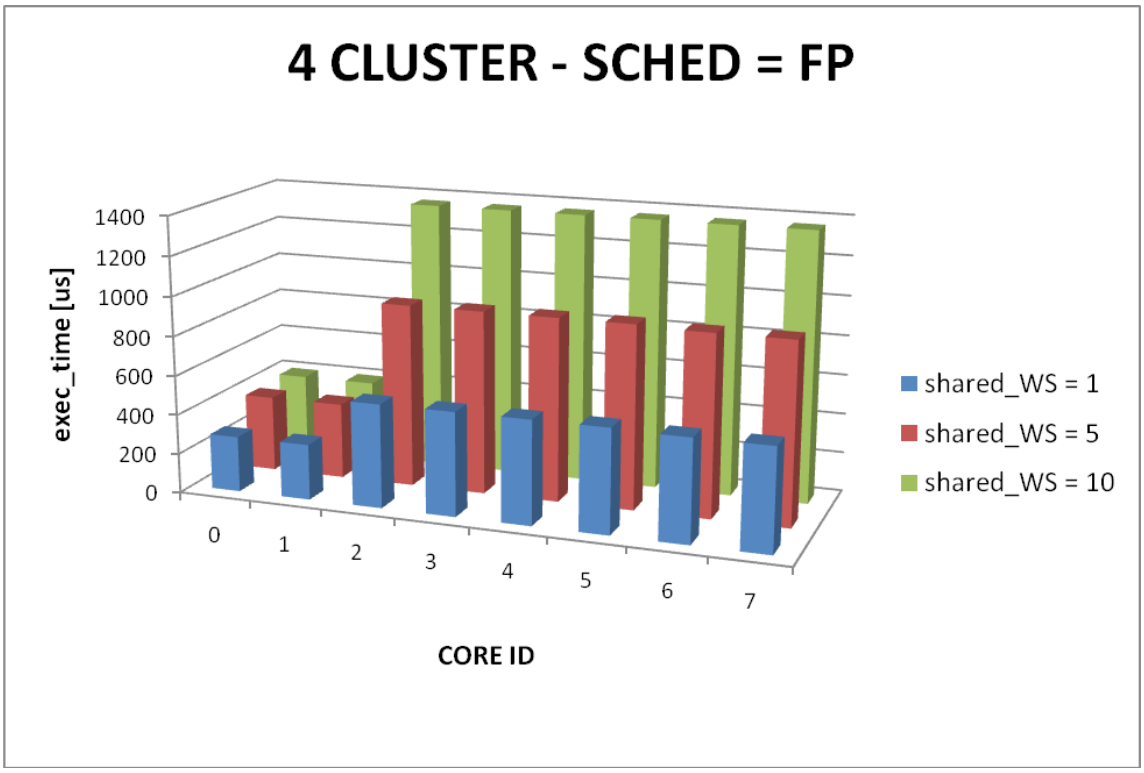


Figure 24: 4 Clusters – Fixed Priority

Looking at the chart in Figure 24, local cores are clearly favoured by this policy; regardless of the memory latency local processors are able to alternate each other between consecutive memory accesses computing their chunk in similar times. The cores in the other tiles are penalized the same way regardless of the memory latency. In this situation you can see how the Network on Chip becomes a bottleneck well before the memory.

Round robin shows (see Figure 25) the same peculiarities of the fixed priority case. Reducing the number of processors that are competing locally for a resource (memory or bridge) there can be accesses in sequence leading to almost identical execution times.

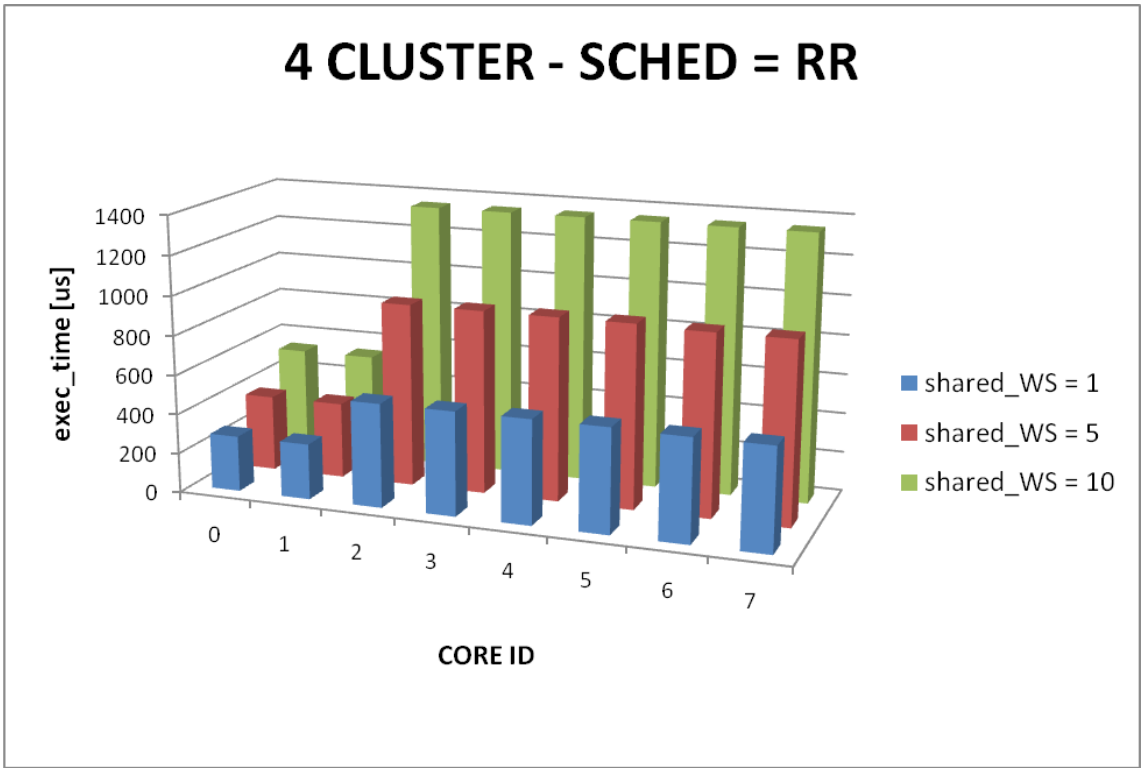


Figure 25: 4 Clusters – Round Robin

NoC Scaling

The final test was focused on the scaling properties of the NoC as the number of active cluster increase. To analyze this behaviour, every single tile has only one processor, in this situation none of the processors will contend access to the NoC (via bridge) and the only bottleneck is the target memory (inside tile 0).

The configuration of Figure 26 in contrast to previous execution times is normalized to the case of a single processor that takes care of the entire matrix multiplication. It is interesting to note that a fast memory can reduce the communication to computation ratio, multiple processors can be served in sequence and increasing to eight active clusters we still have advantages in parallelizing the workload.

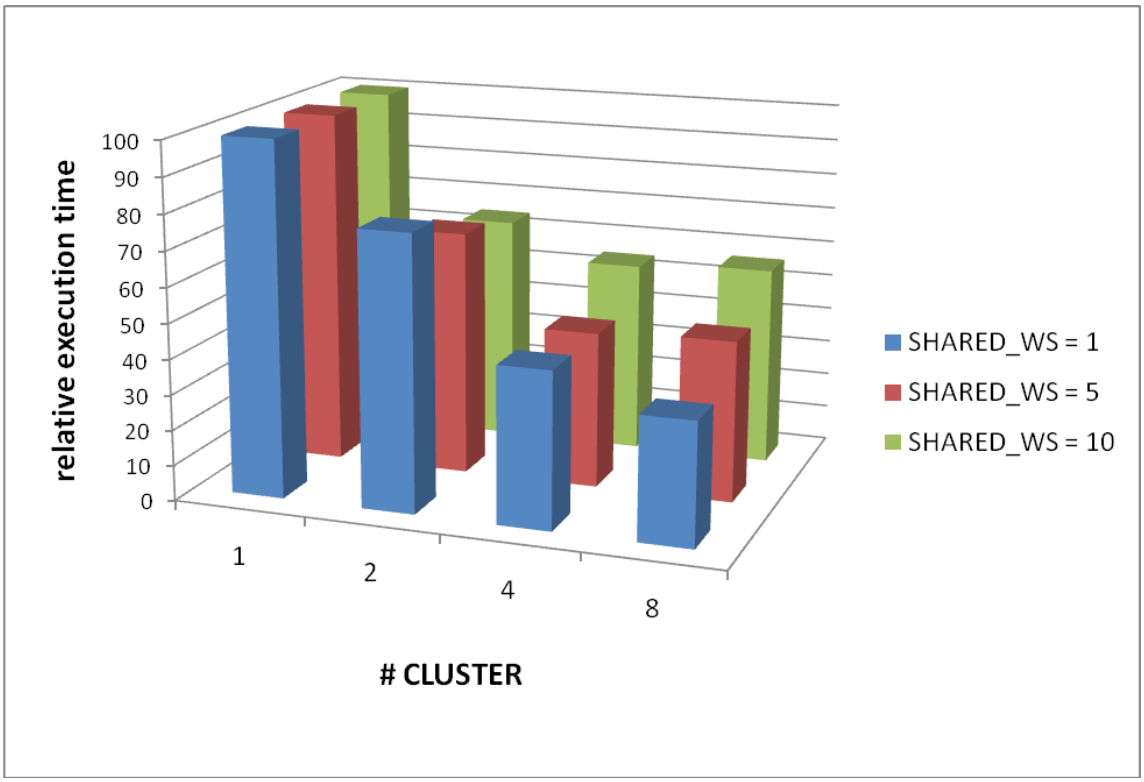


Figure 26: NoC Scaling

Another aspect to highlight is the inversion of the trend when changing from 2 to 4 clusters, in the case of a fast memory, varying from 1 to 2 clusters did not lead to too many benefits, in this case the slower memory benefits are greater because the higher latency of the memory hides the latency of the NoC.

However as the number of active tiles increase, the memory latency becomes a bottleneck, limiting scaling.

4 Lightweight NoC Model

In this section we briefly describe the new NoC model ported in MPARM, the main reason has been speeding up simulation time in order to do fast design space explorations.

4.1 Simulation Time on Host Machine

When carrying out the simulations shown in the previous section, we experienced very long execution times in our host machine (Intel © Core 2 Duo T5700 @ 2GHz), considering the purpose of our tasks, focused on offering a flexible virtual platform to do early architectural exploration, we decided to integrate a lightweight NoC model in order to speed up the simulations.

In order to quantify the simulation time, in Table 1 is reported the host simulation time when simulating a clustered 2x2 NoC architecture, each cluster composed of only one core taking care of 1 / 4 of the computation (refer to section 3.2.3 – 4 TILE case, but consider only 4 active cores instead of 8).

Table 1: Host simulation time

SIZE	**HOST TIME** [sec]
4	1.935
8	4.714
16	25.264
32	183.472

The size of the matrices involved in the matrix multiplication is used as traffic rate, as the size grows, more packets are sent through the network increasing network congestion. Host time grows in an exponential fashion, showing the limits of such an accurate model.

4.2 Noxim

Noxim is a Network-on-Chip Simulator developed at the University of Catania (Italy). The Noxim simulator is developed using SystemC and it can be downloaded [6] from SourceForge under GPL license terms. Noxim has a command line interface for defining several parameters of a NoC. In particular the user can customize the network size, buffer size, packet size distribution, routing algorithm, selection strategy, packet injection rate, traffic time distribution, traffic pattern, hot-spot traffic distribution. The simulator allows NoC evaluation in terms of throughput, delay and power consumption. This information is delivered to the user both in terms of average and per-communication results. In detail, the user is allowed to collect different evaluation metrics including the total number of received packets/flits, global

average throughput, max/min global delay, total energy consumption, per-communications delay/throughput/energy etc.

4.3 Noxim integration in MPARM

Noxim integration in MPARM needed the porting of the source code of the model (already SystemC) inside the MPARM environment and the enabling of the interconnection instantiation. The extra modules needed for this purpose were bridges operating a protocol translation from the PINOUT protocol to the "Noxim flit" protocol used by the routers.

4.4 Models Comparison

In this section we present the simulations run in order to compare the two NoC models available in MPARM Virtual Platform. We will refer to "*accurate model"* for the RTL, cycle-accurate model previously developed by UNIBO, and to "*lightweight model"* as the Noxim based model.

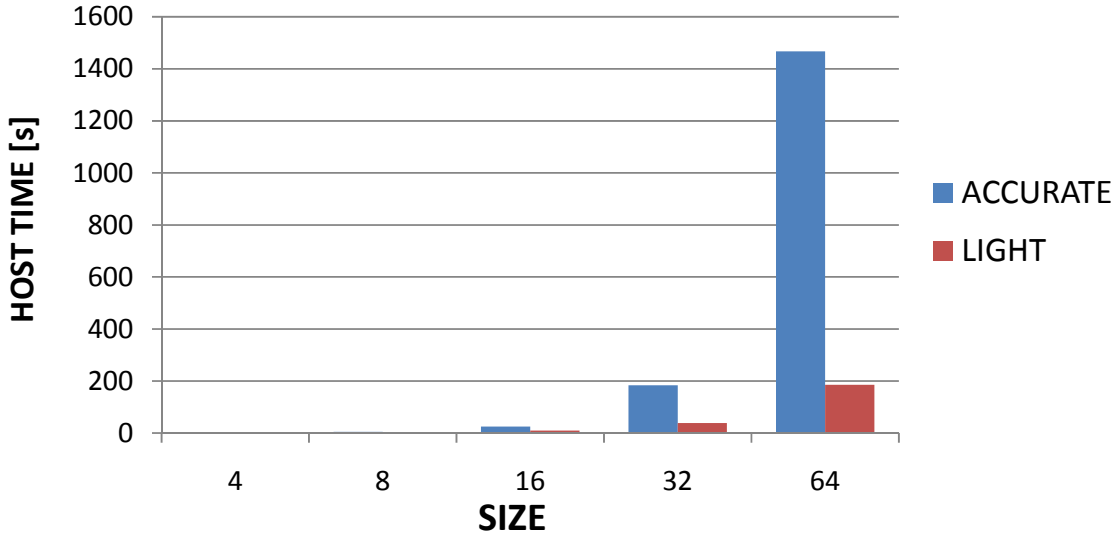


Figure 27: Host time comparison

In order to compare the two models we set up two identical architectures, a 2x2 clustered mesh with 1 core inside each cluster, differing only for the interconnection system, accurate model in one case, lightweight in the other. Every core within its tile takes care of 1 / 4 of the total matrix multiplication, where input matrices are located in private memories, the output one is stored in shared memory of tile 0, and the size of the matrices is an index of NoC traffic congestion. It is important to point out that even considering all matrices located in private memories, thus with any NoC activity, leads to different host simulation times. This is due to

the higher number of modules triggered by the system clock, loading SystemC simulation kernel, thus slowing down the overall host simulation time.

As you can see in Figure 27, the advantages of the lightweight model, in terms of host simulation time, are clear. Figure 28 gives an overview of the overall speedup, calculated as Tacc/Tlight, obtained by using the light model.

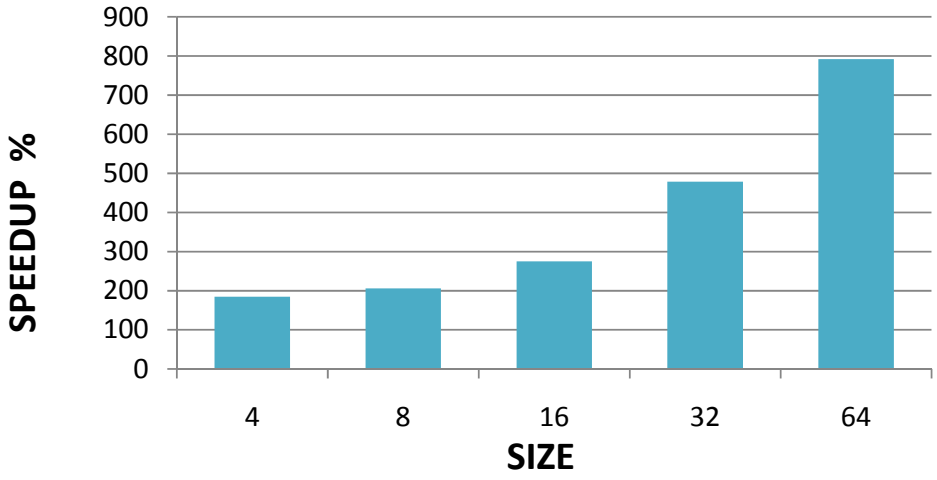


Figure 28: Speedup – light vs. accurate

A more high-level model of NoC, of course ensuring the functional correctness of transactions, leads to a loss of accuracy. This is shown in Figure 29, comparing the real simulation (not the host one) and plotting the relative difference.

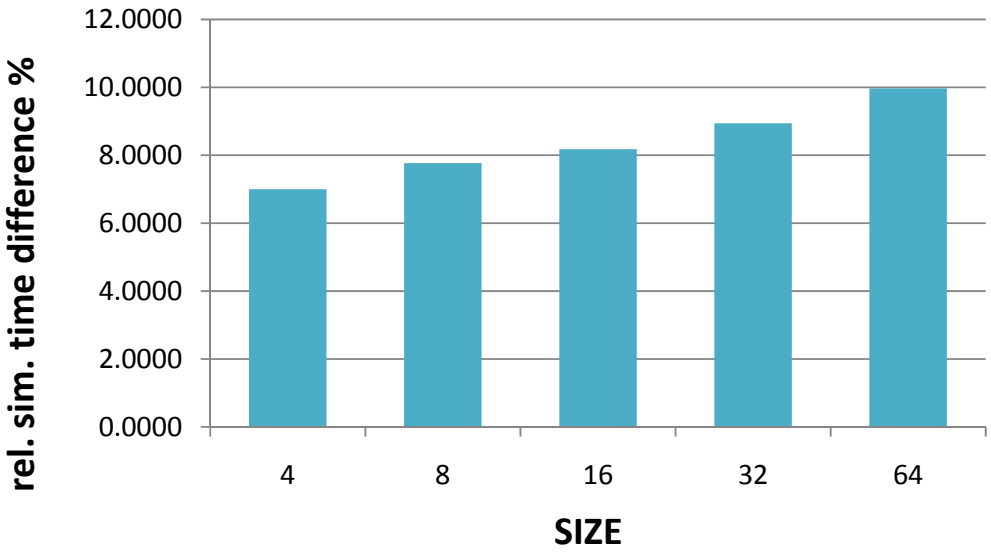


Figure 29: Relative simulation time difference - light vs. accurate

The relative difference increase as the simulated traffic increase, considering the SIZE=64 case it is clear that a 10% difference is highly acceptable when gaining a nearly 800% simulation speedup.

5 Conclusions

This deliverable discussed the software modelling of the components at the basis of on-chip communication in the PRO3D platform. These components (namely the crossbar, the computation tile, the Network-on-Chip) after having been validated, were made available as library elements in the simulation platform MPARM. Particular attention was paid to flexibility, allowing users to configure key parameters of these different components. The analysis and exploration activities demonstrate the high flexibility and accuracy of the tool developed.

6 References

[1] The MPARM virtual platform

http://www-micrel.deis.unibo.it/sitonew/research/mparm.html

[2] L. Benini and G. De Micheli, Networks on chip: a new SoC Paradigm - IEEE Computer, Vol. 35, No. 1, pp. 70-78, January 2002.

[3] OSCI - Open SystemC Initiative - http://www.systemc.org

[4] IEEE Standard - SystemC Language Reference Manual, Version 2.1

[5] M. Dales, SWARM 0.44 Documentation

http://www.cl.cam.ac.uk/users/mwd24/phd/swarm.html

[6] http://noxim.sourceforge.net/

[7] OCP Reference Manual v2.0 2003 – http://www.ocpip.org