UNIVERSITY OF ANTIOQUIA
Faculty of Engineering
Electronics Engineering Department

Noxim Simulator
Extensions to Support Task Mapping

Technical Report – 01 – 2016

PhD Student:
Luis Germán García Morales
german.garcia@udea.edu.co

Thesis Advisor:
José Edinson Aedo Cobo
jose.aedo@udea.edu.co

Research Group:
Embedded Systems and Computational Intelligence
SISTEMIC

Table of Contents

Abstract

The Wireless Network-on-Chip (WNoC) paradigm is one of the proposals aimed at overcoming the latency drawback found in conventional designs for multiprocessor systems with a large number of cores. However, the design of these platforms imposes several important challenges to the designer. At system level design, one of the challenges is the way the executable tasks are distributed onto the different cores. A poor distribution could lead to further increase the latency and the energy consumption. This requires the designer to establish a mechanism that efficiently allocates the executable tasks taking advantage of the characteristics offered by the WNoCs. In addition, the designer needs to establish a simulation tool with the purpose of evaluating the proposed mechanism in terms of a set of relevant metrics. In this technical report we introduce a set of extensions to a WNoC simulator called Noxim to ease the evaluation of different task mapping mechanisms.

1. Introduction

The Wireless Network-on-Chip (WNoC) paradigm is one of the proposals aimed at overcoming the latency drawback found in conventional designs for multiprocessor systems with a large number of cores. A WNoC platform has a set of processing elements (PEs), all of them connected through a two level interconnection network inside the chip or integrated circuit (IC). The first level corresponds to a wired network which uses metal connections between two adjacent nodes; the second level employs wireless channels for the communication between two specific pairs of nodes. Depending on the operating conditions of the system or the distance separating the communicating cores, data can travel through either or both levels inside the network. Thanks to the flexibility in the communication offered by WNoC systems, it is possible to improve significantly the throughput and scalability of the Multi-Processor System-on-Chip (MPSoC) platforms. However, the design of these platforms imposes several important challenges to the designer. At system level design, one of the challenges is the way the executable tasks are distributed onto the different cores. A poor distribution could lead to further increase the latency and the energy consumption. This requires the designer to establish a mechanism that efficiently allocates the executable tasks onto the cores of a WNoC-based system taking advantage of the characteristics offered by it. In addition, the designer needs to establish an appropriate simulation tool to evaluate the proposed mechanism in terms of a set of relevant metrics.

In computer architecture it is common to use simulation tools to evaluate different approaches and hardware designs before turning to the costly process of manufacturing ICs. In WNoC systems, when different task mapping strategies are being evaluated, the simulators can be employed to obtain different figures of merit such as latency, energy consumption, throughput, and many others, which allow to validate the performance of the mechanisms. Unfortunately, the simulators used in most of the reported works in the literature oriented to NoC/WNoC do not allow the mapping of tasks onto the cores of the platform. It means, the generation of payloads from one core to another in a synchronous fashion, according to the task graph of one particular application is not available. The real interest of these tools is to evaluate the performance of the different elements of the network such as buffers, routing algorithms, etc. Therefore, it is necessary to extend the functionality of an existing tool in order to take advantage of the WNoC communication infrastructure and allow the mapping of tasks.

In this report, we show the extensions we have performed to a well-known simulation tool to support the mapping of tasks and allow the evaluation of different mechanisms. The rest of this technical report is organized as follows: section 2 shows the most relevant simulation tools and introduces Noxim as a WNoC-based platform simulator. Then, section 3 presents a set of considerations on task mapping and describes the required changes to Noxim. Next, section 4 shows all the extensions performed to Noxim according to the considerations presented in the previous section. Finally, section 5 concludes this report with the conclusions and final remarks.

2. Simulation Tools

Most of the works reported in the literature oriented to NoC / WNoC-based systems make use of known simulators. Although most of them do not support WNoC systems in principle, it is possible to integrate models that allow them the evaluation of different characteristics of those systems, especially latency, traffic congestion and energy consumption. The majority of these simulators are written in C/C++ language, are available and open source, and offer several characteristics to evaluate. Table 1 shows a summary of the most important simulators for NoC/WNoC. We have chosen Noxim simulator (Catania, Mineo, Monteleone, Palesi, & Patti, 2015) basically because it currently offers support to WNoC and has been reported in several works in the literature. Noxim was developed using System C and provides a command line interface for defining several parameters of the NoC / WNoC such as network size, buffer size, packet size distribution, routing algorithm, selection strategy, packet injection rate, traffic time distribution, number of wireless channels and many others. The simulator allows the evaluation of NoCs in terms of throughput, delay, and power consumption.

Simulator	Framework	Open Source	WNoC Support	Last Update
Noxim	SystemC	Yes	Yes	2016
BookSim	C++	Yes	No	2015
Garnet	C++, Gem5	Yes	No	2015
TOPAZ	C++, Gem5	Yes	No	2014
HNOCS	OMNet++	Yes	No	2013

Table 1. Summary of NoC simulators reported in the literature

All the basic blocks in this simulator have been modelled in VHDL and synthesized using Synopsys Design Compiler considering a 65 nm CMOS standard cell library from United Microelectronics Corporation (UMC – www.umc.com). The WNoC energy contributions were taken from the works presented in (Yu, Baylon, et al., 2014; Yu, Sah, et al., 2014).

Noxim is basically a network simulator. This tool offers the designer the possibility of seeing the performance of the different elements in the network: buffers, distribution of packets, routing algorithms, selection strategy, etc. For the simulation, the user can choose the type of traffic to be used along with the packet injection rate. After the simulation is executed, the designer obtains latency, energy and throughput statistics, all of them related to the interconnection infrastructure. Unfortunately, the mapping of tasks onto the cores of the platform is not entirely possible in Noxim as it will be explained in the next section.

3. Considerations on Task Mapping and Required Changes to Noxim

At system level design, we require to establish a task mapping mechanism to perform the arrangement of tasks onto the WNoC resources taking advantage of its characteristics. In order

to validate its functionality, we need to evaluate the performance of several tasks running onto the WNoC resources in terms of latency, energy and execution time. We are not interested in improving the computational capabilities of the PEs, but reducing the latency, and hence the execution time of the applications, by improving the way the tasks are mapped onto the PEs. Let us illustrate this with an example. We use the task graph shown in Figure 1. We consider a homogeneous 4x4 Mesh-WNoC platform where the PEs are identical and the wireless channels are disabled for the sake of the example. The execution time for each task is indicated in the vertices of the figure. We assume an ideal scenario where context switching, memory access and other aspects take zero cycles. We use a mapping mechanism that determines three task arrangements. For the first arrangement, all the tasks are mapped to the same PE, therefore the communication time between tasks is 0, as shown on the left side of the figure. In this case the execution time of the entire application is 50ms + 40ms + 30ms + 30ms = 150ms. For the second arrangement, the tasks are naively mapped to different PEs. The time to transmit the data from each task to another corresponds to the values shown in the arcs on the middle of Figure 1. The execution time of the entire application is 50ms + (10ms + 30ms + 16ms) + 30ms = 136ms. Observe that tasks 2 and 3 can run almost at the same time. Finally, for the third arrangement, the tasks are cleverly mapped this time, so the latency between tasks is lower as shown on the right side of Figure 1. The execution time of the entire application is 50ms + (4ms + 40ms + 4ms) + 30ms = 128ms. As can be seen, the execution time for the third arrangement is better compared to the ones obtained before.

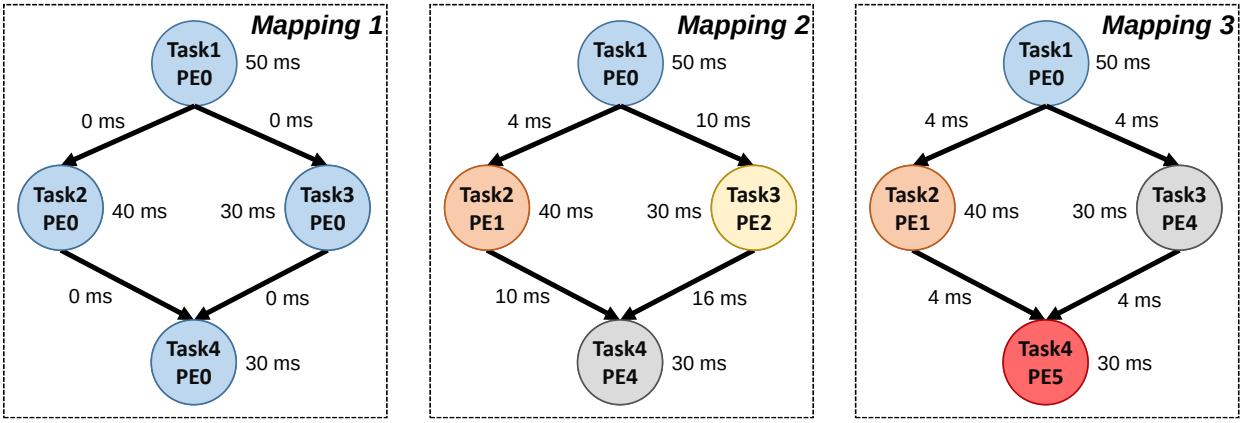

Figure 1. Task graph of a particular application with three possible mappings

According to the previous example, we require a simulation tool that allows the mapping of tasks and collects the latency, execution time and energy consumption statistics according to the application execution. Unfortunately, Noxim does not support the mapping of tasks. Even though there is a traffic option in Noxim called based traffics via input traffic text files, which might allow the simulation of applications running onto the PEs, it does not allow the synchronization between PEs: the generation of payloads from one core to another in a synchronous fashion, according to the task graph of the application is not available. It means, the data arriving to a particular PE will not trigger the execution of the task (or tasks) mapped to that resource, and

hence it will not be possible to account the required metrics accurately.

In order to support the mapping of tasks and allow the collection of the required metrics, the following aspects must be established in Noxim:

1. A semantic domain for the representation of the applications in the simulator.
2. A mechanism that allows the synchronization between PEs: (1) each PE must be able to send payloads to any other PE, including itself. (2) Each PE must be triggered by payloads coming from any other PE.
3. A mechanism to perform the scheduling of tasks running on the same PE.
4. An energy model to estimate the energy consumed by each PE.
5. A mechanism that collects the latency of the payloads transmitted per PE, the execution time of each application and the energy consumed by each PE.

Next section explains the implementation of the aforementioned aspects into Noxim.

4. Extensions Performed to Noxim

In this section we present the extensions carried out to Noxim to support the mapping of tasks according to the aspects mentioned in the previous section. First of all, the semantic domain, which is the way the tasks are introduced to the simulator, is presented. Then, we introduce the new features the simulator offers after performing the extensions. We focus next on the parts of the simulator that were changed or added, and explain the task mapping simulation mechanism introduced. Finally, we show how the user can introduce the applications along with their task mappings onto the WNoC resources for the simulation.

4.1. Semantic Domain

The applications can be described by using a proper and formal representation that combines high-level details such as the specification of the tasks and their dependencies, along with the platform elements. The combination of such a diverse elements in a single representation is normally referred as a common semantic domain, through which is possible to formalize and automate the process of mapping tasks to the elements of the platform with the purpose of yielding an implementation (Sangiovanni-Vincentelli, 2007). Normally, the semantic domain takes the form of a graph with annotations, which is called a task graph. Figure 2 depicts an example of an annotated task graph (ATG). In that figure the nodes or vertices represent a task of the application, the edges represent the dependencies between tasks, it means the connection between master and slave tasks (Maqsood, Ali, Malik, & Madani, 2015), and the annotations are used to display requirements or constraints about the platform or the tasks themselves. An example of a master / slave pair corresponds to the tasks t_1 and t_2 with edge $C_{1,2}$ in the aforementioned figure. In this pair, the slave task t_2 starts the execution after it receives the data coming from its master t_1. The annotations, on the other hand, can be divided in several levels

according to the resources, (e.g. computation and communication).

An annotated task graph is denoted as $ATG = (T, E)$, where T is the set of tasks of the application and E represents the dependence between connected tasks. We have divided the annotations into two levels: task and communication. The task-level annotations for task $t_i \in T$ are $(t\_id_i, t\_pblocks_i)$ where t_id_i represents the task identification and $t\_pblocks_i$ indicates a set of processing blocks (PBs). A PB indicates the number of cycles to be executed by the task before it sends the payload to its subsequent task. Since a task can have several subsequent tasks, there must also exist several PBs which are sequentially executed one at a time. The annotations of $t\_pblock_{i,k} \in t\_pblocks_i$ are $(pb\_id_{i,k}, pb\_exec_{i,k})$ where $pb\_id_{i,k}$ represents the processing block identification and $pb\_exec_{i,k}$ indicates the number of cycles to be executed. Regarding the communication-level annotations, the edge $e_{i,j} \in E$ in $ATG = (T, E)$ has two annotations $(master\_pb\_id_{i,j}, e\_size_{i,j})$. $master\_pb\_id_{i,j}$ indicates the processing block of master task t_i that generates the payload, and $e\_size_{i,j}$ indicates the size of data to be transferred from task t_i to t_j when the associated PB finishes its execution.

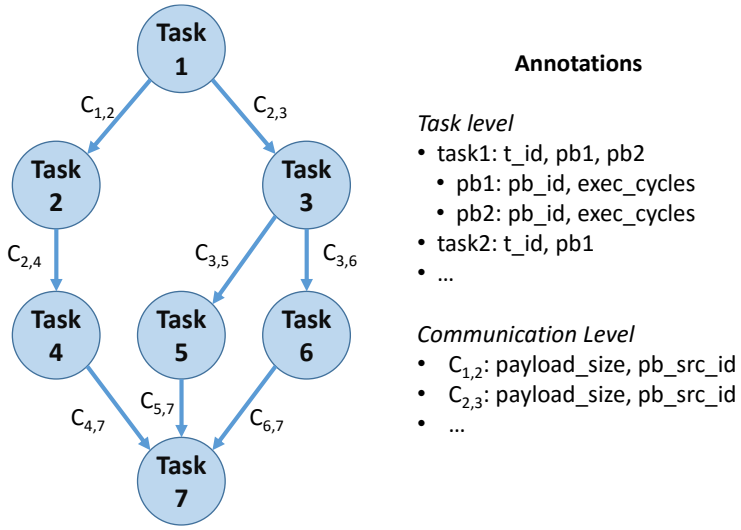

Figure 2. Task graph with annotations

4.2. Task Mapping Features on Noxim

In order to take advantage of the Wireless communication infrastructure already introduced to Noxim and allow the mapping of tasks, we carried out a set of extensions which gives the simulator the following features:

1. Any task from a particular ATG (as shown in Figure 1) can be mapped to any PE.
2. The execution of each task can be emulated by spending the specified time provided in the ATG. The user can define several PBs where each of them spends the specified time and then transmits data to the associated subsequent task. There can be as many PBs as subsequent tasks.
3. The PEs can synchronize among them according to the ATG:
   a. Each task is able to send any payload to the subsequent tasks as indicated in the task

graph. The exchange of data between PEs is carried out through the two level communication infrastructure (wired and wireless) already available in the simulator.

b. Data coming to the destination task (destination PE) triggers the execution of its first PB. If a task has more than one precedent task, the first PB is executed only when the data coming from all the precedent tasks have arrived.

4. Each PE of the simulator allows the mapping of multiple tasks. A simple Round Robin scheduler is employed to this end.
5. Multiple task graphs (multiple applications) can be mapped to the different resources of the platform.
6. Latency, energy and execution time statistics for each application can be collected along the simulation.

4.3. Introducing the Extensions to Noxim

The WNoC-based MultiProcessor System-on-Chip systems are composed of nodes which are connected through a communication architecture based on routers and network interfaces (Li, 2012). A WNoC has available conventional wired links, along with a set of high-speed single-hop wireless links, where the information can be transmitted through the wired links, the wireless links or both (Ganguly, Pande, Belzer, & Nojeh, 2011; Wang, Hu, & Bagherzadeh, 2011). In this scheme of two levels, the information is routed by using either of the links, depending on the operating and traffic conditions, the available bandwidth and the distance the message must travel before reaching its final destination. Noxim simulator supports this two level scheme as shown in Figure 3. This figure shows the global description of the simulator using SystemC language. Each *Tile* block contains basically two sub-modules, *Router* and *Processing Element* as seen in Figure 4. We have carried out the extensions to the *Processing Element* sub-module, since PEs must be able to synchronize among them according to the ATG. The *Processing Element* sub-module basically has two processes, *tx* and *rx*, as shown in Figure 5, which are in charge of transmitting/receiving packets to/from the *Router* in the *Tile* module. Moreover, there are a set of functions that are employed by those processes. We extended the functionality of the processes *tx* and *rx* by adding several functions that allows the synchronization among communicating PEs. In addition, we created a new process called *pe* which is in charge of performing the execution of the PBs of the tasks mapped to that particular PE. *The extended processing element* sub-module can be seen in Figure 6. At every cycle, the process *rx* checks if there is data coming from any other PE through the WNoC communication infrastructure and sends it to the corresponding mapped task on the PE. Tasks waiting for data are in *idle* state as will be explained in next section. When a particular task has received the expected data from all its precedent tasks, the process *pe awakes it and* starts its execution, running the PBs one by one according to the ATG. Any time a PB finishes its execution, the *pe* process puts the payload into a transmission queue. When all the PBs have been executed, the task might go to *idle* state. On the other hand, at every cycle, the *tx* process checks if there is data available in the transmission

queue to be transmitted to a different PE. If so, the *tx* process sends the data in the mentioned queue to the WNoC communication infrastructure.

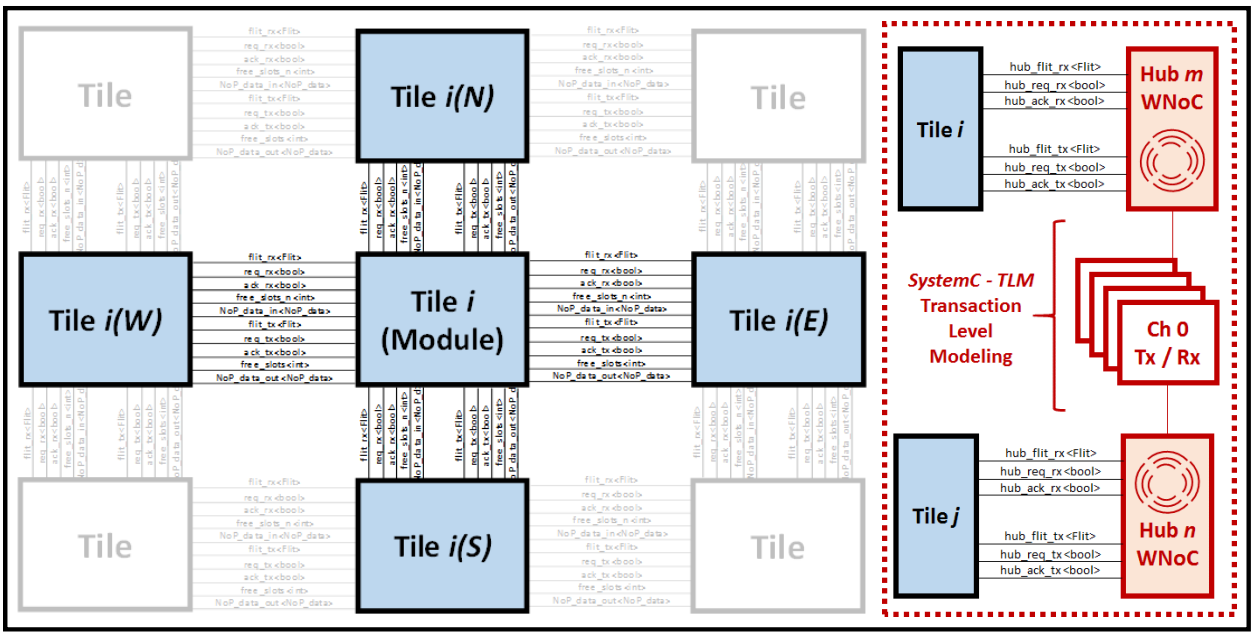

Figure 3. Noxim simulator as described in System C

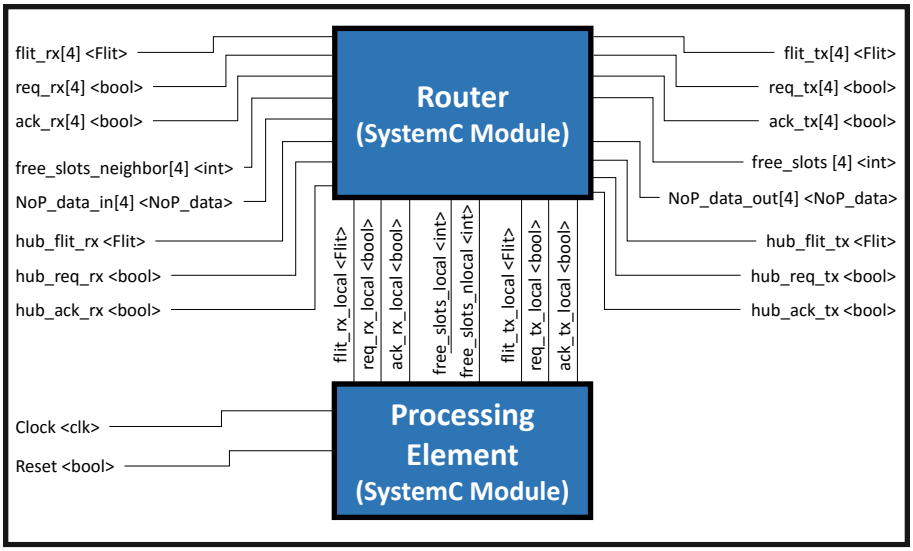

Figure 4. Tile Element Module

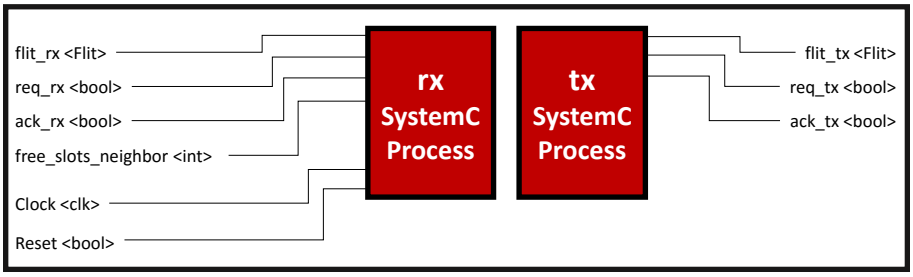

Figure 5. Processing Element Sub-Module

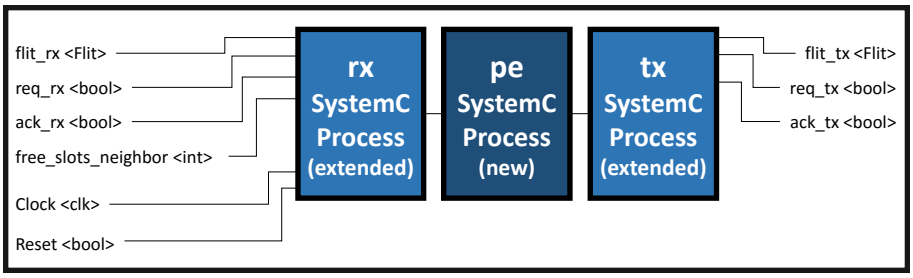

Figure 6. Extended Processing Element Sub-Module

The extended *rx* and *tx* processes and the new process called *pe* in the sub-module *Processing Element* are part of the task mapping simulation mechanism that was added to the simulator, which includes a simple scheduler for the execution of the different tasks. Next section explains the functionality of this mechanism.

4.4. Task Mapping Simulation Mechanism

The task mapping simulation mechanism allows the mapping and execution of several applications. It is possible to specify up to l applications, where each of them must have a proper annotated task graph as described in section 4.1. Each application can have up to q mappings without overlapping. The task graph of each application can have up to n tasks. A task can be mapped to one of the m PEs available in the WNoC platform. All tasks might be mapped to just one PE if necessary. Two types of tasks are considered: root and non-root tasks. The root task is the beginning of the application as shown on the left side of Figure 7. It does not have precedent tasks and must have, at least, one subsequent task. It is possible to define the time between consecutive executions, if required. It means, once the root task finishes its execution, it will be restarted after the specified time (e.g. streaming applications). There can only be one root task in the task graph. On the other hand, the non-root tasks correspond to the rest of the tasks in the ATG. They must have both precedent and subsequent tasks as shown on the middle of Figure 7. There is a particular task where the execution flow of the application converges to it and it is called the leaf task as shown on the right side of Figure 7. This task cannot have subsequent ones and it is used to estimate the execution time of the application.

Each task can have from 1 to p processing blocks. Each PB has a subsequent task associated. When a PB completes its execution, the assigned payload is sent to the subsequent task. Following the parameters that can be specified to each PB:

1. Execution time: in cycles.
2. Subsequent task: task id (required by all the tasks except the leaf one).
3. Payload: the size of the payload to be transmitted to the subsequent task. Normally is given in FLITs (each packet in NoC is broken into small pieces called FLITs)

Figure 8 shows the processing blocks of a particular task. When *task 1* starts its execution, the *pblock0* is executed first for *50 cycles*. Once that PB finishes, the *10-FLIT payload* is sent to *task 2*. After that, *pblock1* starts its execution for 20 cycles and then the *5-FLIT* payload is sent to

task 3. Finally, *pblock2* starts its execution for *15 cycles* and then the *15-FLIT* payload is sent to *task 4*. The moment in which a block is executed depends on various factors that are explained below.

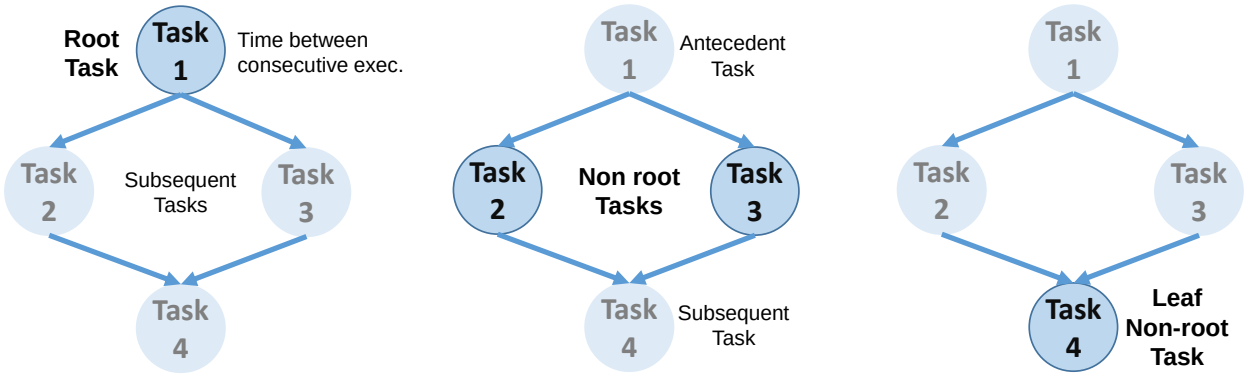

Figure 7. Root task and non-root tasks in the annotated task graph

Figure 8. Processing blocks for a particular task

The mechanism that decides when a task can be assigned to the corresponding PE according to the mapping established is a simple Round Robin scheduler. This allows the execution of multiple tasks on the same PE. A task could be in one of the states shown in Figure 9 which is explained below:

1. Given a particular mapping, a task is put in one of two states according to its type when it is mapped to a PE: sleeping state (root task) or waiting rx state (non-root task).
2. When the start time of one particular root task is reached, the task is put in ready / running state. Non-root tasks, instead, are put in ready / running state only when they have received all the data coming from their precedent tasks.
3. The Round Robin scheduler decides when a particular task must be assigned to a specific PE according to the mapping. Certainly, several context switching might occur during the execution of any particular task.
4. Each task begins its execution with its first PB and runs until the execution time of the last PB expires. After the execution of one particular PB the task sends data to other tasks according to the ATG (except for leaf tasks). When all the PBs have been executed, one of the two following things can happen to the task depending on its type:
   a) Root task: it will be put in sleeping state. The task will be executed again when the time between consecutive executions expires.
   b) Non-root task: it will be put in waiting rx state. The task will be executed again when

all the precedent tasks have sent their data to this task.

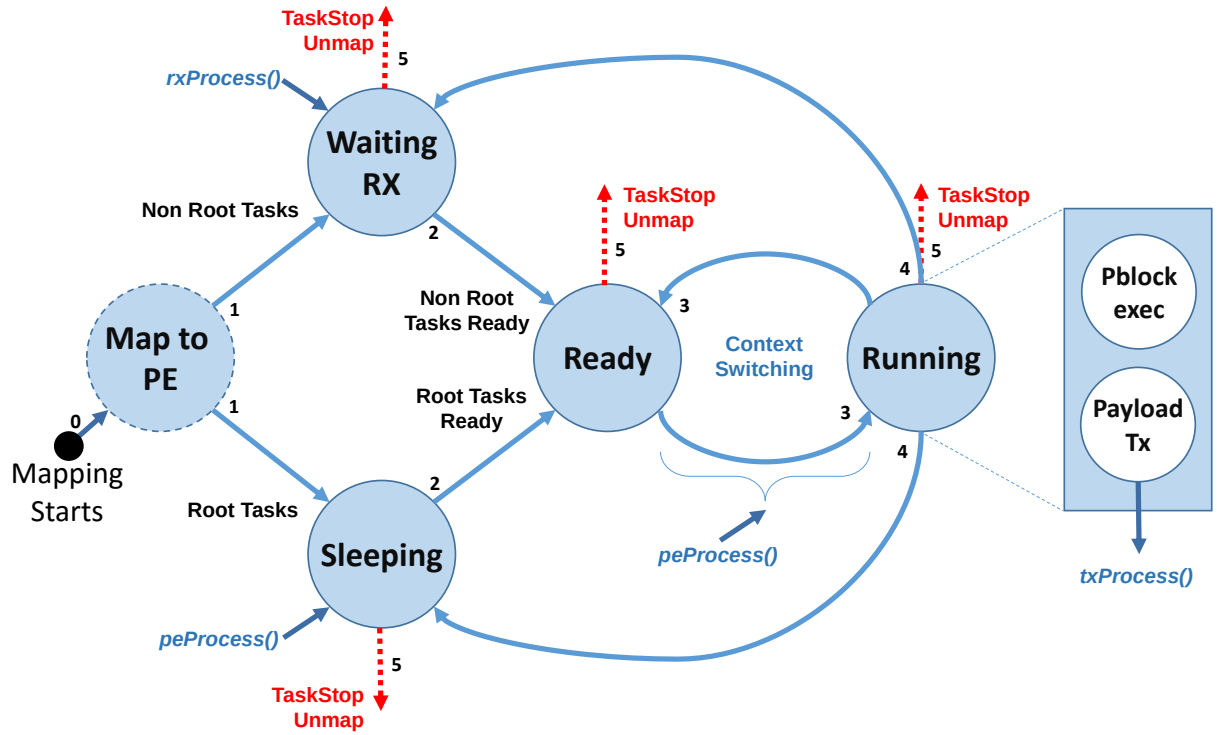

Figure 9. Task Execution States in Noxim

4.5. Collection of Metrics

The fundamental metrics that have been considered in these extensions are: (a) end-to-end latency, (b) energy consumption, and (c) execution time. The latency is the main metric for the task mapping mechanisms we want to evaluate through the simulator; however, the rest of the metrics are also important, because it is necessary to avoid trivial solutions, like for instance, mapping all the tasks to the same core. When the tasks are mapped to the same resource, two things might happen. First, the execution time might increase, and second, the energy consumption can be so high at that core compared to the others (*power sunspots*). These two metrics must tell the mechanism to balance the workload at the different cores, whilst the first metric (latency) allows the approach to efficiently use the communication resources.

The latency is computed by the *tx* and *rx* processes from node to node. At the end of the simulation, the latency between root and leaf task is also computed. It is given in cycles. The execution time is computed from root to leaf task based on the critical branch of the ATG. Finally the energy consumption is the sum of the energy consumed by the WNoC communication infrastructure and by the PEs. The energy consumed by the WNoC is computed by Noxim according to the models proposed in (Yu, Baylon, et al., 2014; Yu, Sah, et al., 2014). Regarding the PEs, the energy consumed is computed according to the work presented in (Atitallah, Niar, Greiner, Meftali, & Dekeyser, 2006). They proposed a simplified power consumption as shown in Equation 1, where n_j^{run} is the cycles during which the PE_j is in running mode, n_j^{idle} is the cycles

during which the PE_j is in idle mode, and finally E_{run} and E_{idle} are the average energy consumed per instruction per cycle by any PE in running and idle modes respectively. This model is meant for processors with relatively simple architectures. The values E_{run} and E_{idle} can be indicated in the *power.yaml* file in Noxim.

$$E_{PE_j} = \left(n_j^{run} * E_{run}\right) + \left(n_j^{idle} * E_{idle}\right)$$

Equation 1. Energy consumption by PE

4.6. Simulation Configuration

For the representation of the applications ATGs are used as mentioned before. The description of the applications can be introduced to the simulator through a regular text file. This file includes an ATG for each application, their corresponding mappings and the simulation configuration. The ATG of each application is the first thing to be indicated in the simulation file as shown below:

app: app_id, root_min_restart_cycles

task: task_id, exec_time_pb0, sub_taskid_pb0, payload_pb0, exec_time_pb1, ...

The first line allows to specify the application id and the time between consecutive executions for the root task, if required. If zero cycles is set, the first PB of the root task is executed again as soon as the last PB finishes its execution. The second line allows to specify a particular task of the application. This line can be repeated as needed according to the number of tasks, having in mind that the first task corresponds to the root and the last one corresponds to the leaf. The first parameter is the task id followed by the definition of the different processing blocks. Each PB defines the execution time, the subsequent task id and the payload.

Once the ATG is established, the mapping for the application must be defined as explained below:

map: mapping_time, unmapping_time, task_id, pe, task_id, pe , task_id, pe, ...

The line indicated before allows to establish the time when the tasks of the application must be mapped and unmapped, followed by the assignation of the different tasks to the PEs of the platform as needed. Keep in mind that the root task starts its execution as soon as the mapping time is reached. If the user wants to delay the starting of the root task to have a more realistic scenario, a simulation option must be given. It will be explained later. There can be multiple mappings without overlapping. It means, a particular mapping time must be greater than the previous unmapping times. This feature can be useful when performing task migration.

When the ATG for a specific application has been defined along with their mappings, it is possible to add a new application to the simulation. The user just need to repeat the aforementioned steps with the new application. Then, the user must specify a couple of options for the simulation process as explained below:

sim: scheduler_tick, scheduler_delay, map_before_starting_root_task, mappings_gap

1. scheduler_tick: specifies the number of cycles a task can use the PE before being evicted.
2. scheduler_delay: specifies the number of cycles when performing a context switching

(emulation).

3. map_before_starting_root_task: number of cycles to map any task before the mapping time. This allows the execution of the root task to be delayed some cycles. In any case the root task will start at the mapping time specified.
4. mappings_gap: minimum cycles allowed between consecutive mappings of the same application (for task migration purposes).

An example of a task mapping file that can be passed to the simulator is shown next. There are two applications with their respective ATGs as shown in Figure 10. It is possible to see also the mappings and the simulation configuration. Each application has two mappings. In the first mapping, all the tasks are mapped to the same PE whilst in the second mapping all the tasks are mapped to different PEs. The platform is a 4x4 mesh WNoC with the Wireless channels disabled for the sake of the experiment.

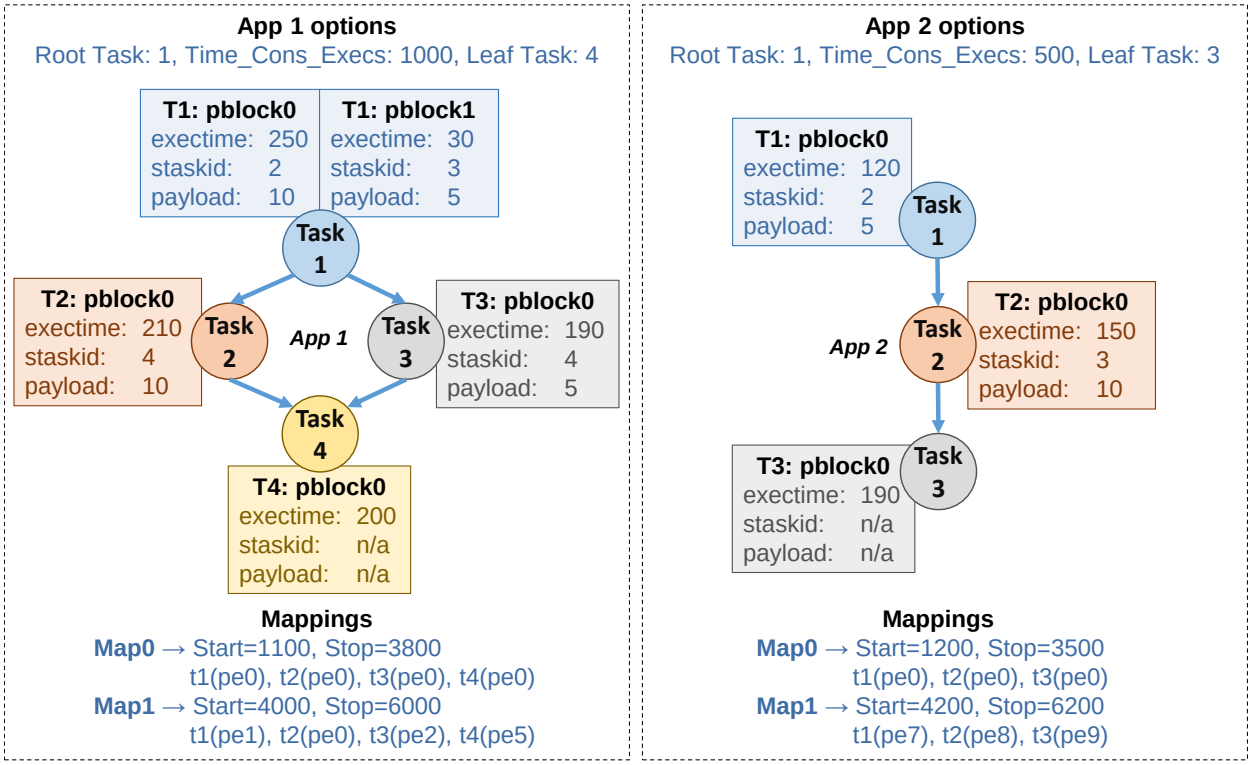

Figure 10. Two applications to be simulated using Noxim

A simulation file that describes the ATGs presented in the figure is shown below:

```
app: 0, 1000
task: 1, 250, 2, 10, 30, 3, 5
task: 2, 210, 4, 10
task: 3, 190, 4, 5
task: 4, 200, -1, 0
map: 1100, 3800, 1, 0, 2, 0, 3, 0, 4, 0
map: 4000, 6000, 1, 1, 2, 0, 3, 2, 4, 5
app: 1, 500
```

task: 1, 120, 2, 5
task: 2, 150, 3, 10
task: 3, 190, -1, 0
map: 1200, 3500, 1, 0, 2, 0, 3, 0
map: 4200, 6200, 1, 7, 2, 8, 3, 9
sim: 100, 10, 10, 12

The simulation file can be introduced to the simulator through the command-line as shown below:

noxim [noxim_options] –taskmapping mapping_file –taskmappinglog result_file

- *noxim_options*: configuration to be adopted by Noxim.
- *mapping_file*: file that contains the description of the applications, their mappings and task mapping simulation options.
- *result_file*: the validation of the task mapping simulation, execution report and the collected matrics are written to this file.

Below it is a summary of the information written to the file *result_file*.

```
Task Mapping Simulation File created on: Fri Aug 12 16:16:26 2016

************************
TaskMapping File Reading
************************
App:          0, 1000
Task:         1, 250, 2, 10, 30, 3, 5
Task:         2, 210, 4, 10
Task:         3, 190, 4, 5
Task:         4, 200, -1, 0
MAP:          1100, 3800, 1, 0, 2, 0, 3, 0, 4, 0
MAP:          4000, 6000, 1, 1, 2, 0, 3, 2, 4, 5
App:          1, 500
Task:         1, 120, 2, 5
Task:         2, 150, 3, 10
Task:         3, 190, -1, 0
MAP:          1200, 3500, 1, 0, 2, 0, 3, 0
MAP:          4200, 6200, 1, 7, 2, 8, 3, 9
SIM:          100, 10, 10, 12

*************************
Task Mapping Configuration
*************************
app: 0, root_task: 1, min_restart: 1000, leaf_task: 4
  task: 1, pb0 ( et: 250, st: 2, pl: 10 ), pb1 ( et: 30, st: 3, pl: 5 )
  task: 2, pb0 ( et: 210, st: 4, pl: 10 ), ant: 1
  task: 3, pb0 ( et: 190, st: 4, pl: 5 ), ant: 1
  task: 4, pb0 ( et: 200, st: -1, pl: 0 ), ant: 2, 3

  mapping: 0, start: 1100, stop: 3800, task1(pe0), task2(pe0), task3(pe0), task4(pe0)
  mapping: 1, start: 4000, stop: 6000, task1(pe1), task2(pe0), task3(pe2), task4(pe5)

app: 1, root_task: 1, min_restart: 500, leaf_task: 3
  task: 1, pb0 ( et: 120, st: 2, pl: 5 )
  task: 2, pb0 ( et: 150, st: 3, pl: 10 ), ant: 1
  task: 3, pb0 ( et: 190, st: -1, pl: 0 ), ant: 2

  mapping: 0, start: 1200, stop: 3500, task1(pe0), task2(pe0), task3(pe0)
  mapping: 1, start: 4200, stop: 6200, task1(pe7), task2(pe8), task3(pe9)

pe: 0
  start-stop: 1100-3800, app0,task1, app0,task2, app0,task3, app0,task4
  start-stop: 1200-3500, app1,task1, app1,task2, app1,task3
  start-stop: 4000-6000, app0,task2
pe: 1
  start-stop: 4000-6000, app0,task1
pe: 2
  start-stop: 4000-6000, app0,task3
pe: 5
  start-stop: 4000-6000, app0,task4
pe: 7
  start-stop: 4200-6200, app1,task1
```

```
pe: 8
  start-stop: 4200-6200, app1,task2
pe: 9
  start-stop: 4200-6200, app1,task3

*******************************
TaskMapping Simulation Execution
*******************************
(00) @001090 pe000,app00,tsk01 -> being mapped to pe0.
(02) @001090 pe000,app00,tsk01 -> in sleeping state.
(00) @001090 pe000,app00,tsk02 -> being mapped to pe0.
(01) @001090 pe000,app00,tsk02 -> in waiting rx state.
(00) @001090 pe000,app00,tsk03 -> being mapped to pe0.
(01) @001090 pe000,app00,tsk03 -> in waiting rx state.
(00) @001090 pe000,app00,tsk04 -> being mapped to pe0.
(01) @001090 pe000,app00,tsk04 -> in waiting rx state.
(03) @001100 pe000,app00,tsk01 -> in ready state.
(05) @001100 pe000,app00,tsk01 -> in context switching state.
(06) @001110 pe000,app00,tsk01 -> in running state.
(09) @001110 pe000,app00,tsk01 -> starts its execution with pb0.
(00) @001190 pe000,app01,tsk01 -> being mapped to pe0.
(02) @001190 pe000,app01,tsk01 -> in sleeping state.
(00) @001190 pe000,app01,tsk02 -> being mapped to pe0.
(01) @001190 pe000,app01,tsk02 -> in waiting rx state.
(00) @001190 pe000,app01,tsk03 -> being mapped to pe0.
(01) @001190 pe000,app01,tsk03 -> in waiting rx state.
(03) @001200 pe000,app01,tsk01 -> in ready state.
(04) @001210 pe000,app00,tsk01 -> in ready state due to context switching.
(05) @001210 pe000,app01,tsk01 -> in context switching state.
(06) @001220 pe000,app01,tsk01 -> in running state.
(09) @001220 pe000,app01,tsk01 -> starts its execution with pb0.
(04) @001320 pe000,app01,tsk01 -> in ready state due to context switching.
(05) @001320 pe000,app00,tsk01 -> in context switching state.
(06) @001330 pe000,app00,tsk01 -> in running state.
(10) @001330 pe000,app00,tsk01 -> continues its execution with pb0.
(04) @001430 pe000,app00,tsk01 -> in ready state due to context switching.
(05) @001430 pe000,app01,tsk01 -> in context switching state.
(06) @001440 pe000,app01,tsk01 -> in running state.
(10) @001440 pe000,app01,tsk01 -> continues its execution with pb0.
(11) @001460 pe000,app01,tsk01 -> pb0 completes its processing.
(13) @001460 pe000,app01,tsk01 -> payload of pb0 is ready to be transferred.
(12) @001460 pe000,app01,tsk01 -> completes its pbs and releases the pe.
(02) @001460 pe000,app01,tsk01 -> in sleeping state.
(05) @001460 pe000,app00,tsk01 -> in context switching state.
(15) @001461 pe000,app01,tsk01 -> payload of pb0 being transferred to task in the same pe.
(19) @001461 pe000,app01,tsk02 -> receives payload of pb0 (task1) from same PE.
(08) @001461 pe000,app01,tsk02 -> receives all antecedent payloads.
(03) @001461 pe000,app01,tsk02 -> in ready state.
(30) @001461 app01,tsk01->tsk02,latency -> acum_flits=0,last=0,min=0,max=0,avg=0
…
(33) @006000 pe000,acum_cycles_performed -> contextswitching=330,taskexecution=2810,pe_idle=1580
(07) @006000 pe001,app00,tsk01 -> stops and is unmapped at @6000 (expected at @6000).
(33) @006000 pe001,acum_cycles_performed -> contextswitching=20,taskexecution=560,pe_idle=1430
(07) @006000 pe002,app00,tsk03 -> stops and is unmapped at @6000 (root stopped at @6000).
(33) @006000 pe002,acum_cycles_performed -> contextswitching=20,taskexecution=380,pe_idle=1610
(07) @006000 pe005,app00,tsk04 -> stops and is unmapped at @6000 (root stopped at @6000).
(33) @006000 pe005,acum_cycles_performed -> contextswitching=20,taskexecution=360,pe_idle=1630
(03) @006090 pe007,app01,tsk01 -> in ready state.
(05) @006090 pe007,app01,tsk01 -> in context switching state.
(06) @006100 pe007,app01,tsk01 -> in running state.
(09) @006100 pe007,app01,tsk01 -> starts its execution with pb0.
(07) @006200 pe007,app01,tsk01 -> stops and is unmapped at @6200 (expected at @6200).
(33) @006200 pe007,acum_cycles_performed -> contextswitching=40,taskexecution=460,pe_idle=1510
(07) @006200 pe008,app01,tsk02 -> stops and is unmapped at @6200 (root stopped at @6200).
(33) @006200 pe008,acum_cycles_performed -> contextswitching=30,taskexecution=450,pe_idle=1530
(07) @006200 pe009,app01,tsk03 -> stops and is unmapped at @6200 (root stopped at @6200).
(33) @006200 pe009,acum_cycles_performed -> contextswitching=30,taskexecution=570,pe_idle=1410

**********************
Task Mapping Statistics
**********************

*** App0 - Mapping0 ***
Execution Time for Completed Executions (from root to leaf) (executions,min,avg,max)=1,2030,2030,2030
Latency per Branch for all Flits Transmitted (from start to stop) (flits,min,avg,max)
  br0: 1-2=0,0,0,0 -> 2-4=0,0,0,0 -->> T=0,0,0,0
  br1: 1-3=0,0,0,0 -> 3-4=0,0,0,0 -->> T=0,0,0,0
Critical Branch Latency for Completed Executions (from root to leaf) (flits,min,avg,max)=0,0,0,0

*** App0 - Mapping1 ***
Execution Time for Completed Executions (from root to leaf) (executions,min,avg,max)=1,740,740,740
Latency per Branch for all Flits Transmitted (from start to stop) (flits,min,avg,max)
  br0: 1-2=20,5,5,5 -> 2-4=20,7,7,7 -->> T=40,12,12,12
```

```
  br1: 1-3=10,5,5,5 -> 3-4=10,27,27,27 -->> T=20,32,32,32
Critical Branch Latency for Completed Executions (from root to leaf) (flits,min,avg,max)=10,32,32,32

*** App1 - Mapping0 ***
Execution Time for Completed Executions (from root to leaf) (executions,min,avg,max)=1,1270,1270,1270
Latency per Branch for all Flits Transmitted (from start to stop) (flits,min,avg,max)
  br0: 1-2=0,0,0,0 -> 2-3=0,0,0,0 -->> T=0,0,0,0
Critical Branch Latency for Completed Executions (from root to leaf) (flits,min,avg,max)=0,0,0,0

*** App1 - Mapping1 ***
Execution Time for Completed Executions (from root to leaf) (executions,min,avg,max)=3,524,524,524
Latency per Branch for all Flits Transmitted (from start to stop) (flits,min,avg,max)
  br0: 1-2=15,11,11,11 -> 2-3=30,5,5,5 -->> T=45,16,16,16
Critical Branch Latency for Completed Executions (from root to leaf) (flits,min,avg,max)=45,16,16,16

*** pe: 0 ***
Total Context Switching Cycles=330
Total Task Execution Cycles=2810
Total PE Idle Cycles=1580
Total Mapped Task Cycles=4720
Total Mapped Task Cycles (expected)=4720
Total Energy (J)=0.00140392
      Context Switching Energy (J)=0.00014751
      Task Execution Energy (J)=0.00125607
      Idle Energy (J)=3.4444e-07

*** pe: 1 ***
Total Context Switching Cycles=20
Total Task Execution Cycles=560
Total PE Idle Cycles=1430
Total Mapped Task Cycles=2010
Total Mapped Task Cycles (expected)=2010
Total Energy (J)=0.000259572
      Context Switching Energy (J)=8.94e-06
      Task Execution Energy (J)=0.00025032
      Idle Energy (J)=3.1174e-07

*** pe: 2 ***
Total Context Switching Cycles=20
Total Task Execution Cycles=380
Total PE Idle Cycles=1610
Total Mapped Task Cycles=2010
Total Mapped Task Cycles (expected)=2010
Total Energy (J)=0.000179151
      Context Switching Energy (J)=8.94e-06
      Task Execution Energy (J)=0.00016986
      Idle Energy (J)=3.5098e-07

*** pe: 5 ***
Total Context Switching Cycles=20
Total Task Execution Cycles=360
Total PE Idle Cycles=1630
Total Mapped Task Cycles=2010
Total Mapped Task Cycles (expected)=2010
Total Energy (J)=0.000170215
      Context Switching Energy (J)=8.94e-06
      Task Execution Energy (J)=0.00016092
      Idle Energy (J)=3.5534e-07

*** pe: 7 ***
Total Context Switching Cycles=40
Total Task Execution Cycles=460
Total PE Idle Cycles=1510
Total Mapped Task Cycles=2010
Total Mapped Task Cycles (expected)=2010
Total Energy (J)=0.000223829
      Context Switching Energy (J)=1.788e-05
      Task Execution Energy (J)=0.00020562
      Idle Energy (J)=3.2918e-07

*** pe: 8 ***
Total Context Switching Cycles=30
Total Task Execution Cycles=450
Total PE Idle Cycles=1530
Total Mapped Task Cycles=2010
Total Mapped Task Cycles (expected)=2010
Total Energy (J)=0.000214894
      Context Switching Energy (J)=1.341e-05
      Task Execution Energy (J)=0.00020115
      Idle Energy (J)=3.3354e-07

*** pe: 9 ***
Total Context Switching Cycles=30
```

```
Total Task Execution Cycles=570
Total PE Idle Cycles=1410
Total Mapped Task Cycles=2010
Total Mapped Task Cycles (expected)=2010
Total Energy (J)=0.000268507
      Context Switching Energy (J)=1.341e-05
      Task Execution Energy (J)=0.00025479
      Idle Energy (J)=3.0738e-07

*** pe overall ***
Total Context Switching Cycles=490
Total Task Execution Cycles=5590
Total PE Idle Cycles=10700
Total Mapped Task Cycles=16780
Total Energy (J)=0.00272009
      Context Switching Energy (J)=0.00021903
      Task Execution Energy (J)=0.00249873
      Idle Energy (J)=2.3326e-06

***************
WNoC Statistics
***************

% Total received packets: 14
% Total received flits: 105
% Received/Ideal flits Ratio: 7.29167e-05
% Average wireless utilization: 0
% Global average delay (cycles): 8.71429
% Max delay (cycles): 26
% Network throughput (flits/cycle): 0.0116667
% Average IP throughput (flits/cycle/IP): 0.000729167
% Total energy (J): 8.93539e-06
%   Dynamic energy (J): 1.76853e-09
%   Static energy (J): 8.93362e-06
```

4.7. Files and Modules Added to Noxim

The following modules were added to Noxim to support the characteristics previously mentioned:

TaskMapping.cpp and *.h (1051 + 926 lines)

- Reads the mapping from a file.
- Creates lists, vectors and maps to manage the mapping.
- Checks the consistency of the mapping configuration.
- Shows info about the mapping.
- Instantiated at the NoC module.

PEMappedTaskExecution.cpp and *.h (805 + 135 lines)

- Manages the mapping and execution states of each task.
- Computes the time to map, wake up and unmap tasks, and check the execution of the tasks.
- Collects latency, execution time and energy metrics.
- Send data from one PE to another through the WNoC infrastructure.
- Gets data from the WNoC infrastructure and sends it to the corresponding PE.
- Instantiated at each PE module.

5. Conclusions and Future Work

In this report we introduced a set of extensions carried out to Noxim simulator to ease the evaluation of different task mapping mechanisms employed to perform the arrangement of tasks onto the resources of WNoC platforms. The evaluation is done in terms of latency, energy and execution time. We also defined a proper and formal representation of the applications called annotated task graph which is used to describe the behavior of the tasks.

The task mapping simulation mechanism adopted in Noxim allows the mapping and execution of several applications. It is possible to specify up to l applications through their ATGs, where each of them can define up to n tasks and any task can be mapped to any of the m PEs available. It is possible to define up to q mappings without overlapping for each application. In addition, each task could have from 1 to p PBs of execution specifying the cycles to be executed and the payload to be sent to the associated subsequent task. The mechanism that decides when a task can be assigned to the corresponding PE according to the mapping established is a simple Round Robin scheduler which also allows the execution of multiple tasks on the same PE.

As a future work, we want to add cluster configuration support to Noxim simulator as presented by some authors in the literature. We also need to improve the model to estimate the power consumption in the processing elements.

6. Acknowledgments

This technical report is part of the PhD thesis project called *A Task Mapping Approach with Reliability Considerations for Multicore Systems based on Wireless Network-on-Chip* being developed by the student Luis Germán García M at the Embedded Systems and Computational Intelligence SISTEMIC research group. It is affiliated to the Electronics Engineering Department, Faculty of Engineering, University of Antioquia, Colombia. This project is supported by the Department of Science, Technology and Innovation COLCIENCIAS, Colombia, under the grant 647 of 2014 named Doctorados Nacionales 2014 (Scholarship 2015-2018). Special thanks to Vincenzo Catania, Andrea Mineo, Salvatore Monteleone, Maurizio Palesi and Davide Patti for developing and making available Noxim simulator.

7. References

Atitallah, R. Ben, Niar, S., Greiner, A., Meftali, S., & Dekeyser, J. L. (2006). Estimating energy consumption for an MPSoC architectural exploration. *Proceedings of the 19th International Conference on Architecture of Computing Systems*, 298–310. http://doi.org/10.1007/11682127_21

Catania, V., Mineo, A., Monteleone, S., Palesi, M., & Patti, D. (2015). Noxim: An Open, Extensible and Cycle-accurate Network on Chip Simulator. In *IEEE International Conference on Application-specific*

Systems, Architectures and Processors 2015.

Ganguly, A., Pande, P., Belzer, B., & Nojeh, A. (2011). A unified Error Control Coding scheme to enhance the reliability of a hybrid wireless Network-on-Chip. *Proceedings - IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, 277–285. http://doi.org/10.1109/DFT.2011.24

Li, X. (2012). Survey of Wireless Network-on-Chip Systems, 60.

Maqsood, T., Ali, S., Malik, S. U. R., & Madani, S. A. (2015). Dynamic task mapping for Network-on-Chip based systems. *Journal of Systems Architecture*, *61*(7), 293–306. http://doi.org/10.1016/j.sysarc.2015.06.001

Sangiovanni-Vincentelli, A. (2007). Reasoning about the trends and challenges of system level design. *Proceedings of the IEEE*, *95*(3), 467–506. http://doi.org/10.1109/JPROC.2006.890107

Wang, C., Hu, W. H., & Bagherzadeh, N. (2011). A Wireless Network-on-Chip Design for Multicore Platforms. *2011 19th International Euromicro Conference on Parallel, Distributed and Network-Based Processing*, 409–416. http://doi.org/10.1109/PDP.2011.37

Yu, X., Baylon, J., Wettin, P., Heo, D., Pande, P. P., & Mirabbasi, S. (2014). Architecture and design of multichannel millimeter-wave wireless NoC. *IEEE Design and Test*, *31*(6), 19–28. http://doi.org/10.1109/MDAT.2014.2322995

Yu, X., Sah, S. P., Rashtian, H., Mirabbasi, S., Pande, P. P., & Heo, D. (2014). A 1.2-pJ/bit 16-Gb/s 60-GHz OOK transmitter in 65-nm CMOS for wireless network-on-chip. *IEEE Transactions on Microwave Theory and Techniques*, *62*(10), 2357–2369. http://doi.org/10.1109/TMTT.2014.2347919