Operating System Support for IPNoSys

Silvio Roberto Fernandes de Araújo
Universidade Federal Rural do Semiárido, Dep. de Ciências Exatas e Naturais,
Mossoró, Brazil, 59625-900
silvio@ufersa.edu.br

and

Ivan Saraiva Silva
Universidade Federal do Piauí, Dep. de Informática e Estatística,
Teresina, Brazil, 64049-550
ivan@ufpi.edu.br

Abstract

The IPNoSys is an architecture that exploits the advantages of NoCs as parallel communication, reusability and scalability to transform the routers in processing elements building a packet-driven architecture that processing while routing the packets. This represents a paradigm break of traditional NoC-based MPSoC systems, which there is the separation between computation and communication. With new paradigm, such architecture already showed superiority in execution time comparing to an equivalent MPSoC. In this paper is presented the operating system support for IPNoSys, including the memory management, process management, I/O management, interruption, exception and timer. Additionally, it is proposed two versions of multi-task scheduling, the first one is a preemptive and the second a non-preemptive. In some cases the scheduling algorithms improvement the throughput of system up to 80%.

Keywords: IPNoSys, Operating System Support, Scheduling

1 Introduction

Dual, quad and eighth-core processors are a market reality, and too soon it will see many-cores architectures [1]. The design of multi-core architectures demands also considerable attention on communication infrastructure characteristics.

Bus-based design remains useful while the number of cores in the processor is kept to a limit. However, with the continuous growth of integration capability this will not hold for a long time. The network-on-chip (NoC) emerged as the most adequate interconnection mechanism to integrated systems that demand mighty processing and great data flow, since that NoC is reusable and has high scalability. NoCs are formed by a set of routers and point-to-point bidirectional channels that link the system cores [2]. The communication performed in NoCs uses messages encapsulated in packets and for the transmission, the packets flow from the source to destination through a path of neighboring routers in a parallel and pipelined way [3]. The main disadvantages of the NoC design are chip area and power dissipation rising [4]. Even so, the source for the most part of area and power dissipation in MPSoCs comes from the processing elements and not from the network's routers. Due to its distributed nature MPSoCs design suffer from problems such as data coherency and consistence in caches, which are even harder when NoCs architectures are used instead of busses. Solutions on this concern can be found in [5].

However, some viability experiments [6] proved that it is possible to take advantage of NoC characteristic to design an architecture based on packet-driven execution where the routers are able to incorporate the execution of application instructions while the packets flow from the source to the destination. Since these experiments was proposed such architecture called IPNoSys [7]. The traditional NoC design uses the set of routers only for the data transmission. In the IPNoSys approach, processing elements are added to the routers. It has also a programming model, where primitives are defined to allow the network to be programmed for general purpose applications[8].

The input/output system for this architecture was proposed in [9], additionally it was also developed a single-tasking scheduler to take advantage of waiting for I/O. In this paper, it is proposed a multi-tasking scheduler using two algorithms, one preemptive and one non-preemptive version. Then, this paper will present the operating system support for IPNoSys that corresponds to the memory management, process management, I/O management, interruption, exception and timer.

The following section presents the related works. Section 3 presents the architecture's features and its components. Section 4 presents the operation system support, following is the section 5 presents the simulation results obtained by the two scheduling algorithms. Section 6 presents the conclusions and future works.

2 Related Works

In this section are presented architectures that have some similarity in the computing model comparing to our proposed architecture. Furthermore, it is also presented some works related operating system support, mainly the scheduling in multiprocessor architectures.

The first kind of architecture that has computing model with some similarity is the queue machine or queue processors [10]. This computing model is based on the idea that the instruction and operands can be placed in a queue in the order that should be executed. The main difference to our proposed is that they use a queue processor instead of NoC to the application execution. The queue machines, or queue-based computers can be an efficient model to pipeline execution once they use implicit reference to an operand queue as a stack machine does [10]. Some researchers noticed that it is possible to build an efficient superscalar or data flow machine through queue machine due to the fact that the operands and instructions are aligned with each other in this machine[11]. According to [12], queue machines are a novel alternative for embedded architectures due to their compact instruction set, high instruction level parallelism and simple hardware that reduce the chip area and power dissipation. Thus, several computing models have been proposed to queue machine[13], [14]. The computing model of the queue machine is, in some way, similar to the system proposed in this paper: operations and operands are only removed from the head of the queue (or packet) and operation results are placed in the operands queue tail (or packet appropriate location). In queue machines, the result can also be placed in any position different from the tail, when it was associated to a priority that determines the position [11].

This computing model suggests that the execution of application could be done by dataflow architecture, the second type of architecture in our comparison. Following this thought, we can present the dataflow architecture proposed in the patent [15], [16], which performed part of computation of the applications in a NoC. The architecture is formed by a normal Von Neumann processor and two networks on chip (data network and instruction network). The processor fetches the instructions and performs those which are out of a loop. When else, the processor configures the networks and distributes the data through a bus to execution in the processing elements, and the connections of the data network create a dataflow graph. This architecture has a great powerful parallel processing only when there are instructions within loop in the applications. And the parallelism exploited in this architecture is only ILP (instruction level parallelism). Furthermore, it is noticed that the scalability is limited by the bus, the dataflow graph which determines the ALU's configuration is limited by NoC size and the instruction network depends on processor's program counter to fetch the instructions.

Other dataflow architecture is presented [17] which is a reconfigurable solution with coarse grain formed by an operative layer, a configuration layer and a custom RISC core with a dedicated instruction set as configurations controller. The operative layer uses a coarse-grained granularity component called Dnode, which is a datapath component, configured by a microinstruction code, with an ALU and few registers. This architecture is thus not intended to be a stand-alone solution, rather an IP core accelerator for data oriented intensive computing, which would take place in a SoC. The Dnodes are organized in layers that are connected to the two others adjacent layers by also dynamically configurable switch component able to make any interconnection between two stages.

In terms of microarchitecture configuration, the iWarp chip [18] is similar to the proposed in this paper. iWarp can implement a variety of processor interconnection topologies including one-dimensional (1D) arrays, rings, two-dimensional (2D) arrays, and tori and is intended for systems of various sizes ranging from several processors to thousands of processors. The communication can be done through two models: message passing and systolic. The systolic model is more similar to our proposal, though the iWarp use static scheduling to ensure the synchronism of the computation between the processors. Therefore, it is necessary a pre-configuration according to application features before execute it.

Architecturally there are several solutions comparable to our proposal, however, such publications do not discuss how is done the operating system (OS) support these solutions. Thus, it will briefly review on operating system support on multiprocessors.

In a multiprocessor environment, the multiple cores (processing elements, memories, I/O etc.) interact with each other through communication mechanisms such as buses, bus hierarchy and more recently networks-on-chip. The multiprocessor design enables today's semiconductor manufacture to integrate all these cores on a single physical chip, and so it is called multiprocessor system-on-chip, or MPSoC [19]. When is being developed to support operating systems, the MPSoC designers must, among other things, worrying if there is shared memory, which the cache coherency mechanism, how to divide the applications between processors, if management of hardware resources are either centralized or distributed and still achieve better performance and lower power consumption.

Considering the task scheduling there are several algorithms for a single processor, some of which are used in multiprocessor systems. These algorithms are used under the partitioning scheme or under the global scheduling scheme [20]. A task-splitting algorithm dispatches in a way that ensures that a task never executes on two or more processors simultaneously. A task-splitting, called slot-based task-splitting dispatching, has ability to schedule tasks with high processor utilizations, however this algorithm is not implemented in real operating systems. Therefore, [21] done some modification in this algorithm to running in Linux kernel, reaching an utilization up to 88% of a 4-core processor with real-time tasks.

Other scheduling algorithms like Rate Monotonic (RM), Earliest Dead line First (EDF) and Least Slack Time first (LST) should be used in multiprocessors if they did not cause deadline miss, thus [20] proposed a new multiprocessor scheduling algorithm, called Highest Actual Density first (HAD) for a real-time scheduling of dynamic priority jobs, using the global approach. This new algorithm avoids the anomalies presented in the other but its major disadvantage is the overhead.

An important aspect, and a reality today, is the use of NoC as communication mechanism in a MPSoC. Following this idea, [22] proposed a new scheduling algorithm for NoC-based MPSoC to overcome the limitation of traditional processor scheduler that fails to maintain optimized node/memory traffic and also it is responsible by traffic contention and network latency rise. The new algorithm takes into consideration of on-chip topology, scheduling decisions are made based on such information. Furthermore, it was proposed a protocol between the MPSoC and the OS to describe the physical topology in order to inform the physical position of the cores used by scheduling algorithm. However, this new algorithm is not yet implemented.

Other mechanism to support task scheduling on multiprocessor system is presented in [23]. The main objective of this method is to maximize system resource utilization, by allocating available platform resources to a task based on the individual task characteristics and performance requirements. To reaching that, it was proposed a system with a high speed database and a supporting hardware mechanism for runtime resource update. The database stores the task related metadata information. The hardware mechanism provides the system resource status information to the scheduler at runtime. When a task is to be scheduled, the scheduler uses the hardware mechanism and the database to evaluate the task behavior with respect to an available system topology. The scheduler can then decide on how the task should be executed. Using this method of dynamic mapping it yielded an improvement of 8% in the execution time and 10% decrease in the power consumption comparing to static mapping.

3 The IPNoSys Architecture

As presented in the previous section, there is an effort, for a long time, to increase performance in the execution of applications. Whether optimization of hardware/software, implementation of new architecture paradigms or improvement of resource management by the OS. Networks-on-Chip (NoC) enable parallel communication of packets between cores in MPSoCs, with packets having temporarily being allocated to buffers in their path to destination. Dataflow architectures are all programmable computers where the hardware is optimized for fine-grain data-driven parallel computation [24]. Taking this scenario, it is possible to syntactically describe programs as a collection of packets comprised by instructions and operands. Also, it is necessary to modify the routers to enable them to execute at least one instruction (part of a packet) while routing packets. The extra area needed to allow routers to execute instructions must be suppressed from other parts of MPSoC. This is achieved in IPNoSys removing regular processing elements and replacing them by Arithmetic/Logical Units inside each router. This way, IPNoSys approach creates a new paradigm in NoCs usage on MPSoC designs, since communication and computation can no longer be considered apart from each other during the design process like dataflow architectures.

The main idea of this architecture is to extend the interconnection of NoC mechanism to turn it also an element of instruction execution as a datapath architecture [25]. IPNoSys (Integrated Processing NoC System) is based in a direct square 2D-mesh NoC with following features: XY routing policy, a combination of VCT (Virtual-Cut-Through) and wormhole switching scheme, virtual channel, credit-based flow control, distributed arbitration and input buffering.

IPNoSys includes, in the router data path design, an arithmetic logic unit (ALU) providing the router to perform all logic-arithmetic operations of the applications. Therefore, a router accomplishes routing and processing tasks and because of that it is called Routing and Processing Unit (RPU). Besides one ALU, the RPU has a Synchronization Unit (SU) that enables it to perform synchronization instructions among RPUs. It is used shared memory space distributed in four memory modules placed in the four network's corners. In the memory modules, data and applications are stored in packet form. The memory modules are accessed by Memory Access Units (MAUs), which are placed also in the network's corners. Fig. 1 shows a 4x4 IPNoSys architecture instance.

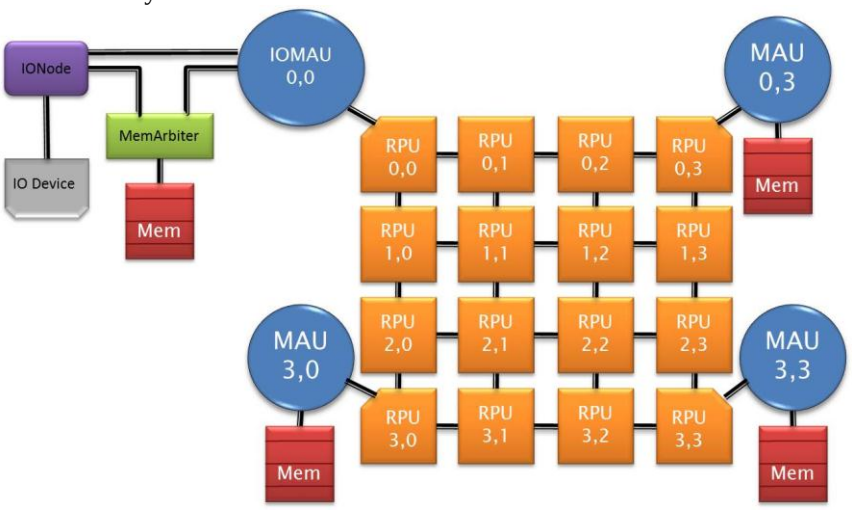


Figure 1: IPNoSys Architecture

In order to perform the input/output (I/O) operations, IPNoSys implements a DMA (Direct Memory Access) mechanism through IONode. For this, one of the four MAUs establishes the communication between the IONode and remaining of architecture. This MAU differs from the other by having an interface with the IONode, due that is called IOMAU. As the IONode should have the same access privileges memory as the IOMAU, it was necessary add an arbiter that controls this access (MemArbiter) that implements a hand shake protocol. The IOMAU is responsible to program the IONode, informing the command to read/write, I/O address, memory address and quantity of bytes to read/write. Therefore the IONode works independently transferring data between the I/O device and memory, and the IOMAU continues working as the other MAUs. It is only when the transfer complete is that IONode signals to IOMAU, as presented in [9].

To describe programs, IPNoSys uses a packet format where it is placed instructions and operands to be executed, and information that assist in routing and support the operating system, as shown in Fig. 2.

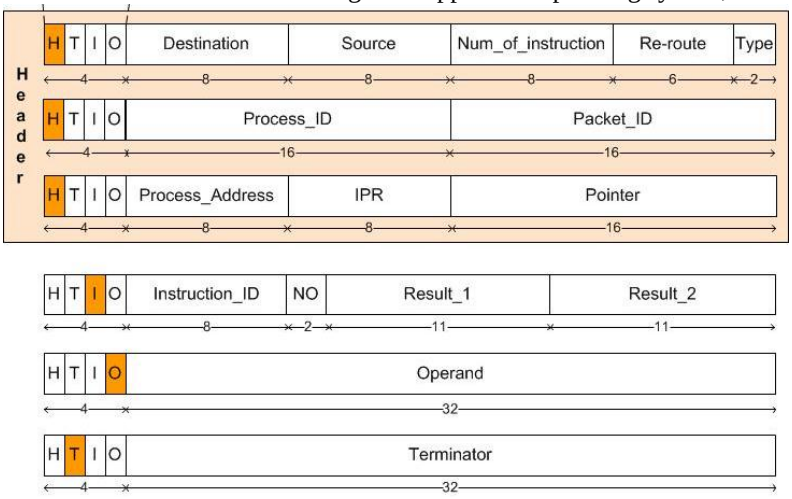


Figure 2: Packet format of IPNoSys

It corresponds to a variable set of 32-bits words. There are four types of words in a packet: header, instruction, operand and terminator. To identify each kind of word are used four control bits in which only one is set at a time.

The header has three words. The first word has the current source and destination, the current number of instruction in the packet, six bits that determine how to calculate the next new destination and the 2-bits flag that determines the type of the packet (regular, control, interrupted and caller packets). The regular packet has the instructions performed in any RPU. Control packets have instructions that are performed only in a target MAU. The interrupted packets are those that cannot be executed due to waiting for I/O. The caller is a packet that also cannot be executed due to waiting the execution of a function packet that it called. The second word has an identifier for the process and an identifier a packet of the process. Each application must have an exclusive process identifier, that possibly can be implemented by several packet, each one with its identification. The third word has the MAU's address that keeps the process' information (Process_Address); the IPR (instruction per RPU) that determines how many instructions that each RPU must be execute before transmit the rest of packet; and a pointer to the next instruction into the packet to be performed. The pointer keeps the number of transmitted words before the next instruction, including the words performed by other RPUs. This allows a global counting of the words that is useful for results insertion in the packet. The instruction word has the identifier of the instruction, the number of operands (up to two operands) and two fields, used, in general, to indicate the number of the words in the packet that are the destination location of the generated result. The instructions of control packets use these two fields to indicate the coordinate of the MAU that will perform the instruction and to indicate the number of operands (more than two). The operand word and the terminator word have only one field, with the operand and the ending pattern, respectively.

Taking into account the pipelined packets transmission on the NoC, the proposed execution mode establishes that in each RPU on the packets path, the first application instruction and operands will be removed and the instruction will be executed. The generated result (if any result is produced) must be inserted in a specific position in the same packet. The remaining packet, without the instruction and operands used (Fig. 3), is sent to the next RPU in the path where the execution procedure begins again for the next instruction or send to a MAU that stores in the memory. In this figure, each word of the packet is numbered only to illustrate how the instructions in the packet indicate the positions where their results should be entered. Notice that the packet before the execution had four instructions and the pointer indicated that the instruction to be executed in the third word of the packet (Fig. 3 (a)). After running (Fig. 3 (b)) the packet changed to three instructions, new operands have been entered and the pointer indicates the next instruction to be executed is the fifth word.

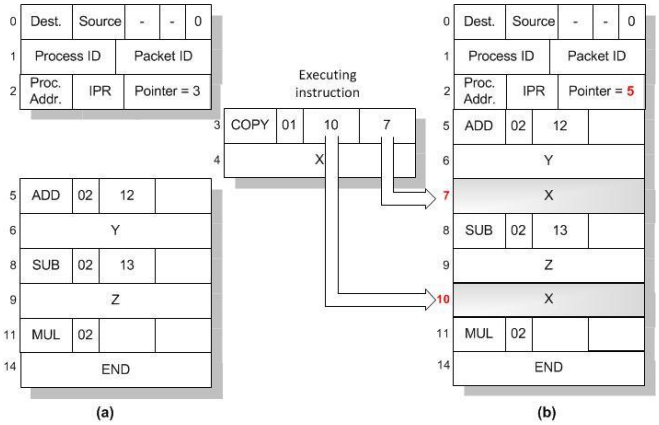


Figure 3: Execution Mode IPNoSys: (a) before executing; (b) after executing

It is noticed that the packet size decreases, on average, while it flows (instructions are executed) from source to destination, which allows reducing the network load. The packet size does not decrease all the time during its processing. When an instruction is executed, the packet size changes dynamically: it may decrease, increase or maintain. The variation of packet size depends on the kind of instruction that is executed. If the executed instruction inserts more result's words than removed words (instruction's word and the operands' words) the packet increases. If the opposite occurs then the packet size decreases. It is also possible that the packet size does not change. This happens when the number of result's words is the same of the removed words. However, on average, the number of removed words is higher than inserted words and, therefore, the packet size decreases.

The IPNoSys routing scheme was not conceived to just send packets from a specific source node to a specific destination node in networks on chip. This routing is intended to provide enough operating resources (RPU's ALU or RPU's SU) in a path to execute all instructions in application packets. In this path, each RPU executes the first instruction of the packet and then removes it. Therefore, the packet destination must be sufficiently distant from the source to execute all instructions of the application. Thus, to guarantee that all instructions are executed, a variation of the XY routing policy was developed. The

execution of all instructions is enabled by the property of the routing algorithm of finding a new destination to packets when they arrive at its destination and still have instructions to be executed.

Fig. 4 represents the packet being injected initially in the upper left corner. Each arrow represents an intermediary destination calculated by the routing algorithm. In this case the end of the spiral is the lower left corner. When a packet passes through a completed first spiral and still has instruction to be performed, a second spiral begins at the lower left corner, ending in the lower right corner. The execution could continue by a third and fourth spiral beginning, respectively, at the lower right corner and the upper right corner, coming back to the begin of the first spiral. The routing algorithm creates a path that allows the execution of all instructions in any packet. Additionally, in a global view, it distributes the data traffic between the physical channels of the network. Depending on the packet size and the network dimensions, the circular movements of the packet could cause a deadlock.

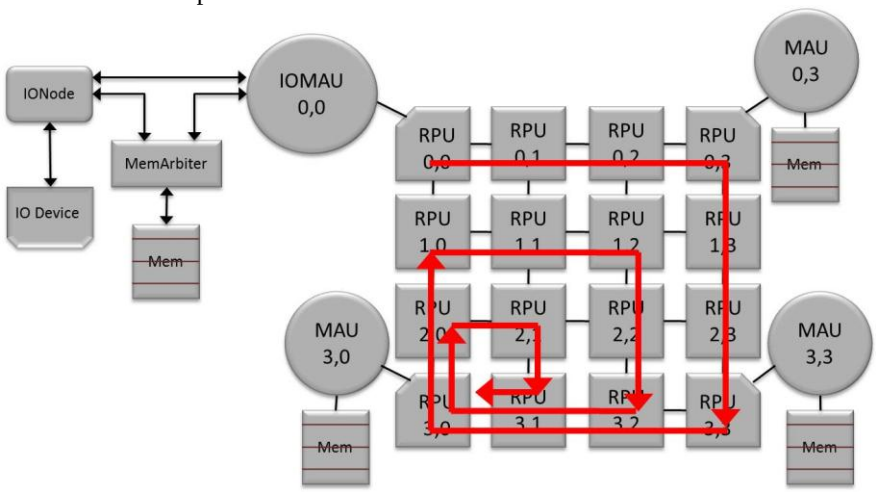


Figure 4: Routing way

The IPNoSys treats an imminent deadlock through a solution that was called local execution. This is detected when it is not possible to transmit the packet's header from the current RPU to the next in the execution path. In this case the RPU that keeps packet's header maintains the header and execute the first instruction on the packet until the moment when it is possible to transmit the packet's header and the current first instruction to the next RPU. Local execution means that a single RPU will remain executing instruction until it can share this task with others RPUs in the execution path.

The IPNoSys's programming model is based on packet structure to comply with the way messages normally are created for NoCs. In each packet, instructions are queued according to the data dependencies among subsequent operations. Such dependencies establish the order that the packets and the instructions will appear in the packets. Thus, the results of previous instructions can be used as operand in following instructions, as presented in [8]. Using this programming model and the new paradigm in NoC has resulted in a powerful parallel architecture that has the execution time up to 3.5 times faster than an MPSoC platform equivalent [25],[8].

4 Operating System Support

An operating system is a logic solution to manage the hardware resources and to provide common services for application software. Among the most important functions of operating systems are: handling basic input and output operation; allocating storage and memory; and providing for sharing the computer among multiple applications using it simultaneously [26]. The architectures must offer some interface to support an operating system, that is in fact and software too.

The IPNoSys architecture offers operating system support through some instructions, specific fields in the packet format and functions embedded in MAUs. The major part of support is implemented in hardware, and therefore is not the operating system yet. Following, it will be discussed how IPNoSys provides the services of memory management, processes management, I/O management, interruption, exception and timer.

4.1 Memory Management

The memory in IPNoSys is available through four independent modules that together form a single address space. Each module is maintained by an exclusive MAU placed in the corners of the network, except the memory module that is accessed by the IOMAU and the IONode through MemArbiter.

Including, the access to this memory module is the most time consuming, since the IOMAU IONode need to order MemArbiter and only after its confirmation the access is performed actually.

The unique addressing mode implemented is the direct mode, which is performed by LOAD and STORE instructions. These instructions indicate the memory address by 32-bits operands, which can address up to 4 GByte in each memory module, i.e. the IPNoSys direct addressing reaches 16 GByte, considering the four modules. Both data and code are stored in the same memory modules, but the object code of programs has information that indicates amount data and code when they are being loaded into memory. Then, the MAUs can define the limits of data and code segments from this information. When a MAU executes LOAD/STORE instructions, it is verified if the address is in the range of data segment, in the opposite an exception is generated. Other instructions, like EXEC and SYNEXEC, are responsible to order the injection of a packet, in these cases they indicate the pair process ID/packet ID, which is converted by the MAU to the memory address where the packet is stored.

The real memory management is done through two linked lists, one for the free words and one for used words in the memory. Each element in the linked list has a bit indicating whether the word is in use or not and a pointer to the next list element. Usually the work of the MAU is to convert a process ID/packet ID to a memory address and inject it, however, some situations (it will be seen later) bring any other packet to be stored in memory. In such cases, the MAU uses the linked list of free words to know where save it.

4.2 Process Management

In the IPNoSys packets the instructions are queued according to the data dependencies among subsequent operations. Such dependencies establish the order that the packets that must be performed and the order the instructions will appear in the packets. Thus, the results of previous instructions can be used as operand in following instructions. Following this same reasoning, a program can be implemented through various packets collaboratively, often featuring some dependency or need for a sync point between them as presented in [8].

From the operating system point of view, a program becomes a process when starts executing. In this way, each IPNoSys packet should be identified by a pair "process/packet", and this is the reason to these two fields are in the packet's header. When the assembler is generating the object code of a program, each packet is numerated from 1 until the maximum of packets of the same program. Each program also has a packet number zero, which is used to define the end of program execution. The packet 1 is the first to be injected and it can start the execution the other packets through EXEC or SYNEXEC instructions from the same MAU or from the other. The MAU that maintains and injects a packet 1 of a program is called "original MAU" for that program. So, this MAU is responsible to make it in a process and updates its status while it is executing, like an operating system, and its address is used to set the “Process_Address” field in the header of all packet of the same process. All four MAUs have a task that works like a scheduler, it means that is implemented in hardware rather than software. Each scheduler is responsible to control only its process.

Every process can assume one of five statuses (ready, executing, waiting, blocked, or finished) at each moment as shown in Fig. 5.

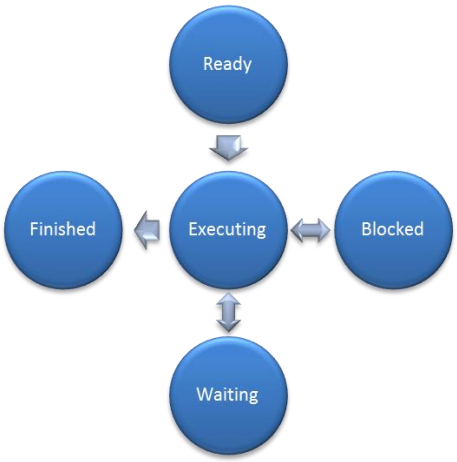


Figure 5: Relationship between the states of a process

All processes that will be started by the MAU have status “ready”, and all processes have the same priority, so the next process in the ready list is started and its status change to “executing”. When the MAU decides to stop the execution of a process to execute other, the stopped process has status changed

to “waiting”, and it can choose a ready process or other waiting process to execute. Eventually the initial packet can start other packets of the same process, and any of these may be interrupted due an I/O, changing the process’ status to “blocked”. When this happens, the MAU is informed that a packet is interrupted through a NOTIFY instruction, consequently the process’ status is modified. In this case the scheduler can choose other ready process to execute.

It is possible several packet of the same process are executing in parallel when one of them is interrupted due an I/O instruction. Only this packet is interrupted and stored in a MAU’s memory, while the other packet are still executing, because there is no dependency between them, since they are running in parallel. However, if there is any packet that depends on the interrupted packet to be injected, then this other will wait the end of the I/O operation when the interrupted packet returns.

When the interrupted packet returns to execute this is informed to the original MAU through a new NOTIFY. This causes the change of status “blocked” straight “executing” without going through “ready” before. If the process previously chosen by the scheduler is still running, both processes will have their status “executing”.

When the packet zero is executed a NOTIFY is sent to the original MAU to finish the process that such packet is part. At this moment the scheduler can choose the next ready to execute.

4.3 I/O Management

The I/O management is actually performed by IOMAU, although other components somehow participate in this process, through the instructions IN, OUT, WAKEUP and NOTIFY [9]. The complete I/O procedure is illustrated in Fig. 6. IN and OUT perform respectively input and output operations. WAKEUP is used to inform the end of I/O operation and consequently the interrupted process/packet can return to perform. NOTIFY is used to communication between the MAUs about the interruption of a process/packet. Only IN and OUT instructions are present in regular packets and they are decoded by the arbiter inside the RPU as other instructions during the regular packet execution (Fig. 6 step 1). However, beyond these instructions cause the production of control packets to send them to IOMAU, and the execution of regular packet which carries them is stopped.

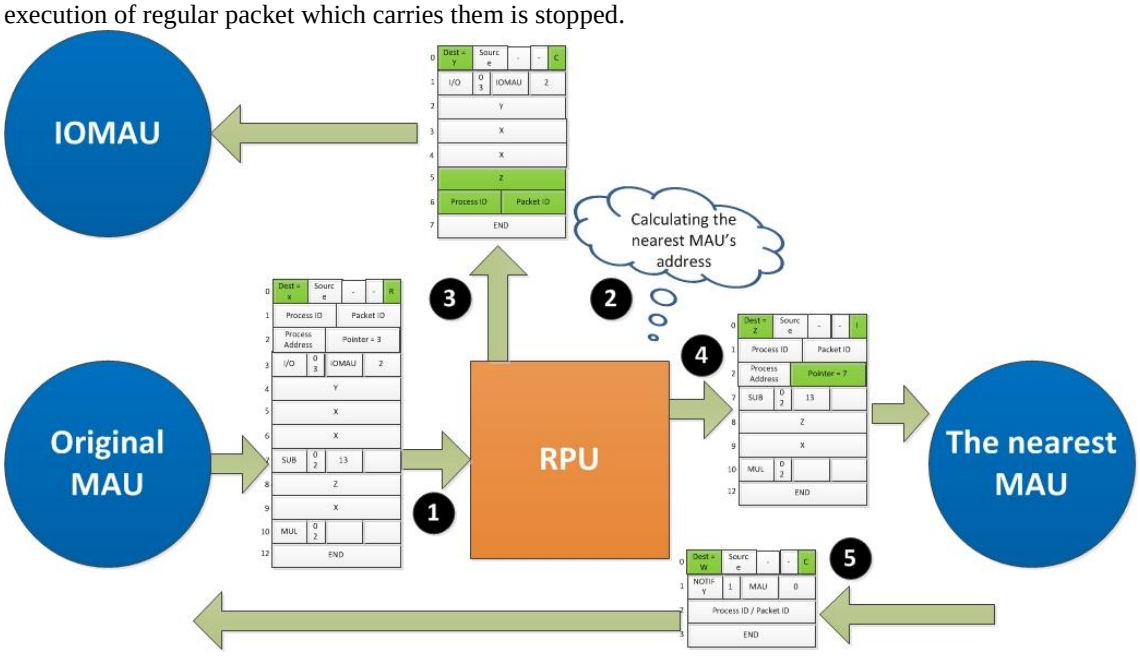


Figure 6: I/O Procedure

Then, this regular packet has its type, in the header’s first word, changed to interrupted packet. Next, it is calculated the address of the nearest MAU from the RPU where the packet was interrupted (Fig. 6 step 2). So, the number of this process/packet and the nearest MAU’s address also are put in the control packet before send it with I/O instruction to IOMAU (Fig. 6 step 3).

The interrupted packet is transmitted by virtual channel of control packet until the nearest MAU, which stores this packet in the memory (Fig. 6 step 4). A control packet with NOTIFY is also built and it is sent from this MAU to the original MAU, which address is obtained in the field “Process_Address” in the third word of regular packet header (Fig. 6 step 5). The notification can be used to inform that a process/packet was interrupted; or when an interrupted process/packet returns; or when a process was finished.

The control packet with I/O instruction arrives at IOMAU, which uses its operands to program the IONode. When the I/O operation is finished, the IONode signals to the IOMAU, which creates a control packet, containing WAKE UP instruction and the number of the interrupted process/packet. This WAKE UP is sent to the MAU that stored the interrupted packet, and it informs that this interrupted packet can return to perform.

When the control packet, containing WAKE UP instruction arrives in the MAU, there is more priority to the interrupted packet to be injected than regular packets of this MAU. However, if this MAU is injecting any packet, the interrupted packet will be injected as soon as the current injection is finished.

After that, this MAU sends other NOTIFY through control packet to original MAU that controls the process to inform about the return of process/packet previously interrupted.

Fig. 6 shows some highlight packet's fields to indicate the type of packets, the destination, the packet ID and the pointer that are updated during the I/O procedure.

4.4 Interruption, Exception and Timer

All external events are treated by IOMAU as well as programed I/O operation between it and IONode. Thus, when the IOMAU receive an interruption signal, it verifies if the interruption identification is an I/O operation started previously. In affirmative cases, it follows the procedure presented before. In other cases, each interruption ID must have a correspondent packet that will be injected from the IOMAU or other MAU. However, these packets correspond to operating system implementation.

With regard to exception handling, IPNoSys is able to detect and notifies each type of exception through the instruction NOTIFY that is driven to the MAU that decides to do some action or inject a packet of operating system, depends on the exception type. Among the exceptions that can be detected are: address of memory out of bounds, invalid instruction, division by zero, arithmetic overflow, nonexistent packet, etc.

The first version of the IPNoSys scheduler was single-tasking, i.e. a new process was chosen to execute only when another was finished or when an I/O operation happened. In this paper is proposed a timer mechanism to allow implementing a multitasking scheduler. There are two versions to new scheduler. The first one is a non-preemptive algorithm, which consist to start a new ready process after a quantum time, that is, starts a process that don't have no one packet executing, does not matter if there is any other process executing. The second one is an evolution of the first one. The main idea is to give a quantum time to each process that is executing and when this quantum expires, this process is changed to "waiting" and it chooses other process to execute. The choice of the next process gives priority to those "ready" processes and only after there is no "ready" process is gave priority to "waiting" processes. However, an important difference in the preemptive version is that while a process is "waiting" no packet of this process can be injected by the MAU that controls this process. When there is no process ready is used a round robin algorithm between "waiting" processes.

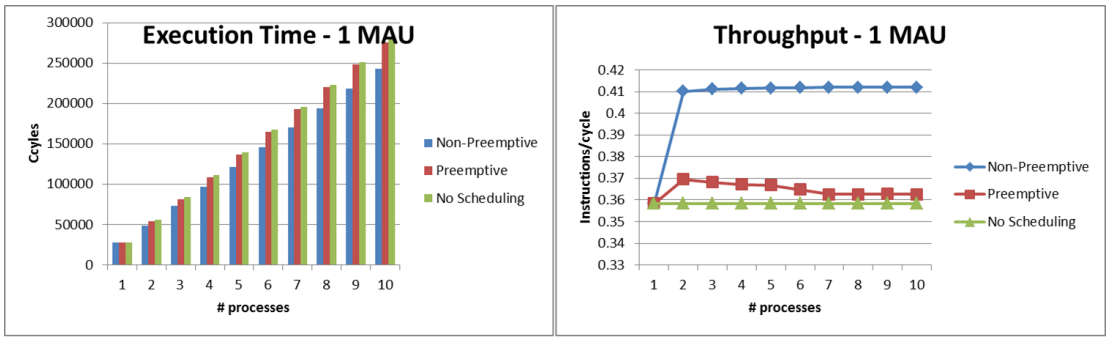
5 Simulation Results

In this section is presented some experiments used to evaluate the operating system support, mainly the scheduling mechanism, since in the task scheduling procedure are involved the memory management, process management and eventually I/O management. The first experiment consists to evaluate the scheduling for a set of simple processes executing in loop. The second is a set of processes that depend on synchronization of various parallel packets. All experiments are simulated using the two proposed scheduling model (using 1000 cycle for timer) comparing them to no-scheduling execution. In the no scheduling version, the execution of the first process is started and only it is finished a new process can execute and so on. While the scheduling versions can be preemptive or non-preemptive as it was discussed before.

In the first experiment each process is formed by only one packet that has 100 arithmetic instructions and the execution for each process correspond to a loop of 100 repetition of this unique packet. This experiment was performed using 1 to 3 MAUs (in parallel), and it was varied the number of processes in each MAU from 1 to 10.

Fig. 7(a) shows the execution time (in cycles) to perform 1 to 10 processes in 1 MAU comparing the three scheduling versions. The execution time is absolutely the same for the three scheduling versions simulating when it has only one process, however, while the number of processes increases, the execution time for non-preemptive scheduling increases in a smaller proportion than the other versions. The worst performance is no scheduling version, but the difference between this version and the preemptive is small. Considering the quantity of instruction in each simulation is the same, independent of scheduling version, the performance difference between them is clearer in Fig. 7 (b) that shows the throughput in instructions per cycle. In this figure is evident that the non-preemptive version performs better than the other two, as

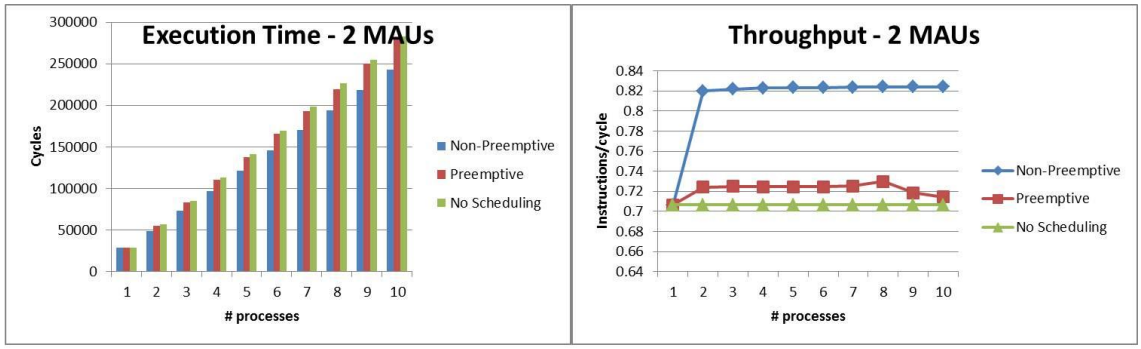
in each case the three version execute the same amount of instruction, it means that the non-preemptive version spends less time.



(a) (b)

Figure 7: Experiment 1 with 1 MAU - (a) execution time; (b): throughput

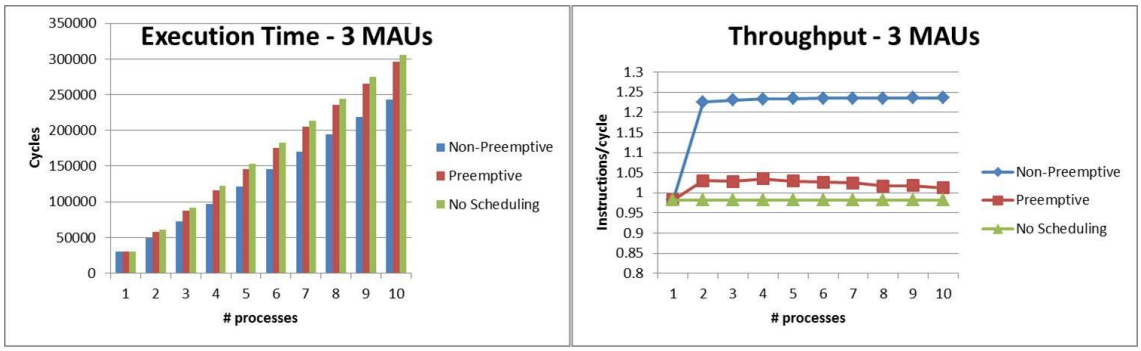
Fig. 8 shows the same experiment simulating in two parallel MAUs, i.e., the quantity of process, and consequently the number of instructions, is doubled, during the experiment. In this case, the behavior between the three version is similar than this experiment with 1 MAU. The non-preemptive version continues being superior than other two in the same proportion.



(a) (b)

Figure 8: Experiment 1 with 2 MAUs - (a) execution time; (b): throughput

Simulating the experiment 1 in 3 parallel MAUs (Fig. 9), it is noticed that the regularity of application conducts the experiment to a constant behavior independent of load offer to the architecture.



(a) (b)

Figure 9: Experiment 1 with 3 MAUs - (a) execution time; (b): throughput

When tripled the amount of load (3 MAUs in parallel), throughput (Fig. 9 (b)) is three times higher compared to the same experiment with a single MAU (Fig. 7 (b)), this is caused because the number of instructions is tripled but the execution time in both cases remains practically the same, compare Fig. 7 (a) and Fig. 9 (a).

This experiment was not performed using 4 MAUs because the memory access time to IOMAU is much higher than in other MAUs, then the IOMAU would be a bottleneck, which could be hide the real difference between the versions of the schedulers.

The second experiment uses processes formed by five packets and its terminator packet as illustrated in Fig. 10. The first packet is responsible to implement a loop of 100 repetitions and starts the execution of 4 parallel packets in each repetition. And, only after the synchronization of the 4 parallel packets the new loop is started. Each parallel packet executes 100 arithmetic instructions and 1 sync instruction that allows the packet 1 starts a new loop. This experiment was performed using 1 to 3 MAUs, and it was varied the number of processes in each MAU from 1 to 10.

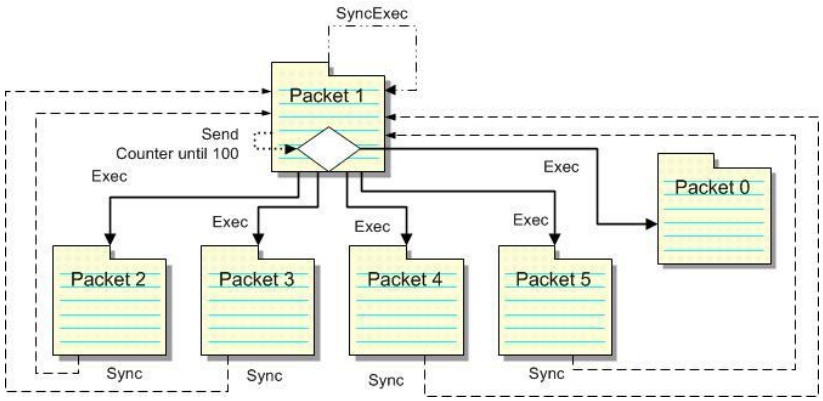
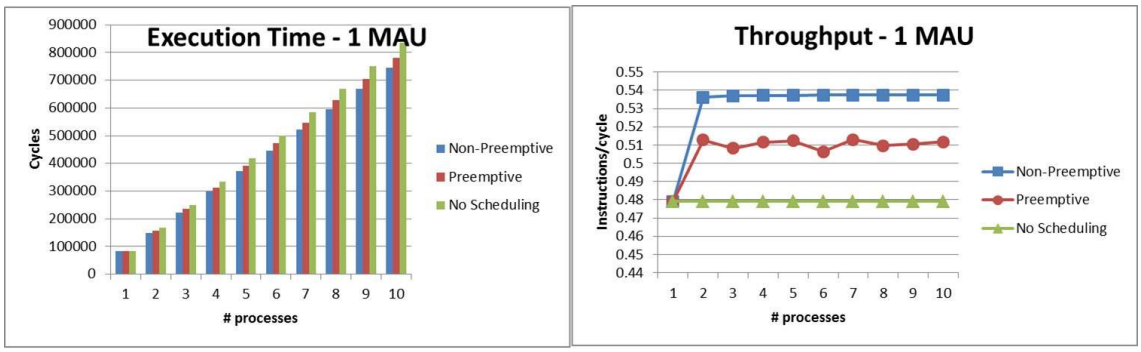


Figure 10: Relationship between the packets in the experiment 2

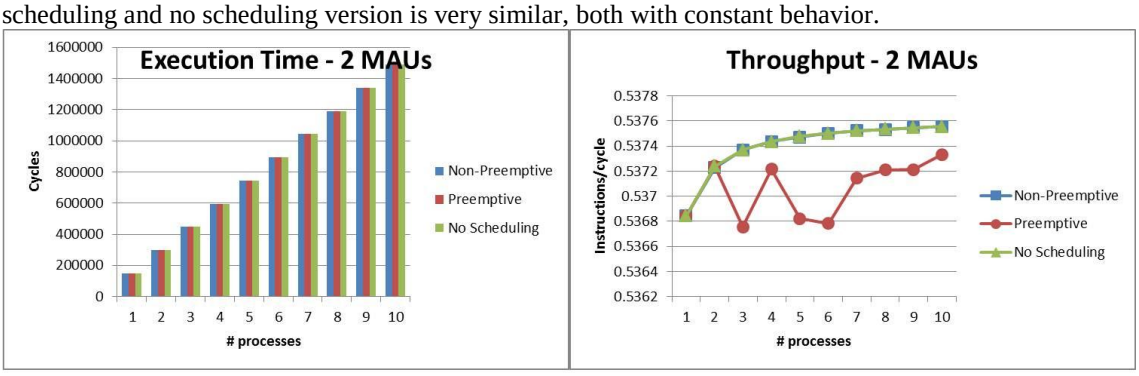
As in the experiment 1, at first, it was used only one MAU to perform the experiment 2 as shown in Fig. 11. In this experiment the difference of execution time in the three versions is more significant (Fig. 11(a)), however the non-preemptive version continues with the better throughput, following by preemptive version and the end the no scheduling version (Fig. 11(b)). The preemptive version has a good performance but its behavior is not constant as the other two as shown in Fig. 11(b).



(a) (b)

Figure 11: Experiment 2 with 1 MAU - (a) execution time; (b): throughput

When the experiment 2 is executed in 2 parallel MAUs (Fig. 12), the performance of non-preemptive scheduling and no scheduling version is very similar, both with constant behavior.

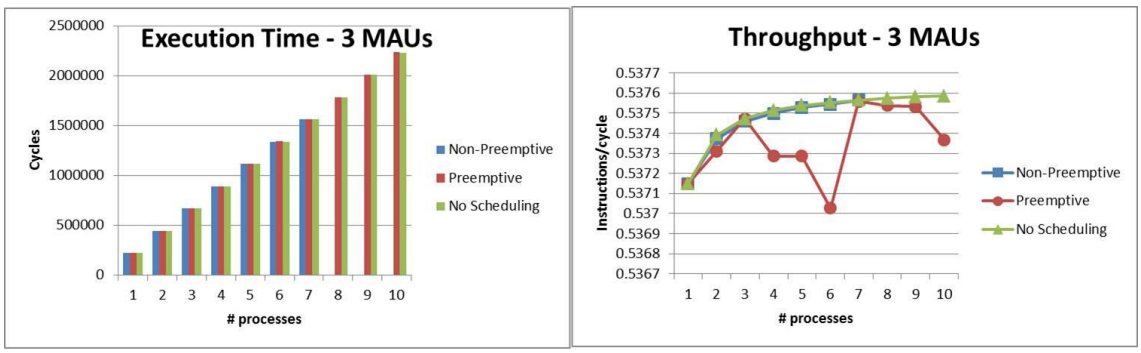


(a) (b)

Figure 12: Experiment 2 with 2 MAUs - (a) execution time; (b): throughput

Despite the difference between these two versions and preemptive scheduling is very small, we still see that the behavior of the last one remains inconstant (Fig. 12(b)).

Fig. 13 shows the experiment 2 to 3 MAUs. The execution time is almost equal between the non-preemptive and no scheduling versions, however in the non-preemptive version happens a deadlock when the number of processes is superior to 7 processes in each MAUs (Fig. 13(a)). The throughput (Fig. 13(b)) shows that the preemptive version doesn't have the same problem and has a performance a little bit inferior comparing the no scheduling version, mainly when the number of processes increases.



(a) (b)

Figure 13: Experiment 2 with 3 MAUs - (a) execution time; (b): throughput

Analyzing the second experiment, varying the amount MAUs, we see that the non-preemptive version has the best performance when the offered load on the network is small, however for very large loads can lead to a deadlock situation. The version without scheduling has the worst performance for small loads but overcomes the versions with scheduling when the load is large. And the version of preemptive scheduling has been proposed to employ justice between the processes and balance the network load. And so this version has a reasonable performance for both small and large loads.

6 Final Considerations

The paradigm break caused by IPNoSys puts processing and communication working together to build a packet-driven architecture with high parallel processing. This idea is the base to the platform that also offers a packet description language, an assembler and a simulator to allow the development and execution of general purposed applications to extract various simulation results.

The architecture transforms the NoC's routers in Routing and Processing Units (RPU) that is responsible to execute the major part of instructions inside the packets that flows through the network. Additionally, there are four Memory Access Units (MAU) in the network's corners, where is also placed the memory modules. And one of these MAUs also works establishing the communication between the processing system and I/O system that implements a Direct Memory Access (DMA) scheme, including MemArbiter and the IONode.

In this paper was presented the operating system support for IPNoSys, including the memory management, process management, I/O management, interruption, exception and timer. Additionally, it was proposed two versions of multi-task scheduling, the first one is a preemptive and the second a non-preemptive. Two experiments were performed to evaluate the scheduling algorithms, each one composed by 10 simulations, using 1 to 3 MAUs in parallel. The first experiment formed by only one packet that implements a loop and the second one formed by several parallel packets performing a loop and using synchronization between the parallel packets.

Through these experiments it was noticed the non-preemptive version has the best performance when the offered load on the network is small, however for very large loads can lead to a deadlock situation. The version without scheduling has the worst performance for small loads but overcomes the versions with scheduling when the load is large. And the version of preemptive scheduling has a reasonable performance for both small and large loads and doesn't present the deadlock.

References

[1] V. Borkar, "Platform 2015: Intel processor and platform evolution for the next decade," in *Intel Corporation white paper*, 2005, pp. 3-12.

[2] S. Kumar, A. Jantsch, J.-P. Soinienen, M. Forsell, and M. Millberg, "A Network on Chip Architecture and Design Methodology," in *Proceedings of the IEEE Computer Society Annual Symposium on VLSI*: IEEE Computer Society, 2002, p. 117.

[3] P. P. Pande, C. Grecu, A. Ivanov, and R. Saleh, "Destination network-on-chip," *EDA Tech Forum Journal*, pp. 6-7, 2005.

[4] R. d. S. Cardozo, "Low Cost Network-on-chip," in *Instituto de Informática*. vol. Master Porto Alegre: Federal University of Rio Grande do Sul, 2005, p. 76.

[5] G. Girão, B. C. Oliveira, R. Soares, and I. S. Silva, "Cache Coherency Communication Cost In A Noc-Based Mp-Soc Platform," in *20th Symposium on Integrated Circuits and Systems Design*, Rio de Janeiro, 2007, pp. 288-293.

[6] S. R. F. d. Araújo, "The study of viability of no processor network-on-chip based integrated system development: IPNoSys system," in *Informatics and Applied Mathematics Department*. vol. Master Natal: Federal University of Rio Grande do Norte, 2008, p. 76.

[7] S. Fernandes, B. C. Oliveira, and I. S. Silva, "Using NoC routers as processing elements," in *Proceedings of the 22nd Annual Symposium on Integrated Circuits and System Design: Chip on the Dunes* Natal, Brazil: ACM, 2009.

[8] S. R. Fernandes, I. S. Silva, and M. Kreutz, "Packet-driven General Purpose Instruction Execution on Communication-based Architectures," *JICS - Journal of Integrated Circuits and Systems*, vol. 5, p. 14, 2010.

[9] S. R. Fernandes, I. S. Silva, and R. Veras, "I/O Management and Task Scheduler on Packet-Drive General Purpose Architecture," in *XXXVII Conferencia Latinoamericana de Informática (CLEI 2011)*, Quito, 2011, pp. 1284-1295.

[10] B. R. Preiss and V. C. Hamacher, "Data Flow on Queue Machines," in *12th Int. IEEE Symposium on Computer Architecture*, 1985, pp. 342-351.

[11] H. Schmit, B. Levine, and B. Ylvisaker, "Queue Machines: Hardware Compilation in Hardware," in *Proceedings of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 2002, pp. 152-160.

[12] A. Canedo, B. Abderazek, and M. Sowa, "Queue Register File Optimization Algorithm for QueueCore Processor," in *19th International Symposium on Computer Architecture and High Performance Computing*, 2007.

[13] B. A. Abderazek, T. Yoshinaga, and M. Sowa, "High-Level Modeling and FPGA Prototyping of Produced Order Parallel Queue Processor Core," *The Journal of Supercomputing*, vol. 38, pp. 3-15, 2006.

[14] M. Sowa, B. A. Abderazek, and T. Yoshinaga, "Parallel Queue Processor Architecture Based on Produced Order Computation Model." vol. 32: Kluwer Academic Publishers, 2005, pp. 217-229.

[15] H. T. L. Nguyen, "Network-on-chip dataflow architecture," U. S. p. a. t. office, Ed. United States: Hanoi Tran Le Nguyen (VN), 2007, p. 9.

[16] H. T. L. Nguyen, "Network-on-chip dataflow architecture," U. S. p. a. t. office, Ed. United States: Hanoi Tran Le Nguyen (VN), 2009, p. 8.

[17] G. Sassatelli, L. Torres, P. Benoit, T. Gil, C. Diou, G. Cambon, and J. Galy, "Highly scalable dynamically reconfigurable systolic ring-architecture for DSP applications," in *Design, Automation and Test in Europe Conference and Exhibition, 2002. Proceedings*, 2002, pp. 553-558.

[18] R. M. Hord, *Parallel supercomputing in MIMD architectures*: CRC Press, Inc., 1993.

[19] W. Wolf, A. A. Jerraya, and G. Martin, "Multiprocessor System-on-Chip (MPSoC) Technology," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 27, pp. 1701-1713, 2008.

[20] M. Pelleh, "A Study of Real Time Scheduling for Multiprocessor Systems," in *Electrical and Electronics Engineers in Israel, 2006 IEEE 24th Convention of*, 2006, pp. 295-299.

[21] P. B. Sousa, B. Andersson, and E. Tovar, "Implementing slot-based task-splitting multiprocessor scheduling," in *Industrial Embedded Systems (SIES), 2011 6th IEEE International Symposium on*, 2011, pp. 256-265.

[22] T. C. Xu, A. W. Yin, P. Liljeberg, and H. Tenhunen, "Operating System Processor Scheduler Design for Future Chip Multiprocessor," *Architecture of Computing Systems (ARCS), 2010 23rd International Conference on*, pp. 1-7, 2010.

[23] B. Kakunoori and S. C. Madathil, "Hardware support for dynamic scheduling in multiprocessor Operating System," in *Intelligent Solutions in Embedded Systems (WISES), 2010 8th Workshop on*, 2010, pp. 108-113.

[24] J. L. E. Silva and J. J. Lopes, "A dynamic dataflow architecture using partial reconfigurable hardware as an option for multiple cores," *W. Trans. on Comp.*, vol. 9, pp. 429-444, 2010.

[25] S. Fernandes, B. C. Oliveira, M. Costa, and I. S. Silva, "Processing while routing: a network-on-chip-based parallel system," *IET Computers & Digital Techniques*, vol. 3, pp. 525-538, 2009.

[26] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: the Hardware/Software Interface*, 3rd ed. San Francisco: Elsevier, 2005.