Processing while routing: a network-on-chip-based parallel system

S.R. Fernandes1 B.C. Oliveira2 M. Costa2 I.S. Silva2
1Department of Exact Science and Natural, Rural Federal University of the Semi-Arid, BR 110, Km 47, Bairro Presidente Costa e Silva, CEP 59.625-900 Mossoró, RN, Brazil
2Department of Informatics and Applied Mathematics, Federal University of Rio Grande do Norte, PO Box 1524, Campus Universitário Lagoa Nova CEP 59072-970 Natal, RN, Brazil
E-mail: silvio@ufersa.edu.br

Abstract: Technology integration has increased to the point where the development of multi-core processor architectures is a market reality nowadays. In this scenario, the interconnection network has a critical function when the number of cores increases, since it is impossible to use bus-based solutions. Other interconnection solutions have been employed. However, they are area and power expensive. This paper approaches this problem with a new NoC-based architecture and a new computation mode. It proposes the utilisation of network-on-chip not only as interconnection but also as the processing datapath.

1 Introduction

Nowadays, multi-core processors, dual-, quad- and eighth core are a market reality and probably, in the next generation of computing systems, we will see many-core architectures [1]. It will be possible not only because we are actually in the billion transistors era but also because we have been pushed up by design of on-chip interconnection networks.

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. On the other hand, more powerful interconnections, such as network-on-chip (NoC), are largely studied in a number of scientific papers. This communication architecture comes at the price for more area overhead because of the routers needed for messages sending. In this reality, the NoC becomes the interconnection network that, at the same time, symbolises a solution and a problem.

As a solution, NoC approach to system design answer with scalability, parallel communication, reusable structure and point-to-point interconnection. As a problem, NoC requires more chip area and more power.

The most traditional NoC architecture consists of a set of directed connected routers with a local port dedicated to the source and destinations for messages flows. In this design and in interconnection networks, in general, the routers are responsible only for the data transmission.

This paper proposes original NoC-based system architecture and a computational model, the IPNoSys system, where the routers are also responsible for the execution of operations, besides the routing process.

The following section presents the related works including the concepts of NoC and queue machines, which are technical premises to IPNoSys. Section 3 presents the IPNoSys architecture and its main concepts. Section 4 presents the simulation scenarios and results as well as a comparison between IPNoSys and a cycle accurate virtual platform. Section 5 presents the conclusions and future works.

2 Related works

In the literature it is possible to find lots of papers concerning the design of NoC and bringing a number of technical contributions. However, such contributions are related only to interconnection mechanisms, none of them are able to incorporate the execution of application instruction as the
packet flows. The NoC has been used as interconnection subsystem, never as the main architecture able to perform computations.

The interconnection mechanisms are fundamental in the performance and scalability of the systems, mainly in the system-on-chip (SoC) and multiprocessor-SoC (MP-SoC). The buses have been used since the beginning of computer design and now, concerning the design of multi-core systems, they are an inadequate and old solution. Furthermore, they are constantly identified as the main responsible for the ‘Von Neumann bottleneck’ [2] and also exhibit limited scalability. The bus hierarchy was proposed to minimise the scalability problem but it is an ad hoc solution, thus, it is not reusable.

Therefore the NoC has emerged as the most adequate interconnection mechanism to integrate systems that demand heavy processing and data flow, since it 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 [3]. The communication performed in NoCs uses messages encapsulated as packets. For the transmission, the packets flow from the source to destination through a path of neighbouring routers in a parallel and a pipelined way [4]. The main disadvantages of the NoC design are chip area and power dissipation. Other problems such as data coherency and consistence are found when NoC is used to minimise the scalability problem but it is an ad hoc solution, thus, it is not reusable.

Regarding the queue machine scheme it is possible to propose a packet construction strategy where operation and operands are placed in a queue fashion.

To take advantage of this fact, one can design an NoC-based architecture where the routers are able to incorporate the execution of application instructions, together with routing tasks. This will allow reducing significantly the cores needed to process concurrent tasks in the system. Reusing the area (originally) dedicated only to data communication to accomplish also instruction execution can save large area when an entire system is considered. This paper does not analyse precisely the area and power dissipation of the proposed architecture. For now such subjects are out of scope. The focus of this paper is the presentation of the architecture features and computation mode, as well as the performance comparison with MP-SoC architecture.

In particular, the queue machines [10] present a computing model with some similarity to this proposal, but they use a queue processor instead of NoC for 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 because of 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 because of their compact instruction set, high instruction level parallelism (ILP) and simple hardware that reduces the chip area and power dissipation. Thus, several computing models have been proposed for queue machine [11, 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].

Regarding the queue machine scheme it is possible to propose a packet construction strategy where operation and operands are placed in a queue fashion.

This proposal suggests that the execution of application could be done by dataflow architecture. A recent patent [15] proposed a NoC dataflow architecture where part of execution of the applications is done by the 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 not within a loop. For the other case, the processor configures the networks and distributes the data through a bus for execution in the networks. The author affirms that his architecture is powerful only when there are instructions within loop in the applications. Furthermore, it is noticed that the scalability is limited by the bus, the dataflow graph (DFG) that determines the ALU's configuration is limited by NoC size and the instruction network depends on processor's program counter (PC) to fetch the instructions.
Architecturally, the iWarp chip [16, 17] is similar to the one proposed in this paper. iWarp is a product of a joint effort between Carnegie Mellon University and Intel Corporation, whose goal is to develop a powerful custom VLSI single-chip processor for various distributed memory parallel computing systems. 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 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, although the iWarp uses static scheduling to ensure the synchronism of the computation between the processors. Therefore a pre-configuration based on application features is necessary before executing it.

3 IPNoSys architecture

Considering the hypotheses in the previous section, we developed an original NoC-based architecture called integrated processing NoC system (IPNoSys). In this architecture, the NoC is not only an interconnection mechanism but also becomes an active element in the execution of applications. In IPNoSys, the NoC is a direct square 2D-mesh and uses the following features: XY routing policy, a combination of virtual-cut-through (VCT) and wormhole switching scheme, virtual channel, credit-based control flow, distributed arbitration and input buffering.

IPNoSys includes, in the router data path design, an arithmetic logic unit (ALU) allowing the router to perform the most common logic-arithmetic operations usually found in applications. Therefore a router in this architecture accomplishes both tasks: routing packets and processing, being called routing and processing unit (RPU). In addition to one ALU, the RPU has a synchronisation unit (SU) that allows performing synchronisation instructions among RPUs. The memory strategy used was 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 cores (MACs), which are placed also in the network’s corners. Fig. 1 shows a 4 × 4 IPNoSys architecture instance.

In a traditional direct NoC-based architecture, each router is linked to a processor or other core. However, the IPNoSys system does not use a PC nor fetches instructions nor keeps only one execution flow. Therefore Von Neumann processors are not necessary in such system, since the applications are performed in the RPUs where the packets pass through. The processors or cores are replaced by four MACs on the network’s corner.

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. 2), is sent to the next RPU in the path where the execution procedure begins again for the next instruction. 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 remain the same. 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 as the removed words. However, on average, the number of removed words is higher than inserted words and, therefore, the packet size decreases. To demonstrate the potential of size reduction of an application packet we can use

\[
\text{Growth ratio} = \frac{(D_i + D_o)}{(I_o + D_o)}
\]

where \(D_i\) is the number of inserted data as a result of the executed instructions in the packet, \(D_o\) is the number of inserted data by load instructions, \(I_o\) is the number of original instructions in a packet in the memory and \(D_o\) is the number of original data in a packet in the memory.

In the equation above \((D_i + D_o)\) represents the inserted words in the application packet and \((I_o + D_o)\) represents the original words that are there in the packet before it is executed. If the equation returns 1, the execution of the application inserts words at the same ratio it removes them. If the equation results in a greater or lower number then the application packet, respectively, increases or decreases.

Additionally, to avoid deadlocks, LOAD, STORE and synchronisation instructions flow through a dedicated virtual channel. It means that results that need to be stored in the memory are not inserted into application packet and routed from the point where they are generated to the memory. Instead of that, a STORE instruction (followed by a data and an address) in the application packet, when executed by the RPU, will generate a STORE packet that flows to the memory through the dedicated virtual channel. LOAD instructions are there in the application packet followed by an address. This instruction, when executed by the RPU, will generate a LOAD packet that flows to the memory and a LOAD response packet that flows from the memory to the RPU. The loaded data will be inserted in a specific position in the packet as an operand word. This means that these data will be removed from the packet as well as original data in the packet. Synchronisation instructions are used to control the flow of packets, especially the injection of packets in a parallel execution.

In the IPNoSys routing scheme is not a way to carry a packet from a source to a destination, but a way to provide a sequence of RPU in the routing path, aiming to execute all instructions in the packet. In this path, each RPU executes the first instruction of the packet when it comes from a neighbouring RPU through an input port and removes it from the packet. Therefore the packet destination must be sufficiently distant from the source to execute all instructions of the application. Since the proposed architecture is a 2D-mesh, the longest path encloses less than two times the number of RPU in a row. This path can be still insufficient to the execution of all instructions in a packet. Thus, to provide sufficient resources in the routing path to the execution of all instructions in the packet, the spiral complement routing algorithm was developed to provide sufficient resources in the routing path to the execution of all instructions in the packet. This is possible because the algorithm finds a new destination to the packet when it arrives at its destination and instructions remain to be executed.

The bases to the proposed algorithm are the XY routing policy and the complement traffic pattern. The packets are injected into the system through the MACs in the corners. Thus, the first destination of the packets is the source’s complement, the longest router in the square NoC. If this path is not enough to perform all the instructions a new destination must be designed. This is performed at the end of the first routing path and the packet is injected once again. The new selected destination must provide once again the largest number of execution resources, providing the longest path. However, to find a new destination different from the original packet source, the algorithm virtually reduces the NoC size by one row or one column. Once again the longest destination is the opposite corner of the reduced NoC. Thus, the algorithm sends out the packet through a spiral path, as shown in Fig. 3.

Fig. 3 shows the packet being initially injected in the superior and left corner. In this case the end of the spiral is the inferior and left corner. When a packet passes through a completed first spiral and it still has instruction to be performed, a second spiral begins in the inferior and left corner ending in the inferior and right corner. The execution could continue by a third and fourth spiral beginning, respectively, at the inferior and right corner and superior and right corner.

The spiral complement 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

![Figure 3 Spiral complement path](image)

**Figure 3** Spiral complement path
physical channels of the network. Notice that each spiral concentrates the data traffic in a different corner, avoiding the centre of NoC. As a perilous side effect, depending on the packet size and the network dimensions, the circular movement of the packet might cause a deadlock.

3.1.1 Deadlock treatment: Suppose that a packet is being injected through the corner with coordinate 0,0 (Fig. 3), and this packet is so long that after it was forwarded by the first 16 RPU of the first spiral, it reaches the RPU with coordinate 3,2, which keeps transmitting the remainder of the packet from RPUs 3,3 to 3,1. The header of the packet arrives in the RPU 3,2 through the RPU 2,2 and it should be routed to RPU 3,1, according to the spiral complement algorithm. However, the header is not transmitted while the channel to stay allocated to the transmission of the remaining data of the packet. However, the remainder of the packet does not flow, waiting for the header, and the channel will be kept allocated, causing a deadlock.

Usually, the deadlock problem in NoC is solved through virtual channels. In this case, the number of virtual channels is the number of times that the packet should pass through the same physical channel in the same direction, in our case the maximum is three times (Fig. 3). However, if the packet was long enough to pass through four spirals and it comes back to the first spiral, the deadlock would happen again, since all virtual channels would be allocated.

Thus, the IPNoSys system treats the deadlock through a solution called local execution. Such solution consists of continuing performing the next instruction, in the RPU in charge of the header of the packet, until the output target channel becomes available to the header transmission.

3.2 Packet format

The IPNoSys packet corresponds to a variable set of 32-bit words. There are four types of words in a packet: header, instruction, operand and terminator. To identify each kind of word four control bits are used, of which only one is set at a time. Fig. 4 shows the packet format.

The header has three words. The first word has the current source and destination, the current number of instruction in the packet, three bits that determine how to calculate the next new destination and the 5-bit flag that determines the type of the packet. Currently, there are two types of packets: regular packets and control packets. The regular packet has the instructions performed in any RPU. Control packets have instructions that are performed only in the destination MAC. The second word has a single identifier for the packet. The third word is 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 result 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 (or address) of the words in the packet that are the destination location of the generated result. Some instructions use these two fields to indicate the coordinate of the MAC that will perform the instruction and to indicate the number of operands (more than two) necessary to execute it.

The operand word and the terminator word have only one field, with the operand and the ending pattern, respectively.
3.3 Routing and processing unit (RPU)

The RPU is a router with capacity to perform logic-arithmetic operations and branch instructions. As the architecture used a 2D-mesh topology, the central RPUs have four ports and the side RPUs and corner RPUs have three ports to communicate. The corner RPU, beyond the two ports to link it to adjacent routers, has a third port (local port) to link it to the MAC. An RPU with four ports is presented in Fig. 5, which also shows the ALU and the SU into the RPU. The corner RPU is shown in Fig. 6.

When a packet arrives in a RPU’s input buffer, the type of the packet is checked. If it is a control packet then it is only routed, using the traditional XY routing, and transmitted through an exclusive virtual channel. If it is a regular packet, before routing, it is verified if this RPU is the packet destination to calculate a new destination, according to the spiral complement algorithm. Next, the packet is routed to a new destination through XY routing. When the RPU is not the destination, the packet is only routed using XY routing.

In the output port, an arbiter solves the conflicts between packets requesting this output as a traditional NoC’s arbiter. However, before the arbiter transmits a regular packet, it requests the execution of the first instruction in the packet. When the first instruction is a logic-arithmetic or branch instruction, the arbiter requests the ALU to perform it. If it is a MAC instruction (memory access or application synchronisation), the arbiter requests the SU, which will create a control packet to such MAC. In both cases, the arbiter waits for the answer to remove the instruction and its respective operands and starts the packet transmission to the next RPU. The number of words removed from the packet is added to the pointer in the packet header. If the performed instruction returns a result, the arbiter maintains this result in a result buffer waiting the insertion point in the packet indicated in the instruction. Result buffers must include one result data (32 bits) and up to two result addresses (11 bits each).

When the RPU cannot transmit the packet header because of lack of buffer space in the receiver RPU or because of deadlock situation, the arbiter performs the local execution. Thus, it continues requesting the ALU or SU to perform the next instruction in the packet until the transmission becomes possible.

3.4 Memory access core

In the IPNoSys, an application can be described through one or more packets that are stored in memory. The MACs, placed in the corners, are responsible for reading the packets from memory and to injecting them into the NoC.
and also for reading and writing data from/to memory. This is done through the MAC’s instructions. The current MAC’s instruction set includes only six instructions: to read data (LOAD); to write data (STORE); to inject a packet immediately in an asynchronous way (EXEC); to inject a packet synchronously (SYNCEXEC), after the MAC receives one or more synchronisation signals; to send a synchronisation signal (SYNC); and to send a data to be put in another packet when it will be injected (SEND).

These instructions are responsible for establishing the communication or synchronisation between the processing modules (RPUs and/or MACs). Through these instructions, a result obtained in a node of the NoC can be sent to the memory where it can be stored or used (inserted as data operand) in another packet. In addition, as synchronisation signals, a control packet or a group of control packets can be used to start the application packet injection in the NoC, especially in parallel execution. When a RPU finds one of these instructions in a packet application, it removes this instruction from this packet, it generates a control packet with such instruction and sends it to the MAC that will execute it.

Control packets are sent by an exclusive virtual channel, which is deadlock free once the control packets are quite short and do not travel through circular paths. Beyond that, the virtual channel is used only to transmit control packets.

Therefore, in IPNoSys the processors are replaced by simple MACs, which inject packets in the NoC, performing memory access and synchronising the applications with control packets.

4 Results

The IPNoSys architecture was implemented in cycle-accurate SystemC [18]. IPNoSys is fully parametric and scalable, which makes it possible to create different instances of the architecture. In the accomplished experiments different NoC dimensions were used: $2 \times 2$, $3 \times 3$, $4 \times 4$, $5 \times 5$ and $6 \times 6$, but in all these cases the same configuration for RPUs was used. Each RPU has two buffers per input port, associated with the virtual channels. In the simulation, the buffers are first in first out (FIFO) with ten 36-bits words (32 data bits and four control bits, see Fig. 4). The number of ports depends on the position of the RPU in the 2D-mesh. Corner RPUs and side RPUs have three ports and central RPUs have four ports. The number of input ports is equal to output ports. In each output port there is an arbiter that, among other tasks, is responsible for storing the results of the instructions executed in the RPU in its result buffer. Buffers with twenty 54-bit words (one 32-bit data and two 11-bit result address) were used in the result buffer arbiters. In the worst case for a $4 \times 4$ NoC, a packet has 256 instructions that insert their results after the half of the all words in the packet. As in this NoC size a local execution starts in the 16th instruction, a result buffer of at most 121 words is necessary. In order to solve the conflicts for the physical channel the arbiter uses the round-robin policy. The RPU uses a crossbar switch partially connected, once one packet cannot be routed to the same input port. The ALU in each RPU can execute ten logic or arithmetic operations with 32-bit operands.

To simulate applications execution in this architecture, a packet description language was also developed. The language defines macros to specify the values of packets fields.

To describe a high level application to the packet description language, a DFG is used as an intermediate language. Such graph determines the data dependencies. Using the dataflow it is possible to see how many packets an application needs to execute and also it is possible to understand the synchronisation relationship between the packets. This information is useful not only for the accuracy of the execution, but also to formulate an execution approach aiming at better performance.

To validate and evaluate the IPNoSys architecture, three simulation cases are performed. The first one is a simple benchmark (simple counter) aiming to evaluate the local execution performance based on two simulation scenarios, the sequential execution and the parallel execution. The second simulation case was the execution of a domain transformation largely used in embedded applications, the two-dimensional discrete cosine transform (2D-DCT). It was executed in IPNoSys instances with different dimensions and it was also executed using different parallel strategies to compare the IPNoSys with a virtual platform. The third simulation case was a decoding algorithm [run length encoding (RLE)] used to evaluate the performance of the IPNoSys and the comparative virtual platform for an application where output data are larger than input data.

Additionally, a detailed comparison between the IPNoSys and the virtual platform is shown in Section 4.4, which presents simulation results to analyse the performance of both architectures.

4.1 Simple counter

The simple counter is a set of successive unitary add-accumulate operations. The final result was stored in the memory. In general, each add-accumulate was performed in a single RPU whereas in deadlock situations this can be different. Then, in order to evaluate the performance of the local execution in deadlock situations, the simple counter was performed with a variable number of add-accumulate instructions in the packet. Packets with 8, 16, 32, 64, 128 and 256 add-accumulate instructions were generated and a sequential and a parallel execution were performed. In the sequential execution, a single packet includes all the instructions and the packet was injected in the upper and left corner. In the parallel execution, four packets injected at the same time in the four corners perform the counter. In all these cases $4 \times 4$ IPNoSys architecture was used.
Fig. 7 shows the amount of instruction bytes (including instructions and operands) performed in each RPU for the six sequential executions. In all cases, the MAC placed in the superior and left corner injected the packet. Thus, because of the network dimensions ($4 \times 4$), the deadlock happens in the counter with more than 32 instructions in the RPU 3,2 as it is shown in Fig. 7. Such figure shows that local execution solved efficiently the deadlock problem; however a larger buffer in the arbiter might be necessary to temporarily store the instruction results.

The same execution is performed through four parallel packets. The simulation results are presented in Fig. 8. Notice the distribution of the execution load between the RPUs.

Consequently, when the packet size increases, the number of instructions performed in each RPU also increases. This figure also shows that as a result of the spiral complement algorithm, the network’s corners receive more packets traffic, avoiding concentration on the middle of the network, since this region is the linking way of the complementary RPUs. Thus, as the application was described in four parallel packets, the highest amount of instruction bytes is performed in the corners RPUs (0,0; 0,3; 3,0; 3,3).

Fig. 9 shows the throughput (in bit/cycle) in each link between the RPUs for the parallel execution in the counter with 256 instructions. In this particular situation, it is noticed that the spiral complement algorithm avoided the inconvenient bottleneck in the central links, and distributed the data flow to the network’s borders, where there are the highest throughputs.

These results also demonstrate the capacity of parallel processing in the IPNoSys system, which allowed to reduce the maximum number of performed instructions around 80% comparing the sequential and parallel execution, and furthermore to accelerate the execution time as can be seen in Fig. 10.

4.2 DCT

The 2D-DCT is largely used in compression process of images. We implemented a DCT using the separability property [19] in $8 \times 8$ pixels blocks of the image. It was used to show the execution capability of the IPNoSys system with different network’s dimensions and to compare its performance with a virtual platform. In all simulations, images in subQCIF format ($128 \times 96$) were used.

As it was mentioned before, the spiral complement algorithm and IPNoSys architecture allow that all the instructions in an application are performed independent of the NoC dimensions. To evaluate this property we
executed a sequential implementation of the DCT in five IPNoSys instances, with the following network's dimensions: $2 \times 2$, $3 \times 3$, $4 \times 4$, $5 \times 5$ and $6 \times 6$. Two aspects were considered in this evaluation: execution time and the maximum number of solved instructions, which are shown, respectively, in Figs. 11 and 12.

The execution time increases as IPNoSys dimensions increases and practically stagnates at $4 \times 4$ dimension. This behaviour is related with quantity of RPU that store part of the packet, execute at least one instruction and transmit the remaining packets.

The fastest execution happened in the smallest IPNoSys instance (see Fig. 11), since in this instance ($2 \times 2$) there are few links and little communication. In other words, when the NoC is smaller the number of instructions performed per RPU is larger (Fig. 12). However the smaller NoCs are more restrictive concerning the application parallelisation because they are more susceptible to the size of packets. This eliminates the possibility to have a single RPU as a parallel architecture.

The DCT application was also evaluated considering the parallelism in two levels of granularity: fine grained and coarse grained. In the fine-grained level, the instructions parallelism was considered and in the coarse-grained level the DCT process in $8 \times 8$ blocks was considered, in parallel.

The fine-grained DCT was described through three scenarios. In all scenarios, the $4 \times 4$ IPNoSys instance was used. The first scenario is the sequential execution where all the packets are injected through the same MAC. In the second scenario, the packets that control the execution loop of the application (looping packets) are injected in the same corner and the packets that perform the DCT calculus are injected in the nearest corner of the RPU where the loop packets finished. Finally, in the third scenario, a single RPU injects the loop packets (same corner injection) and the RPUs of the four corners inject the calculus packets (parallel injection).

To evaluate the performance of the three scenarios the DCT execution was also performed in a MP-SoC virtual platform – STORM [20]. STORM uses SPARC V8 processor, a direct 2D-mesh $4 \times 4$ NoC [21] and has a distributed memory and shared memory (with and without cache) versions.

In all cases a parallel and a sequential DCT implementation was performed, as in IPNoSys. To compare IPNoSys and STORM the observed results were the execution time and required memory.

Fig. 13 shows the required memory for each instance of STORM and the scenarios of IPNoSys. The required memory corresponds to total of global data, local data and code, in bytes. The IPNoSys first scenario (sequential execution) uses a little less memory than two other scenarios. As it was expected, these results are equivalent to STORM with distributed memory.

Comparing the execution time (Fig. 14), the IPNoSys best case is the first scenario that is slightly surpassed only by the STORM best case (parallel execution with cache read/write). The sequential execution in IPNoSys has less execution time than parallel because of the application characteristics and the parallelism level (ILP). The DCT application has much data dependencies, which is the worst case in ILP. Even so, the parallel execution (scenario 3) has short execution time. However, the IPNoSys does not use processors; therefore, the chip area can be substantially reduced since FPGA prototyping results show the MAC’s area is 84% smaller comparing with NIOS II/f processor.
The parallelism in ILP is impaired because of data dependency. Therefore the DCT was implemented in higher level granularity parallelism. As the DCT process consists of a series of calculations on 8 × 8 pixels blocks, in the coarse-grained DCT we parallelised the DCT block calculations. Thus, each parallel DCT block calculations were considered as an execution flow, which is started by a MAC. Therefore the coarse-grained DCT was also simulated in three scenarios. The first one is the sequential implementation where there was only one execution flow to calculate the DCT in all image’s blocks. In the second scenario, the DCT was parallelised through two execution flows, each one executed on half of the blocks. And in the third scenario the DCT was executed through four execution flows.

The three coarse-grained scenarios were also compared with the best case of STORM platform, considering the execution time and required memory. Fig. 15 shows that the required memory for IPNoSys is slightly increased with more parallelism because of the rise of the communication. It is also larger than STORM’s best case, which is a sequential implementation executed in cache read/write version. However, the efficiency of the parallelism in the IPNoSys system is shown in Fig. 16 that compares the execution time between the three IPNoSys scenarios and the STORM’s best case (parallel implementation executed in a cache read/write version). STORM’s best case is faster than sequential execution in IPNoSys (one flow). However, Fig. 16 shows that the execution through four flows in IPNoSys is 3.5 times faster than STORM’s best case, which confirms the potential of parallelism of the IPNoSys system.

4.3 RLE

Run-length encoding [22, 23] is a data compression algorithm that is supported by most image file formats, such as TIFF, BMP, PCX and JPEG also. RLE is suited for compressing any type of data regardless of its information content, but the content of the data will affect the compression ratio achieved by RLE. This algorithm is both easy to implement and quick to execute, making it a good alternative to either using a complex compression algorithm or leaving your image data uncompressed. RLE works by reducing the physical size of a repeating string of characters. This repeating string, called a run, is typically encoded into two bytes. The first byte represents the number of characters in the run and is called the run count.

For example, an uncompressed string formed by 15 ‘A’ characters would normally require 15 bytes to store: AAAAAAAAAAAAAA. The same string after RLE encoding would require only two bytes: 15A. In a JPEG compression the RLE algorithm can be useful as the first stage of the AC components encoding [23]. This fact occurs because of the previous stage of JPEG, the quantisation stage, which in general generates long sequences of zeros. Thus, the RLE algorithm can be used to encode how many zeros are there before another pixel value in the image.

The RLE decoding process is similarly easy to be implemented. However, an interesting feature of this application, as far as IPNoSys is concerned, is that the amount of processing output data is larger than input data.

This algorithm was implemented in a sequential and parallel way to be simulated in IPNoSys. Figs. 17 and 18 show the organisation and synchronisation between the packets in the respective cases. In Fig. 17 the ‘Packet 1’ calculates the load address and stores address that are sent, respectively, to ‘Packet 2’ and ‘Packet 3’. The ‘Packet 2’ loads each run (zero quantity and the other value) from the memory and sends it (ZQ and N) to ‘Packet 3’ that decodes it and stores the results in the memory. ‘Packet 2’ is executed as many times as quantity of run of the compressed image. This is controlled by a counter that is sent through send instruction, which updates it in each execution. When the counter reaches the quantity of run
the ‘Packet 2’ sends a sync signal that provokes the execution of the final packet (Packet 0).

Fig. 18 shows the RLE decompression algorithm parallel version. It is a coarse-grained implementation with four execution flows. Each flow decodes 1/4 of the compressed data using pair of the packets: 2 and 3, 4 and 5, 6 and 7 and 8 and 9. Each pair works as the ‘Packet 2’ and ‘Packet 3’ of the sequential version, that is load a run, send it to another packet to decode it and store it in the memory as partial results. In the end of execution of each flow, such packets send one sync signal for execution of the ‘Packet 10’. This packet is responsible for indicating to ‘Packet 11’, ‘Packet 12’ and ‘Packet 13’ the quantity of data and where the partial results of the second, third and fourth flows have been stored. In this way, ‘Packet 11’, ‘Packet 12’ and ‘Packet 13’ move the partial results to the sequential position in the memory where the first flow’s results are stored, and so, they constitute the final result. At the end of the execution of these three packets, synch signals are sent to execute the ‘Packet 0’, the final packet.
Although the quantity of stored data is larger than loaded data in the RLE decoding algorithm, the application packets decrease, on average, during the execution in IPNoSys. To demonstrate this fact, theoretically, (1) presented in Section 3 to calculate the growth ratio of the application packets will be used. The variables of this equation are calculated observing the application packets statically, independent of the number of times each packet will be injected.

Variable $D_i$ corresponds to the sum of all inserted data of all packets as results, which was 233 for RLE decoding. Variable $D_l$ is the sum of the inserted data by load, and in this case was 11. The algorithm was implemented using 281 instructions and 233 operands originally present in the packets before its injection for execution.

$$\text{Growth ratio} = \frac{(233 + 11)}{(281 + 233)} = 0.41$$

As the calculated growth ratio was a value lower than 1, the number of removed words is bigger than inserted words, which means the number of packets decrease, on average, at the end of its execution. The next section will present the difference between injected load and received load in the MACs, calculated dynamically by the simulator.

In order to analyse the performance of this application, the execution time in cycles is compared between IPNoSys and STORM, as shown in Fig. 19. A sequential and a parallel versions were computed. As the best performance of STORM uses cache read/write, different instances of this platform with cache were used, changing the number of processors and consequently the NoC dimensions. Fig. 19 shows that the best performance of the RLE algorithm in STORM happens in the instance with 15 processors; however, the execution time still is two times higher than the parallel execution in IPNoSys and 1.6 times higher than the IPNoSys sequential implementation. The indicators to the IPNoSys performance superiority is presented in the next section, through analysis of other measurements.

### 4.4 Detailed comparison (STORM × IPNoSys)

To have a detailed comparison between IPNoSys and STORM the RLE decoding algorithm was simulated on both architectures and internal results were collected. The goal of this section is to provide some numerical and simulation-based considerations that make it possible to the reader to understand the reasons for the better results obtained by IPNoSys.

Before the comparison it is necessary to show STORM's configuration and simulation minimal costs. All the simulations performed in STORM architectures have used instances with one, two, four or 15 SPARC V8 processors using a 2D-mesh NoC with two, three, five and 16 routers, respectively. The routers are very similar to those used in IPNoSys. STORM is a cache coherent directory-based MPSoC platform. It means that the memory module of STORM is coupled with a directory unit that performs cache coherency. Also STORM has atomic LOAD/STORE instruction, used to provide memory consistency. Based on the observed DCT benchmark results, the RLE decoding algorithm was simulated only on the shared memory STORM version, using a read only cache, a read/write cache and a version without cache. The cache used in the read/write or read only version has 512 bytes with blocks of 32 bytes. The cache and block size are justified in [5].

Considering the implementation of the directory solution for the cache consistency of STORM, it was presented in [5], there is a set of read/write minimal costs. These costs are shown in Table 1.

IPNoSys does not have similar costs and the coherency and consistency can be guaranteed using control packets in an appropriate way. These packets use a dedicated virtual channel following a deterministic XY routing scheme. So, transmitted packets to a same destination arrive always in the same order.

A direct comparison between STORM and IPNoSys concerning read/write time can be performed with read/write waiting cycles. In the results shown in Table 2.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Cost (cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read Hit</td>
<td>1</td>
</tr>
<tr>
<td>Read Miss Clean</td>
<td>36</td>
</tr>
<tr>
<td>Read Miss Dirty</td>
<td>61</td>
</tr>
<tr>
<td>Write Hit with Permission</td>
<td>1</td>
</tr>
<tr>
<td>Write Hit without Permission</td>
<td>27</td>
</tr>
<tr>
<td>Write Miss Clean</td>
<td>36</td>
</tr>
<tr>
<td>Write Miss Dirty</td>
<td>60</td>
</tr>
</tbody>
</table>

![Figure 19 Execution time for RLE](image)
it is possible to see that IPNoSys waits less than STORM (two and four processors) and waits more than STORM (one and 15 processors). For STORM 1 processor, the cache is exclusive and the shared memory is just one hop distant. STORM 15 processors use 15 tasks that work less than IPNoSys. IPNoSys is comparable in amount of work with STORM 4 processors. In this case IPNoSys waits 29.4% of the STORM waiting cycles, even though STORM uses a shorter NoC.

Table 2 shows also that concerning the number of executed instructions IPNoSys is better than all instances of STORM. Concerning the load injected in the NoC for the execution of the application (RLE decoding), IPNoSys is more costly in all cases. However, as theoretically demonstrated in the previous section, the packets (the injected load) reduce while routing. This theoretical demonstration can be verified in practice, that is the simulation shows that only 57.7% of the injected load arrives at a destination MAC. This number differs from that found with the utilisation of (1). Equation (1) considers only the injected and removed words. So control instructions like EXEC; SYNC; SEND; STORE or SYNCEXEC are included only in the words to be removed. However, these instructions, as LOAD, generate packets that arrive to a destination MAC, resulting in additional load that flows through the NoC without being removed. Despite the difference both the numbers confirm, theoretically and experimentally, that the packet size reduces while flowing in the IPNoSys architecture.

Finally, the simulation shows that execution of only the instruction in the head of the packet implies only a limited power overhead for the processing resources. The simulation shows that the number of hops per instruction, that is the number of RPU instructions that an instruction crosses without having been executed, is on average, 9.8, considering all the packets shown in Fig. 18. Considering only the largest packet, with 52 instructions, this number grows to 16.5 hops, on average. The last instruction of this packet flows through 32 RPU instructions before being executed. Another interesting information obtained from the simulation is the number of instructions executed by RPU instructions. For the RLE parallel implementation (Fig. 18), on average, each RPU has executed 27 647.1 instructions (the total number of instructions can be found in the #Executed instruction row of Table 2).

5 Conclusions

This paper presented an innovative NoC-based architecture that does not use traditional processors, denominated IPNoSys. Such architecture has simple MACs, only in the network’s corners, and the routers become RPU instructions. Applications are described in a packet format that include instructions which are performed when the packets flow from the source to the destination, using a new routing algorithm (spiral complement). In addition, the packets’ size decreases while routing, increasing the capacity to inject new packets.

IPNoSys also identifies and avoids deadlock situations. It adopts the local execution to solve that problem. This solution showed its efficiency through simulations. A largely used domain transformation application (DCT) also was simulated in different NoC dimensions to show the architecture’s execution capability independent of the number of application instructions and NoC dimensions.

The DCT was used to evaluate the performance of IPNoSys with the performance of a MP-SoC, cycle accurate, virtual platform, considering the parallelism in two granularity levels. This simulation showed that IPNoSys required as much memory as the STORM platform and short execution time despite different parallelism strategies. However, the execution time in the IPNoSys is 3.5 times smaller than the STORM best case that shows the efficiency of the parallelism in this system.

A decompression algorithm (RLE) was simulated in both architectures, where the output processing data are larger than input data. And the IPNoSys performance also was better than STORM. In order to explain the reasons for this performance, a detailed comparison between the architectures through the analysis of other simulation results was accomplished.

Future work includes utilisation of virtual channels to transmit regular packets, new routing algorithms, pre-configuration of RPU to inform how many instructions are
executed when the packet arrives at RPU, compiler development, FPGA prototyping and chip area and power dissipation evaluation.

6 References


