Packet-driven General Purpose Instruction Execution on Communication-based Architectures

Sílvio R. Fernandes[1], Ivan S. Silva[2] and Marcio Kreutz[3]

[1] Departamento de Ciências Exatas e Naturais, Universidade Federal Rural do Semiárido, Mossoró/RN, Brazil
[2] Departamento de Informática e Estatística, Universidade Federal do Piauí, Teresina/PI, Brazil
[3] Departamento de Informática e Matemática Aplicada, Universidade Federal do Rio Grande do Norte, Natal/RN, Brazil,
e-mail: silvio@ufersa.edu.br

ABSTRACT

In the last few years, the development of Multi-Core architectures was driven by the crescent advance in integration technology. In this scenario, when the number of cores increases, problems found on shared communication media, such as busses, can be addressed by using a network approach. This paper proposes the usage of communication capabilities of Networks-on-Chip (NoCs) to execute general purpose instructions. The main idea behind this approach is to allow networks-on-chip architectures to execute general purpose instructions inside each router architecture. This paper addresses the main architectural concerns involved on creating datapaths for routers as well as the programming model suggested to pack instructions on messages. Simulation results on two case studies illustrate the benefits of the proposed architecture when compared to an equivalent NoC-based MP-SoC.

Index Terms: System-on-Chip, Network-on-Chip, NoC, routing algorithm, IPNoSys.

1. INTRODUCTION

Multi-core processors are currently being established as a standard for general purpose processing due to energy consumption concerns. In this scenario, moving to more, simpler processors seems to be a natural trend for computers. Thus, one may not be surprised when many-cores architectures [1] will be establisehd as a paradigm in the near future. The design of multi-core architectures demands also considerable attention on communication infrastructure characteristics.

At a first attempt, one may consider busses to keep up with the communication demands. However, the bus-based design remains useful only to a certain (limited) number of cores [2]. When MP-SoCs come to the picture, busses can no longer support the increasing amount of communications demanded by (a constantly growing) number of cores [3].

By offering concurrent communication channels, networks-on-chip can be tuned to trade-off design constraints for distributed applications. However, due to replication of resources, area and power consumption may impact on the design [4] being thus, always subject to a careful analysis on design requirements. Even so, the source for the most part of area and power dissipation in MP-SoCs comes from the processing elements and not from the network's routers. Thus, is up to designers to fit processing and communication architectures implementations to comply with applications requirements. This paper focus on helping designers on such a task by proposing an original packet-driven communication architecture with the ability to execute instructions. The main idea here consists on setting processing capabilities inside the routers of a specific NoC architecture, called IPNoSys [5].

The traditional NoC design consists of a set of routers interconnected with its neighboring and having a local port dedicated to the connection with a computing element, a memory or other useful core in a computing system. In this design the routers are responsible only for the data transmission. In the IPNoSys approach processing elements are added to the routers.

However, it is important to notice that, newer architectures normally are not useful if one is not able to program it. Concerning this, the paper also proposes a programming model, where primitives are defined to allow the IPNoSys network to be programmed for general purpose applications.

The paper is organized as follows: in the next section related works are presented. Section 3 presents

the proposed architecture and its components. Section 4 presents the proposed programming model. Section 5 presents the simulation scenarios and results as well as a comparison between the proposed architecture and a cycle accurate virtual platform. Section 6 presents the conclusions and future works.

2. RELATED WORKS

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[6]. The communication performed in NoCs uses messages encapsulated in packets. For the transmission, the packets flow from the source to destination through a path of neighboring routers in a parallel and pipelined way[7]. The main disadvantages of the NoC design are chip area and power dissipation rising. 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 [8]. However, some viability experiments [9] 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.

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 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 for queue machine [13], [11], [14].

This computing model suggests that the execution of application could be done by dataflow architecture. A recent patent [15] proposed a 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. Otherwise, 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 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 in [16]. This architecture 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. The configuration layer uses the same principle as FPGAs, it's a RAM which contains the configuration of all components (Dnodes and interconnects). 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 has in fact two execution modes: global mode, which the Dnodes execute microinstruction code that comes from configuration layer, managed by the RISC configuration controller; and local mode, which each Dnode stores, in its registers, up to 8 microinstruction codes, that are executed each cycle, increasing also the value on the multiplexer address input the content of a register to the datapath part of the Dnode. 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 [17], [18] is similar to the one proposed in this paper. iWarp is a product of a joint effort between Carnegie Mellon University and Intel Corporation, which aims 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, focusing on systems ranging from few to thousands of processors. The communication can be implemented as message passing or systolic. The systolic model is similar to our proposal, though iWarp uses static scheduling to ensure synchronism among processors.

3. PROPOSED ARCHITECTURE

Networks-on-Chip enable parallel communication of packets between cores, with packets having temporarily being allocated to buffers in their path to

destination. 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 a MP-SoC. This is achieved in IPNoSys by removing regular processing elements and by replacing them by Arithmetic/Logical Units inside each router. This way, IPNoSys approach creates a new paradigm in NoCs usage on MP-SoC designs, since communication and computation can no longer be considered apart from each other during the design process.

In the IPNoSys (Integrated Processing NoC System) architecture [19], the NoC is not only viewed as an interconnection mechanism; instead, it becomes an element for instructions execution. IPNoSys is a direct square 2D-mesh 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. Figure 1 shows a 4x4 IPNoSys architecture instance.

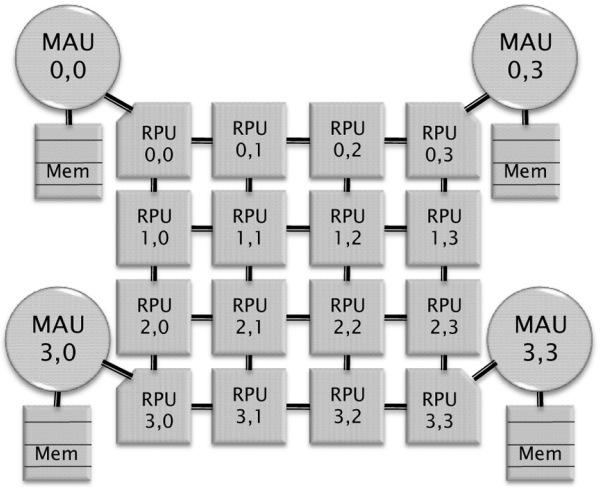

Figure 1. 4x4 IPNoSys architecture

In a traditional direct NoC-based architecture, each router is linked to a processor or other core. However, as in the IPNoSys the instructions are executed inside the RPUs where the packets pass through, the Von Neumann processors are no longer needed as in regular NoC-based model. Considering the whole chip, the area added to routers, due to the insertion of ALUs and SUs, is compensated with the absence of processing elements on local ports. There are only four MAUs in the network's corners.

The proposed execution mode of instructions considers a pipelined transmission for packets in the NoC. In each RPU, instructions on the head of the packet are executed and then removed, setting the next instruction in packet order as ready for execution. The generated result (if any result is produced) must be inserted in a specific position in the same packet. The remaining instructions on packet are sent to the next RPU in the path where the execution proceeds. As the packet size decreases during execution, the same happens to the network load. The packet size does not decrease all the time during instruction execution, since this is depended of the kind of instruction being executed. If the executed instruction insert more result words than it removes words, the packet increases. If the opposite occurs, the packet size decreases. Also it is possible that the packet size keep stable. This happens when the number of results words is the same of the removed ones. The equation (1) demonstrates the potential of size reduction.

$$GrowthRatio = \frac{(Di + Dl)}{(Io + Do)} \quad (1)$$

Where,
Di: number of inserted data as result of the executed instructions in the packet
Dl: number of inserted data by load instructions
Io: number of original instructions in a packet in the memory
Do: number of original data in a packet in the memory

If the equation results 1, during execution, the application inserts words at the same ratio then its removes. If the equation results in a number greater than 1, the packet increases, if the result is lower than 1, it 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. Routing in IPNoSys 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, called spiral complement routing algorithm, was developed.

A. Routing Algorithm

The routing algorithm is called spiral complement and was developed to provide sufficient resources in the routing path to allow the execution of all instructions on packets. The execution of instructions is enabled by the property of the spiral complement algorithm of finding a new destination to packets when they arrive at its destination and still have instructions to be executed.

The bases to the proposed algorithm are the XY routing policy and the complement traffic pattern. The packets are injected in the system through the MAUs in the corners. Thus, the first destination of the packets is the source's complement, the longest router in the square network-on-chip. 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 choice 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 reduces virtually the NoC size by one row or one column. Once again the longest destination is the opposite corner of the reduced network-on-chip. Thus, the algorithm sends out the packet through a spiral path, as showed in Figure 2.

Figure 2 shows the packet being injected initially in the upper left corner. Each arrow represents an intermediary destination calculated by the spiral complement 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 forth spiral beginning, respectively, at the lower right corner and the upper right corner, coming back to the begin of the first spiral.

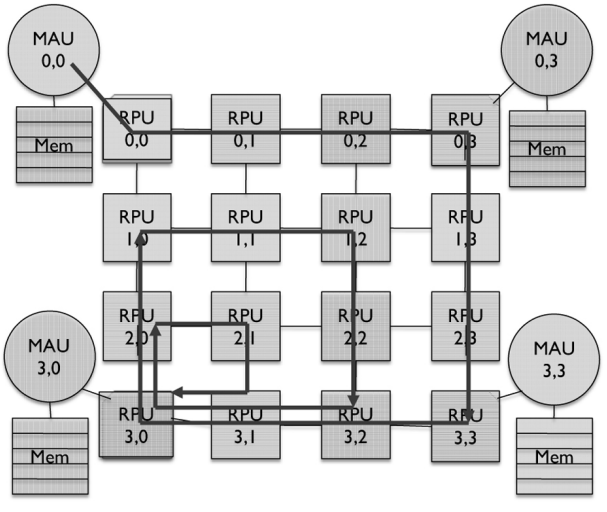

Figure 2. Spiral Complement path

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 physical channels of the network. Notice that each spiral concentrates the data traffic in a different corner, avoiding the center of NoC. As an inconvenient constraint, depending on the packet size and the network dimensions, the circular movements of the packet could cause a deadlock.

Usually, the deadlock problem in NoCs is solved through virtual channels. Unfortunately, in IPNoSys it is not possible to avoid deadlocks using only virtual channels, due to the circular nature of the spiral complement routing: when a packet arrives in the end of a spiral, it still may compete with newer packets being injected in the network. Thus, the IPNoSys system treats an imminent deadlock through a solution that was called local execution. An imminent deadlock 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 current RPU (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. As future work, we plan to combine local execution with virtual channels. Previous experiments [9] with spiral complement show that the number of time that a packet should pass through the same physical channel in the same direction, can be used to determine the number of virtual channels necessary.

B. Packet Format

The IPNoSys packet 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. Figure 3 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-bits flag that determines the type of the packet.

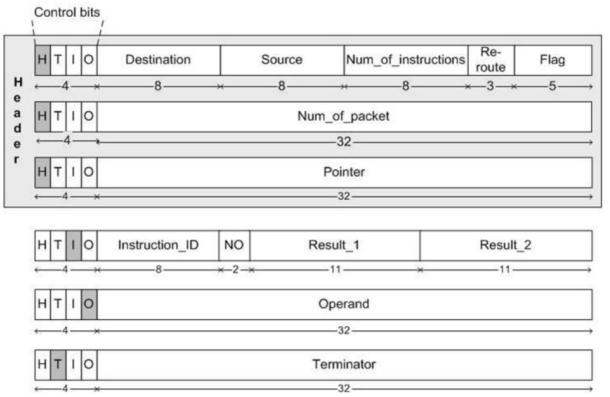

Figure 3. IPNoSys packet format

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 MAU. 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 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. Some instructions 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.

C. Routing and Processing Unit

The Routing and Processing Unit (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 communication ports (see Figure 1). The corner RPU, besides the two ports to link it to adjacent routers, has a third port (local port) to link it to the MAU (Figure 4).

When a packet arrives in a RPU 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, it is first verified if the current RPU concerns for the packet destination. This information is used to calculate a new destination, according to the spiral complement algorithm. Next,

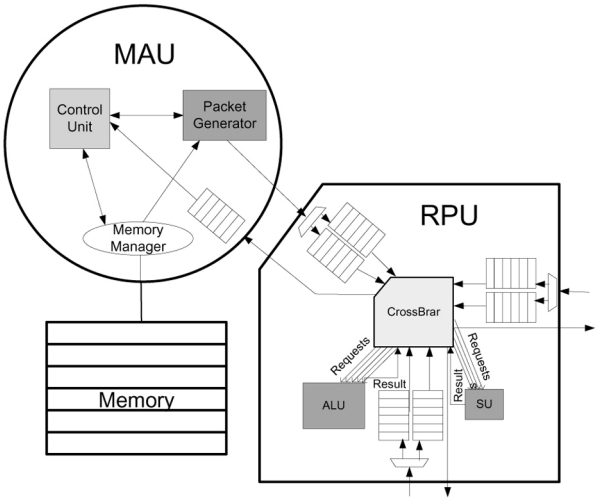

Figure 4. RPU and MAU architectures

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 each output port, there is an arbiter that solves the conflicts between packets requesting its output as a traditional arbiter for NoCs. 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 MAU's instruction (memory access or application synchronization), the arbiter requests the Synchronization Unit (SU), which will create a control packet to such MAU. In both cases, the arbiter waits 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 due to lack of buffer space in the receiver RPU or due to 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.

D. Memory Access Unit

In the IPNoSys, an application can be described through one or more packets that are stored in the memory. The Memory Access Units (MAUs), placed in the corners, are responsible to read the packets from memory and to inject them in the NoC, and also to read and write data from/to memory used as operand of instructions. This is done through the

MAU's instructions or control instructions (see section 4). These instructions are responsible to establish the communication or synchronization between the processing modules (RPUs and/or MAUs) or share computation results among packets. 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. When a RPU identifies one of these instructions, it creates a control packet with the instruction and sends it to the specific MAU which 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 MAUs placed in the corners, which inject packets in the NoC, performing memory access and synchronizing the applications with control packets.

4. PROGRAMMING MODEL

A. Instructions set

The IPNoSys' instructions set correspond to 26 instructions, where 4 of them are arithmetic, 4 logics, 2 shifts, 7 to memory access/synchronization, 7 conditionals and 2 auxiliaries, as showed in Table I.

All instructions are executed by RPUs, except the memory access and synchronization instructions that are sent through control packets to a MAU, where they are executed. That's the reason such instructions are called control instructions and the other ones, regular. The only access memory instruction executed in RPU is the RELOAD, which carries the loaded value from memory to the RPU that identifies the previous LOAD instruction. When a RPU identifies a LOAD instruction in a packet, it creates a control packet with this LOAD and sends it to the specific MAU that will execute such instruction. At this moment, the execution of the packet is stopped until the result of LOAD came back through another control packet, as indicated by RELOAD instruction.

All instructions follow the format presented in Figure 3, however, some of them using the fields of the packet in a different way. Arithmetic and logic instructions use the fields as explained in section 3.

As the control instructions must be executed in a MAU, then the fields *Result_1* and *Result_2* (see Figure 3) are not used to indicate the position in the packet where the result have to be inserted. All control instructions (except RELOAD) use *Result_1* to indicate the MAU's address that has to execute such instruction. Only LOAD and SEND instructions use the *Result_2* field as well as the regular instructions.

Table I. Instructions set

Instruction	Type	Description
ADD	Arithme-tic	Add 2 integers
SUB	Arithme-tic	Subtract 2 integers
MUL	Arithme-tic	Multiply 2 integers
DIV	Arithme-tic	Divide 2 integers
NOT	Logic	Negation of 1 value
AND	Logic	Conjunction of 2 values
OR	Logic	Disjunction of 2 values
XOR	Logic	Exclusive-or of 2 values
RSHIFT	Shift	Shift n bits to right
LSHIFT	Shift	Shift n bits to left
LOAD	M. access	Load a word from memory
RELOAD	M. access	Return a loaded word to packet
STORE	M. access	Store a word in memory
EXEC	Synchro.	Order to inject a pac. immedi-ately
SYNEXEC	Synchro.	Inject a pac. after synchroniza-tion
SYNC	Synchro.	Synchronization signal
SEND	Synchro.	Send a value to other packet
BE	Condit.	Branch on equal
BNE	Condit.	Branch on not equal
BL	Condit.	Branch on less
BG	Condit.	Branch on greater
BLE	Condit.	Branch on less or equal
BGE	Condit.	Branch on greater or equal
JUMP	Condit.	Unconditional branch
COPY	Auxiliary	Copy a value to same packet
NOP	Auxiliary	No operation

LOAD uses *Result_2* to indicate the position in the same packet to insert the loaded value (see Figure 5(a)) brought from the memory by RELOAD. In this figure, it showed that LOAD needs only one operand, the memory address. The result of LOAD is brought back to RPU thought RELOAD, in the unique operand field, and instead of the MAU's address, the *Result_1* field indicates the RPU's address that waits for the loaded value (Figure 5(b)).

The SEND instruction, showed in Figure 5(c), uses *Result_2* to indicate the position in other packet, identified by the packet number in the first operand field, to insert the value carried in the second operand field. The value carried through SEND instruction is inserted in a packet that stay in the memory, before it is injected.

In IPNoSys, a same value can be stored in many memory positions though only one STORE instruction. Thus, the value to be stored corresponds to the first operand, followed by the n memory addresses, as showed in Figure 5(d). As the number of operands can be more than two operands, then the *NO* field is set with 3 and the exact value (quantity of address) is informed in the *Result_2* field.

When a MAU receives a EXEC instruction, it injects immediately the packet that stay in the memory, identified by the number of packet in the operand field. EXEC does not use the *Result_2* field (see Figure 5(e)).

However, the injection of a packet can be realized in a synchronized way through the SYNEXEC instruction (see Figure 5(f)). This instruction identifies the packet number to insert, in the first operand field, and stored it together with the number of the other

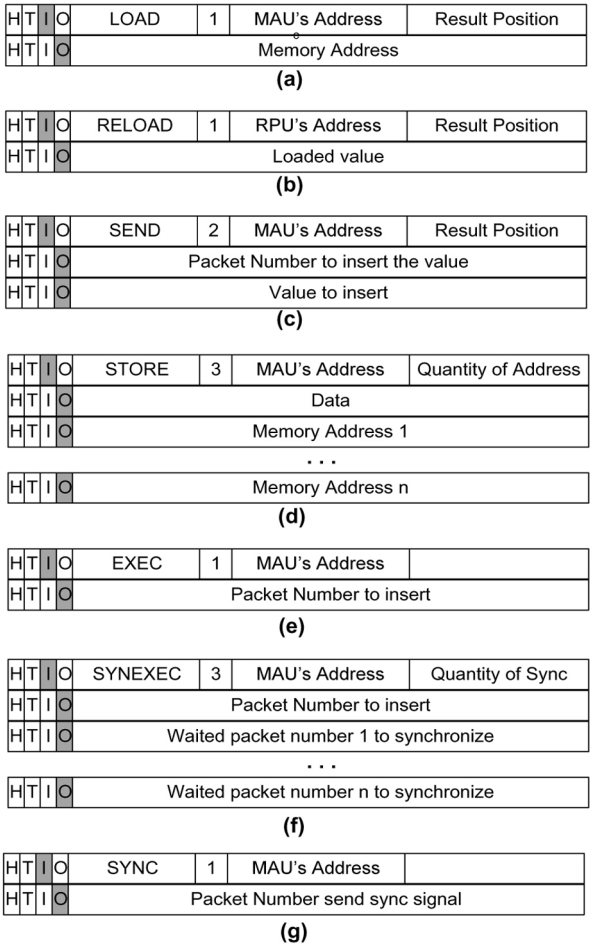

Figure 5. Control Instructions: **(a)** LOAD; **(b)** RELOAD; **(c)** SEND; **(d)** STORE; **(e)** EXEC; **(f)** SYNEXEC; **(g)** SYNC

packets (next operand fields) that MAU will wait synchronization signals. The injection of this packet only will be accomplished after all synchronization signals arrive to this MAU. However, since the MAU is not blocked, it continues executing other instructions when they arrive. As the STORE, the SYNEXEC set the *NO* to 3 and indicates the number of operands (quantity of synchronization signals) in the *Result_2* field.

Finally, when a RPU finds a SYNC instruction in a packet, it sends it to a specific MAU (identified in the *Result_1* field) with the number of this packet in the operand field. The *Result_2* field is not used, as showed in Figure 5(g).

The branch instructions use the fields as the original purpose, but there is a peculiarity. As branch instructions don't produce results, then the *Result_1* field is used to inform the position in the packet where the execution has to take place when the condition is satisfied. Figure 6 shows the format for branch instructions. This kind of instruction uses 2 operands that are compared; if the condition between them is true all words until the "position to jump" indicated in *Result_1* are discarded. Otherwise, the next instruction (after the branch instruction) is normally executed.

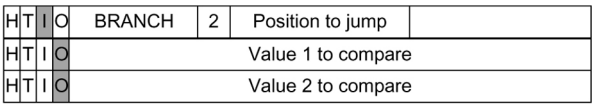

Figure 6. Branch instructions format

The instructions set has two auxiliary instructions: COPY and NOP. The first one is used to replicate a value that can be used as operand for other instructions. The COPY instruction can be replicated to two positions in the packet. The NOP instruction does not produce results or modify the packet.

B. Programmability

The IPNoSys' 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.

Figure 7(a) shows a simple example of arithmetic operations, where the third operation (multiplication) has data dependency with the previous two ones (addition and subtraction). Following the programming model, such instructions are piled up in the packet according to data dependency. For this example a packet as shown in Figure 7(b) is created.

Notice also, that each word of the packet is numbered, however some number are intentionally omitted. Such words are omitted since they represent operands that depend on the previous results. By the time a packet is injected, each RPU in the packet's path removes the first instruction in the packet, exe-

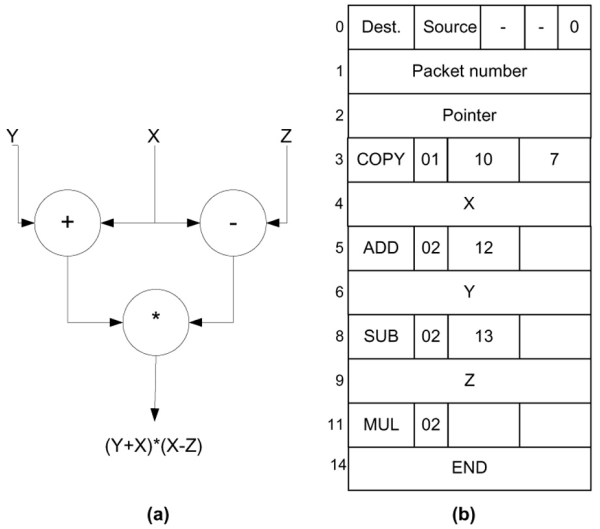

Figure 7. Simple Example: (a) data dependency; (b) correspondent packet

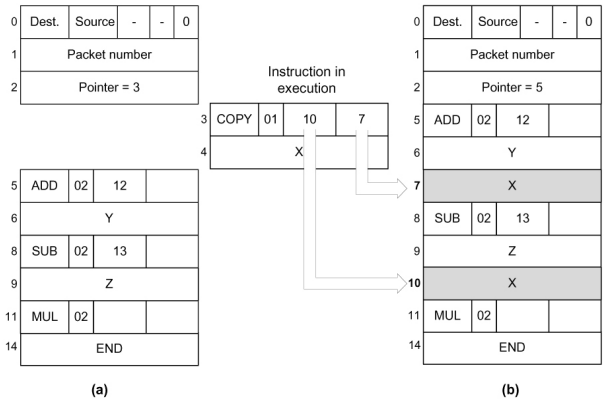

Figure 8. Execution of an instruction: **(a)** at the moment of execution; **(b)** after the execution with inserted words

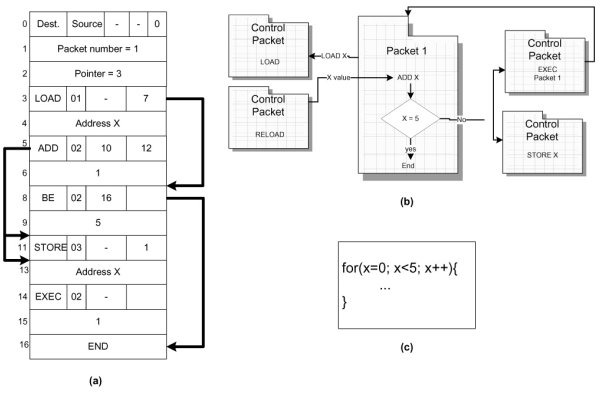

Figure 9. Simple example: **(a)** detailed packet; **(b)** relationship between the instructions; **(c)** equivalent code in C language

cutes it, update the pointer to the next instruction, transmits it to the next RPU and inserts the result in the positions informed by *Result_1* and *Result_2* fields in the packet structure (see Figure 3). Figure 8(a) shows the packet at the moment of the execution of an instruction. Figure 8(b) shows the packet after it executes the instruction, updates the pointer and inserts the result in words 7 and 10. The next instructions will be executed the same way in subsequent RPUs.

Despite the first instruction is always in the fourth word in the current packet, it is necessary keep the global count in the pointer word, since the insertion of results by RPU should consider the words that have not been included. During this count it is also necessary to know how many operands each instruction needs, since these operands could not be present yet. This is indicated by the number of operands" (NO) field on instruction.

The instructions set presented before are a tool to build executable programs to IPNoSys. In this section it will be presented the way to use the instructions set to create programs to IPNoSys.

First of all, the injected packets in the system are copies of packets that continue stored in the memory. Thus, it is possible to modify any packet's value in the memory and inject it again as a new instance. Secondly, IPNoSys does not use registers, so the variables in a program are places in memory that can be read or written. During the execution the value of variables and intermediate results are performed in the packets, as shown in Figure 8. Thirdly, the instructions are executed in the exact order they appear in the packet, which hinders a jump to a previous instruction to be executed (this is in fact, not allowed). However it is possible to execute loops and jumps as explained in the first property. Fourthly, a packet can be injected on demand, depending upon the number of EXEC or SYNEXEC instructions. Fifthly, a packet can be made self-run by pointing an EXEC/SYNEXEC to itself.

Figure 9 presents a simple loop example. The variable identified by "Address X" will be read from the memory (LOAD instruction), used in a counter (ADD instruction) and stored back to memory. The variable "Address X" will be updated until the value 5 is reached. The arrows in this figure represents where the instructions' results have be to inserted (words 7, 10 and 12) in Figure 9(a).

The LOAD instruction is sent through a control packet to a MAU which will execute it. The loaded value comes back through another control packet, which is incremented, as it is showed in Figure 9(b).

Immediately after the ADD instruction, the result of this addition is compared to 5. If two values are equals then, all words until the word 16 should be discarded, meaning the end of execution. However, if the compared values are different, then the incremented value would be stored at the same "Address X", through control packet with STORE (see Figure 9(b)), and the same packet (packet number 1) would be self-run, through control packet with EXEC, in order to perform a new increment and a new comparison.

This simple example represent a loop that increments by one any value loaded from memory until until value 5 is reached, which is equivalent to the code in C presented in Figure 9(c). The increment is performed by injecting the same packet (with updated value) several times until the stop condition is satisfied. The repetition is driven by the EXEC instruction which self-runs this packet.

C. Packet Description Language

Since the IPNoSys architecture has a new programming paradigm, there are some peculiarities in its assembly code necessary to implement it. This code, actually, is called Packet Description Language (PDL) that reflects exactly what was presented in previous sections.

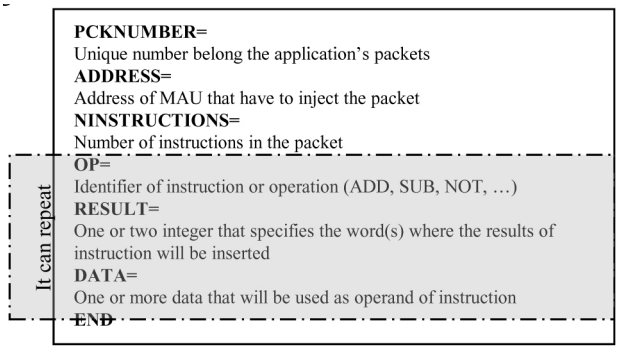

Figure 10. PDL's macros

```
PCKNUMBER= 1
ADDRESS= 0
NINSTRUCTIONS= 5
OP= LOAD
RESULT= 7
DATA= Address X
OP= ADD
RESULT= 10 12
DATA= 1
OP= BE
RESULT= 16
DATA= 5
OP= STORE
RESULT= 1
DATA= Address X
OP= EXEC
RESULT= 0
DATA= 1
END
```

Figure 11. Example using PDL

The PDL is set of macros that determine the value of each field in the packets. These macros are presented in Figure 10.

The three first macros are used to create the packet's header. The next three macros have the information about the instructions that describe the application, then after the macros NINSTRUCTOINS and before the END can appear several set of macros of instructions. Figure 11 presents the example of previous section described in PDL.

Currently, it is being developed a tool to translate an application written in the C language to PDL. This tool is the first step toward software compatibility efforts to allow IPNoSys to be programmed at higher levels. The idea is to generate PDL descriptions from control and data flow usually found in C programs.

The tool should than identify the statements in C and translates them to one or more packets according to control structure of the PDL as it was explained above. It is important to notice that this translation is straightforward in terms of operations, since the operations executed by IPNoSys are compatible with regular logic/arithmetic operations of instruction-set architectures (see Table I).

5. RESULTS

The IPNoSys architecture description was implemented in cycle-accurate SystemC [20]. IPNoSys is fully parametric and scalable in order to allow for different instances of the architecture with different characteristics. In the experiments in this text, it was used a 4x4 NoC with all RPUs having the same configuration. That is, each RPU has two buffers per input port, associated with the virtual channels. The buffers are FIFO (First In First Out) with ten 36-bits words (32 data bits and 4 control bits, see Figure 3). 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-bits data and two 11-bits result address) were used in the result buffer arbiters. 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, since one packet cannot be routed to the same input port. The ALU in each RPU can execute ten logic or arithmetic operations with 32-bits operands.

Regarding the chip area of the IPNoSys' router, it's not possible yet to generate an accurate information. However it is possible to compare it, for example, with RaSoC router of SoCiN Network on chip [2]. To do this, it is necessary to: (i) include an ALU like MIPS ALU; (ii) include some additional buffering for results registration before including them in the packet; (iii) change the control flow unit to work with credit, but not to all packets, only three words of packet header, one instruction word and two operands words; (iv) change the routing unit according to the spiral complement algorithm.

To validate and evaluate the IPNoSys' architecture and programmability, two simulation cases are performed. The first simulation case was the execution of a domain transformation largely used in embedded applications, the Two Dimensional Discrete Cosine Transform (2D-DCT). Different parallel strategies were executed to compare the IPNoSys with a virtual platform executing an application with several arithmetic instructions. The second simulation case was a decoding algorithm (RLE – Run Length Encoding) used to evaluate the performance of the IPNoSys and the comparative virtual platform for an application that performs many memory access and the number of data output is bigger than the data input.

Despite the maximum operating frequency and area evaluation between the architectures cannot be accomplished, since both of the architecture was implemented in SystemC, such implementations were done with cycle-accurate and similar critical paths.

A. DCT

The Two Dimensional Discrete Cosine Transform (2D-DCT – we will just call DCT) is largely used in compression process of images. We implemented a DCT using the separability property [21] in 8x8 pixels blocks of the image. This property consists in calculating 1D-DCT over the rows of an image's block, next to calculate 1D-DCT over the result block's columns of the previous 1D-DCT.

To evaluate the performance the DCT execution was also performed in a MP-SoC virtual platform – STORM [22]. Such platform uses SPARC V8 processor, a direct 2D-mesh 4x4 NoC [23] and it can use a distributed memory or shared memory (with and without cache) versions. In all cases it was performed a parallel and a sequential DCT implementation, as in IPNoSys.

The IPNoSys DCT implementation was performed in three scenarios, considering the parallelism. As the DCT process consists of a series of calculations on 8x8 pixels blocks, we parallelized the DCT block calculations. Thus, it was considered each parallel DCT block calculations as an execution flow, which is started by one MAU. The first scenario is the sequential implementation which there was only one execution flow to calculate the DCT in all image's blocks. The second scenario the DCT was paralleled through two execution flows, each one executed on half of the blocks. And the third scenario the DCT was executed through four execution flows. IPNoSys' flows can be parallelized due the injection of packets at the same time through different MAUs.

Figure 12 represents the relationship between the packets with only one execution flow (sequential implementation). The "Packet 1" is the starter that orders the injection of the "Packet 2" (through EXEC) and "Packet 0" (through SYNEXEC). The "Packet 2" keeps a counter to control the image's current block. This counter is incremented each time that "Packet 2" is injected and its value is updated in the memory (through SEND). When the "Packet 2" is injected, the counter's value is compared, if this value is equal to the number of block of the image, then it sends a synchronization signal (through SYNC) to allow injection of the "Packet 0". Otherwise, the number of the current block is informed to "Packet 3" (through SEND) and the injection order to "Packet 3" is performed (through EXEC). The "Packet 3" is responsible for deciding if it will run the 1D-DCT on the rows or on the columns. For each one the "Packet 3" calculates the memory address of the first element in the current block and informs to "Packet 4" (through SEND) and orders to inject such packet. The 1D-DCT is calculated in fact in the "Packet 4" on a block 8x8 representing the pixels (rows or columns). If it is the first 1D-DCT in the block the data are loaded

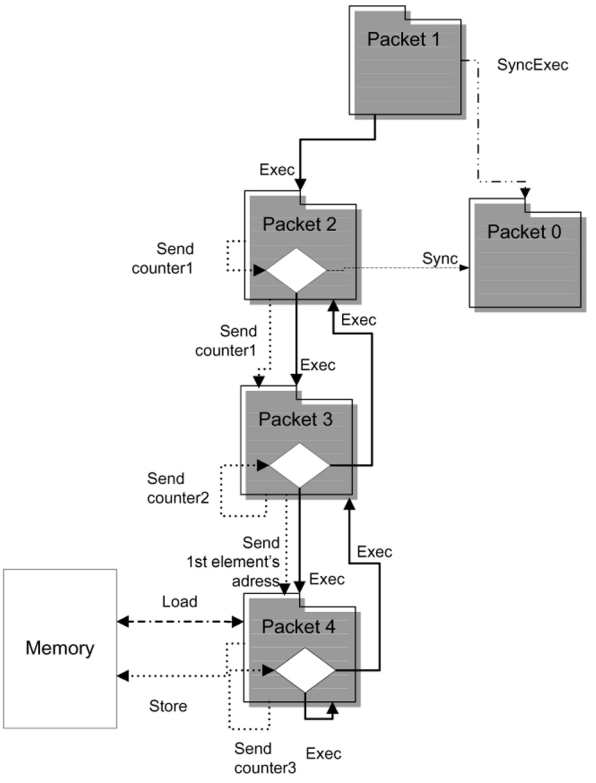

Figure 12. 2D-DCT implementation with one execution flow

from the address informed by "Packet 3" and the results are stored in the temporary memory addresses. If it is the second 1D-DCT the data are loaded from previous temporary memory addresses and the final results are stored in the specific memory address. The arithmetic in the "Packet 4" is performed on 8 pixels (one row or one column), therefore it keeps a counter that determines if such packet have to be self-run (through EXEC) until the counter achieves 8 (maximum of rows or columns). When the counter achieves 8, the execution returns to "Packet 3" which determines if it is necessary to calculate other 1D-DCT or to return to "Packet 2" which in turn determines if all image's blocks were already calculated, then sends the synchronization sign to "Packet 0". By definition, the "Packet 0" always is used to finalize the application execution. The unique instruction presents in this packet is the NOP.

In order to parallelize the DCT, the application also was implemented using two and four execution flows. For each execution flow it is necessary the repetition of the packets 2, 3 and 4, that can be injected simultaneously. Figure 13 shows the relationship between the packets in the implementation with four execution flows. Each execution flow is injected by one different MAU. The difference between the sequential and parallel implementations is that "Packet 1" orders the injection for more than one flow (packets 2, 5, 8 and 11) through a set of EXEC instructions, and the "Packet 0" wait for more than

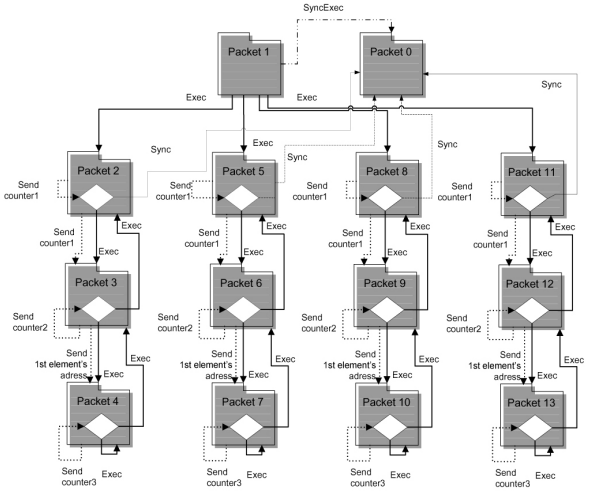

Figure 13. 2D-DCT implementation with four execution flows

one synchronization sign (sent by packets 2, 5, 8 and 11). Also, the access to the memory is done by packets 4, 7, 10 and 13.

The three scenarios (one, two and four execution flows) were compared with the STORM platform, considering the execution time and required memory. Figure 14 shows that the required memory for IPNoSys is lightly increased with more parallelism due the rise of the communication. Also it is bigger than all STORM experiments.

IPNoSys results are similar to parallel distributed memory STORM implementation, as it was expected. This is explained by the parallel and distributed features that add instructions to synchronization. In particular, the IPNoSys implementations, include extra instructions for communication and synchronization between the packets, even the sequential implementations (as EXEC, SYNEXEC and SEND instructions). Other reason can be found in the optimizations done by the compiler in the STORM source code. Contrasting to that, applications for IPNoSys are still manually written, since the compiler is under development.

However, the efficiency of the parallelism in the IPNoSys system is shown in Figure 15 that compares

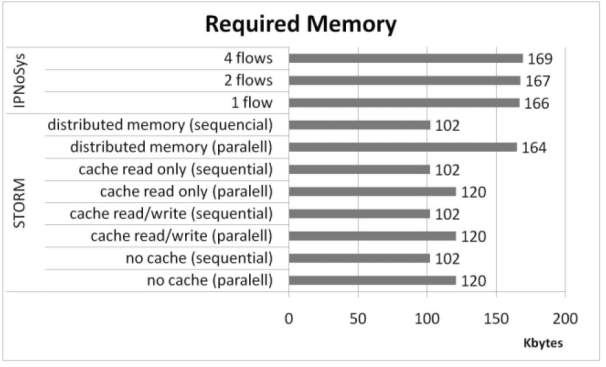

Figure 14. DCT required memory

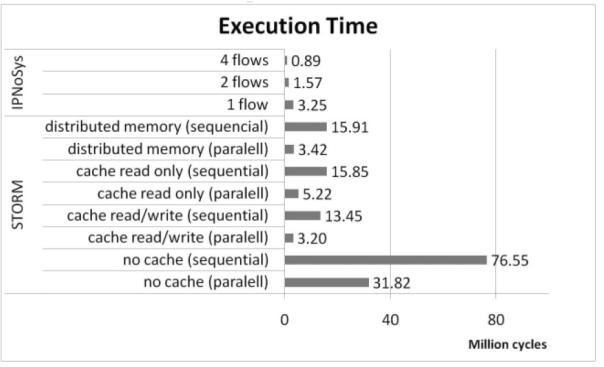

Figure 15. DCT execution time

the execution time between the three IPNoSys scenarios and the STORM instances. The STORM best case (parallel implementation executed in a cache read/write version) is faster only than sequential execution in IPNoSys (1 flow). However, Figure 15 shows that the execution through four flows in IPNoSys is 3.5 times faster than STORM best case, which indicates the potential of parallelism of the IPNoSys system.

B. RLE

Run-Length encoding [24], [25] 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. The second one is the character that is repeated. For image compression techniques, as JPEG format [26], the RLE can be adapted using the first byte to inform the number of zeros preceding other character, once the previous processes (DCT and quantization) generate large quantity of zeros. It is the method used in our implementations.

The RLE decoding process is similarly easy to be implemented, once it is necessary to extract the quantity of data indicated in a pair of bytes (run). However, an interesting feature of this application, to IPNoSys experiments, is the fact 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. The parallel version was performed with four execution flows. Other interesting feature of this application is the larger quantity of traffic to memory where the decompression results are stored.

Although the amount 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 theoretically demonstrate this, the equation (1) will be used to calculate the growth ratio of the application packets. The variables of this equation are calculated observing the application packets statically, independent of the number of times each packet will be injected.

For this implementation the sum of the all inserted data of all packets as results was 233. 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 presents in the packets before its injection for execution. Using (1), the growth ratio was 0.41, a value lower than 1, so the number of removed words is bigger than inserted words, that meaning the packets decreasing, on average, at the end of its execution.

The RLE decompression algorithm was implemented in a sequential and parallel way and simulated in IPNoSys. Figure 16 and Figure 17 show the organization and synchronization between the packets in the both cases. In Figure 16 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 (ZQ) and the other value (N)) from the memory and sends them (ZQ and N) to "Packet 3", that decodes them and stores the results in the memory. "Packet 2" is executed as many times as quantity of runs (a pair ZQ and N) 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 runs the "Packet 2" sends a SYNC signal that provokes the execution of the "Packet 0", which to finalize the application execution.

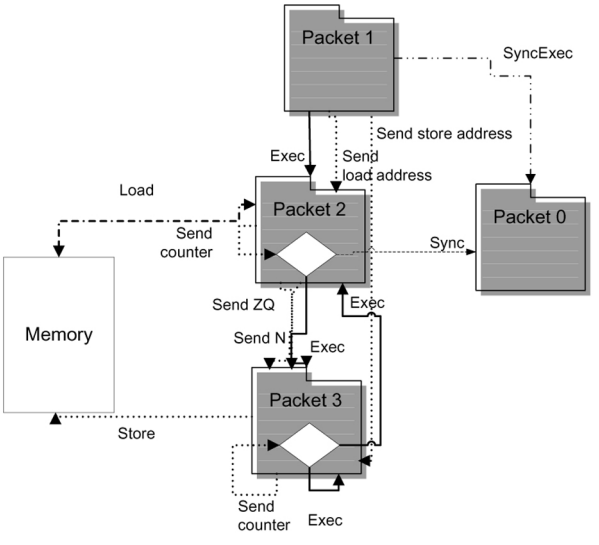

Figure 16. RLE decompression implementation using one execution flow

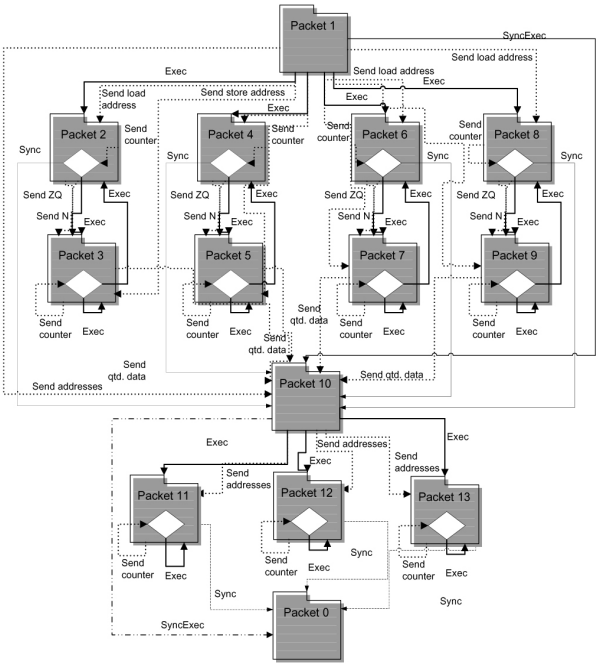

Figure 17. RLE decompression implementation using four execution flows

Figure 17 shows the RLE decompression algorithm parallel version (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 to 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" only 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, SYNC signals are sent to execute the "Packet 0", the final packet.

To analyze the performance of this application, a comparison (in cycles) was made between IPNoSys and STORM. Figure 18 shows the achieved results.

It was implemented a sequential and a parallel version for IPNoSys. As the best performance of the STORM is a parallel implementation that uses cache read/write, it was used different instances of this platform with this kind of cache, changing the number of processors and consequently the NoC dimensions. Figure 18 shows that the best performance of the RLE algorithm in STORM happens in the instance with 15 processors (each one with a cache), however the execution time still is 2 times higher than the par-

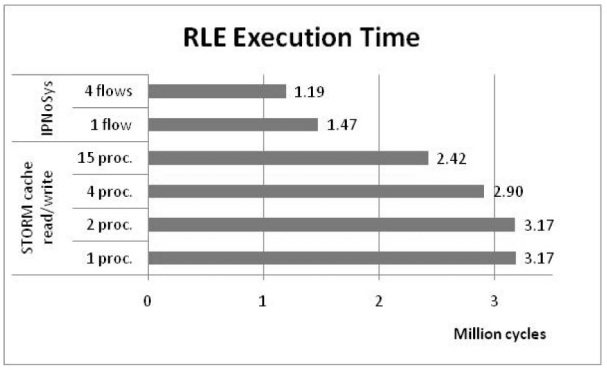

Figure 18. RLE Execution time

allel execution in IPNoSys and 1.6 times higher than the IPNoSys sequential implementation. This is indicate that despite this application produces more output data than input data, and consequently, there are a great quantity of control packet to store data in memory modules, the latency of this packets is compensated by the computation speed achieved by all the RPUs in the routing way of the packets.

6. CONCLUSIONS

This paper presented an innovative communication architecture, based on packet-driven execution, which does not use traditional processors to execute general purpose programs: the IPNoSys approach. Such architecture has simple Memory Access Units (MAUs) only in the network's corners, while the routers become Routing and Processing Units (RPUs). Applications instructions are bundled in packets and are executed by the time the packets flow from the source to the destination, using a new routing algorithm, called spiral complement. In addition, the packets' size decreases during execution allowing new packets to be injected within the same bandwidth.

The IPNoSys also identifies and avoids deadlocks by implementing a technique called "local execution".

The paper also has presented the IPNoSys programming model, including the Instruction Set, programmability resources and the Packet Description Language (PDL), used to create executable programs for IPNoSys. In order to evaluate and validate the architecture and its programmability, two applications were considered: 2D-DCT and decompression RLE. For each one, a sequential and parallel version were implemented. For the parallel version, the relationship between the packets and its synchronization were also discussed.

The DCT was used to compare the performance of IPNoSys against the performance of a MP-SoC cycle accurate virtual platform, called STORM. This simulation showed that IPNoSys required the same amount of memory than the STORM platform. However, the execution time in the IPNoSys was 3.5 times smaller than the STORM best case. This shows the efficiency of the parallelism in this system.

The IPNoSys performance for the RLE algorithm was also better than STORM. This has shown how the parallelism could be explored in IPNoSys with an application where the output processing data is much larger than input ones. This means that the high data rate demanding on memory access, could be better implemented in IPNoSys then in the STORM platform.

Future works includes: utilization of virtual channel to send regular packets; new routing algorithms; RPU configuration before the application execution; compiler development; FPGA prototyping for chip area and power dissipation evaluation and larger system simulations using clusters of IPNoSys.

REFERENCES

[1] V. Borkar. (2005, Jun 2009). Platform 2015: Intel processor and platform evolution for the next decade. *Intel Corporation white paper,* 3-12.

[2] C. A. Zeferino and A. A. Susin, "SoCIN: A Parametric and Scalable Network-on-Chip," in *Proceedings of the 16th symposium on Integrated circuits and systems design*, 2003, pp. 169-174.

[3] I. Walter*, et al.*, "The era of many-modules SoC: revisiting the NoC mapping problem," in *Network on Chip Architectures, 2009. NoCArc 2009. 2nd International Workshop on*, 2009, pp. 43-48.

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

[5] S. Fernandes*, et al.*, "Processing while routing: a network-on-chip-based parallel system," *IET Computers & Digital Techniques,* vol. 3, pp. 525-538, 2009.

[6] S. Kumar*, et al.*, "A Network on Chip Architecture and Design Methodology," presented at the Proceedings of the IEEE Computer Society Annual Symposium on VLSI, 2002.

[7] P. P. Pande*, et al.*, "Destination network-on-chip," *EDA Tech Forum Journal,* pp. 6-7, 2005.

[8] G. Girão*, et al.*, "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.

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

[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*, et al.*, "Queue Machines: Hardware Compilation in Hardware," presented at the Proceedings of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2002.

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

[13] B. A. Abderazek*, et al.*, "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*, et al.*, "Parallel Queue Processor Architecture Based on Produced Order Computation Model," vol. 32, ed: Kluwer Academic Publishers, 2005, pp. 217-229.

[15] H. Tran Le Nguyen, "Network-on-chip dataflow architecture," United States Patent, 2006.

[16] G. Sassatelli*, et al.*, "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.

[17] Intel. (1988, Dec 2008). *iWarp Project*. Available: http://en.wikipedia.org/wiki/IWarp

[18] R. M. Hord, *Parallel Supercomputing in MIMD Architectures*: CRC Press, 1993.

[19] S. Fernandes*, et al.*, "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, 2009.

[20] OSCI. (2005, 20 jan. 2008). *SystemC*. Available: http://www.systemc.org

[21] M. Kovac and N. J. Ranganathan, "A Fully Pipelined VLSI Architecture for JPEG Image Compression Standard.," in *Proceedings of the IEEE*, 1995, pp. 247-258.

[22] R. S. d. L. S. Rego, "Design and implementation of a MP-SoC platform using SystemC," Masters degree Master dissertation, Informatics and Applied Mathematics Department, Federal University of Rio Grande do Norte, Natal, 2006.

[23] R. Soares*, et al.*, "When reconfigurable architecture meets network-on-chip," presented at the Proceedings of the 17th symposium on Integrated circuits and system design, Pernambuco, Brazil, 2004.

[24] J. D. Murray and W. V. Ryper, *Encyclopedia of Graphics File Formats*, 2nd ed. Sebastopol: O'Reilly & Associates, 1996.

[25] L. V. Agostini, "Design of integrated architecture to JPEG images compression," Master Master, Instituto de Informática, Federal University of Rio Grande do Sul, Porto Alegre, 2002.

[26] J. P. E. Group. (2007, Jan 2010). *JPEG Homepage*. Available: http://www.jpeg.org/jpeg/index.html