Grant Agreement number: 288574

Project acronym: **vIrtical**

Project title: SW/HW extensions for virtualized heterogeneous multicore platforms

Seventh Framework Programme

Funding Scheme: Collaborative project

FP7 -ICT -2011-7

Objective ICT-2011.3.4 Computing Systems

Start date of project: 15/07/2011 Duration: 36 months

D 4.1 *Hardware hooks for the programmable features of the system*

Due date of deliverable: Month 18

Actual submission date: Month 20

Organization name of lead beneficiary and contributors for this deliverable: UPV, UNIBO, TEI
Work package contributing to the Deliverable: WP4

Dissemination Level		
PU	Public	X
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

APPROVED BY:

Partners	**Date**
All partners	**March 6th, 2013**

Table of Contents

Abstract

This deliverable describes the key hardware extensions for embedded system virtualization. Such extensions are runtime programmable, thus augmenting the hardware platform with the needed flexibility and dynamism that a virtualized environment requires. The hardware extensions include a monitoring facility and a reconfiguration facility, which will be thoroughly illustrated throughout this deliverable. One one hand, hardware primitives will form the monitoring facilities towards dynamic observation and management of a heterogeneous SOC target architecture. In WP3, they will assist the system software (OS and hypervisor) in the configuration of the embedded hardware in the best appropriate way to maximize performance of applications and user experience. On the other hand, this deliverable details the hardware support for the effective virtualization of the GPPA. To achieve such property, we report on the partitioning support at network-on-chip level, while at the same time achieving partition isolation, on a runtime reconfiguration strategy that yields flexible partitioning while avoiding deadlock, and the on a soft-QoS package at packet and flow level. Above all, the smooth integration between the partitioning, reconfiguration and QoS features is addressed. Hardware-level partitioning will also provide means to find a balance between performance, safety, and security for system integrator. While this document focuses on algorithms which are implemented at the HW level, the configuration and run-time API to the OS/hypervisor/applications will be defined in WP2 and WP3.

.

Glossary

DMA - Direct Memory Access

DRM - Digital Rights Movement

DVM - Virtual Memory Management

GPPA – General Purpose Programmable Accelerator

KVM - Linux Kernel Virtual Machine

LBDR – Logic-Based Distributed Routing

NoC - Network-on-Chip

SoC - System-on-Chip

VM - Virtual Machine

VMM - Virtual Machine Monitor

VP - Virtual Platform

VC – Virtual Channel

1. Introduction

Within the EU/ICT Collaborative Project vIrtical - workpackage WP4, the current deliverable describes synergistic hardware extensions involving monitoring components and techniques necessary to support efficiently not only existing processor virtualization, but also *hardware-assisted full virtualization* towards a heterogeneous SoC target architecture.

The proposed heterogeneous multicore system platform follows a trend towards convergence of high-end embedded systems and general purpose platforms for nomadic computing. Thus, programmable accelerators coexisting with ARM SMP multicore hosts are able to achieve the required energy efficiency (MOPS/mm/W) for embedded devices. Hardware-level enhancements advance virtualization technology by alleviating software overheads, ultimately exploiting the high performance capabilities of the underlying physical layer. At the same time a heterogenous multicore platform is inherently complex with a multitude of different subsystems that need careful tuning and runtime management.

In this deliverable we pursue the hardware extensions required to support virtualization in the architectural template of the project. Four main directions are taken: Monitoring facilities, partitioning support, runtime non-intrusive reconfiguration, and QoS provisions, in addition to the co-design and co-optimization of the three latter features, since they actually determine and change the operating mode of the embedded system device

Monitoring support. Given the additional dynamic information, decisions at system level can become more intelligent and achieve better performance and adaptivity. We address the use of a distributed infrastructure to **monitor** the state and dynamics of the system and provide feedback to the runtime environment (OS and hypervisor) and possibly to the application.

Based on the additional monitoring information available, decisions at system level can become more intelligent and achieve better operational characteristics and adaptability. Runtime monitoring support serves as a basis for the important tasks of providing security, performing debugging and improving performance of executing programs [7] [8] [9] . In general, monitoring management refers to the ability to track a number of events so as to offer better insight into the system's resource usage and into the behavior of applications at the same time. On top, the monitoring information that can be obtained and refers to the effectiveness of the different components in the system at a specific time interval can guide dynamic decisions of the operating system and of the hypervisor. Representative system responses could be frequency throttling, voltage reduction or resource reconfiguration depending on the exact nature of the deviation from expected system operation.

In a virtualized environment, multiple OS instances are involved, namely the hypervisor (VMM) and the guest(s). One of the hypervisor roles involves scheduling processor and system resource access to virtual machines as they need them. Apparently, with multiple VMs running on a single system, the virtual machines that aren't actively serviced by the hypervisor actually enter a type of wait state until their next turn. Hence, effective management of resources can result in optimized utilization and performance.

The objective of a monitoring scheme is to view how many resources those virtual machines (VMs) are consuming inside a virtual host or at the system level. However, besides processor's utilization in a heterogeneous multi-core system monitoring is required across all activities being done on the system. The deliverable details newly proposed subsystem components for monitoring NoC-based heterogeneous system and in particular the network interface infrastructure and memory controllers.

Partitioning support and **reconfiguration.** In addition, partitioning support and reconfiguration is reported in this deliverable. Partitioning support aims at defining sets of resources and isolating them at the NoC-level from the other partitions. This is a requirement to guarantee no influence between the accelerated sections of running applications. In this report we show the hardware support that delivers partition definition and isolation in the GPPA of the target heterogeneous SoC architecture. As a relevant contribution, such support is developed on top of a logic-based distributed routing mechanism, which better matches technology and scalability requirements than table-based routing. The devised solution builds up a routing framework that leaves many degrees of freedom for the end designer concerning the choice of specific routing algorithms, partition shapes, and number of virtual channels. Of course, such choices are tightly interrelated, therefore the deliverable will propose a few relevant global architecture solutions that the designer can choose from. Also, this deliverable reports on a runtime reconfiguration protocol and its optimized implementation for an on-chip setting. The implemented protocol avoids deadlock during the reconfiguration process of the network routing function, and also aims at minimum intrusion on running traffic and mapped applications on the GPPA resources.

QoS. One of the key features for effective virtualization in embedded systems is that of providing APIs to application developers in both the open and embedded virtualized environment to allow them to negotiate with the hypervisor in terms of QoS/SLA. While such programming model implications will be investigated in WP2 (D2.1), this deliverable reports on the suitable hardware extensions in the GPPA to effectively support the QoS framework. In this deliverable, we aim straigth for a soft-QoS package, complemented by on-demand circuit switching whenever tighter guarantees are needed.

It should be observed that QoS not only concerns application-perceived performance metrics, but is a good-to-have feature even for platform management. In fact, vertical hardware/software exchange of monitoring information and/or configuration commands requires a suitable service level discriminating this kind of control traffic with respect to typical data and instruction traffic. As a main innovation with respect to state-of-the-art QoS NoC frameworks, vIrtical fosters two main approaches:

1- QoS for NoCs is impractically implemented in hardware only, due to the large and hard-to determine number of possible use cases at run-time. A proper mix of hardware facilities and software controlled management policies is vital to achieving efficient results. In this direction, vIrtical is able to deliver circuit switching in specific network segments without incurring the burden of setting up or tearing down circuits.
2- In this project we are going to offer runtime QoS differentiated services by leveraging runtime reconfiguration of the NoC backbone. It is worth observing that virtualization and QoS are two tightly interrelated requirements for embedded systems. In this direction, the extension framework of NoCs for QoS will be synergic with the routing mechanism extensions for network partitioning and isolation illustrated above.

Overall, while this document focuses on algorithms and primitives which are implemented at the HW level, the configuration and run-time API to the OS/hypervisor/applications will be defined in WP2 and WP3 (deliverables D2.1 and D3.1).

2. Hardware hooks for monitoring at system-level

In the scope of enhancing monitoring at system-level in VIRTICAL, a monitoring infrastructure is developed to provide efficient hardware primitives operating in a distributed fashion while facilitating dynamic measuring of metrics of interest and management of system resources. The monitor agents include counter-based blocks that capture configured events. Their contents may be set and read by software and used to analyze and optimize the performance and power consumption of the entire system. System-level metrics such as read or write data throughput, interconnect read latency can be computed by obtaining all metrics after selecting the agents of interest in the system.

The design of the monitoring subsystem involves a number of tradeoffs from architectural point of view, including communication protocols and software interfacing, as well interaction and interoperability; as such, the design must be performed in concert with the design of main multicore SoC resources. Collected real-time monitor information can involve substantial amounts of non-critical data (i.e., performance statistics, throughput, and jitter) that may require separate system resources to transfer and process them. On the other hand critical monitor information, such as power and temperature or soft-error failures require instantaneous attention at system level. At the same time system dependability is increasingly important in the face of numerous environmental and process-related variability that can affect operation and performance of modern complex SoCs; for instance these cover unexpected voltage drops in the power supply network, temperature fluctuations, process variations (gate length and doping concentration), and cross-coupling noise.

2.1. Monitoring Primitives in Hardware

Counter-based monitoring units comprise the primary components for collecting system events and maintaining statistics for performance optimization. Two main methods are integrated to retrieve monitor information: i) event-driven sampling that relies on interrupt notification when counter overflow happens, and ii) time-driven sampling where periodically the monitor manager collects monitor statistics. The following components are developed to facilitate accounting of various events.

- *Event counters*: Currently single 32-bit counters are used to accumulate the captured events and are programmable in order to be controlled by the software running on the manager of monitor components or on the processors itself. The layout of an event counter is depicted next.

Overflow Flag	Counter Value [31:0]
Set when wrap-around occurs	Number of events. The trigger condition is set by the associated control register. Each event can be qualified by a filter register.

The control register of each event counter contains the following fields:

Software Reset (bit [0]):	a write sets the event counter to zero
Counter Enable (bit [1]):	when set counting is enabled, otherwise the counter is frozen
Interrupt Enable (bit [2]):	when set the counter causes an interrupt when overflow occurs; reading the value of the counter causes the interrupt to deassert
Counter Trigger (bits [7:4]):	allows any of the four triggers to increment the counter when asserted
Privilege Level (bit [9:8]):	determines the privilege level of the counter manager in order to protect it from unauthorized access

- *Free-running timers*: These timers increment at processor or sub-system clock rate. Timestamps can be enabled tomark the timing of captured events related to a free-running timer that operates as a wall-clock reference. Additionally, it is useful to obtain events from different clock domains associated with a local timestamp when insight is needed at a block level. However, currently we opt for a centralized counter/clock to avoid distributed time consensus issues. A tuple {timestamp, event} forms an event-time structure.

- *Event Filter*: A monitor filter commonly is build on the basis of a masking operation that applies on the captured sample in order to isolate the field of interest. The developed multi-filter unit mainly consists of three masks which are user programmable. The following figure shows the signal vectors/inputs *Vdata* and *Vtrigger* that comprise the data bus to monitor and the signal to trigger the sampling of this data bus. The masks can be optionally programmed by the user software and specify which part of the *Vdata* and *Vtrigger* is desired to be qualified.

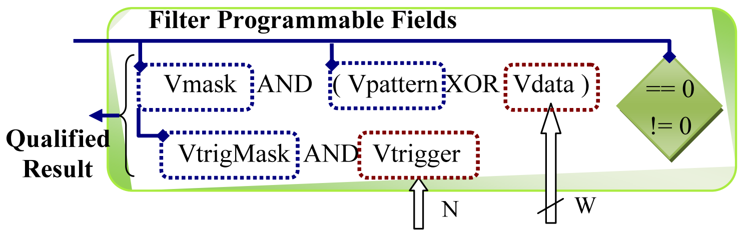


Figure 1: Structure of a single programmable monitor *Event Filter*; vector inputs Vdata and Vtrigger are specified by the designer at configuration time, while Vmask and Vpattern are programmable at run-time

- *Statistic Counter – Moving Average*: Consists of a counter with associated logic in order to manage an input stream of events (x) indexed by time (e.g. xt is the value of x at time t), or counter values, while a new piece of data is received in a sliding window time interval measured in clock cycles. Hence, for a sliding window of eight entries the computations needed are an addition and one subtraction:

  Sumt+1 = *Sumt* - *xt*-8+*xt*

  Avgt = *sumt*/8

  To implement this hardware structure a circular buffer is configured with eight entries (the size is configurable at design-time).

- *Switching Activity Counter*: besides the counter this unit includes a circuit to compute the number of bit transitions in order to provide support for energy monitoring. A coarse-grain activity measurement can be based on accounting of traffic through counting number of packets. This particular switching activity circuit offers very fine-grain metrics.

- A *system-level block featuring multi-counter-based measurement units* is architected in slices, providing a scalable solution in order to accommodate for capturing an extended number of events. On the other hand, the reduction of utilized slices can amortize the cost in terms of area and energy consumption. The developed infrastructure follows a middle ground approach by employing a shared control and interface glue logic that does not provides the ultimate performance but caters for the needs of medium multicore SoCs.

  The developed monitoring structure is developed to operate in dual mode. The horizontal shadowed slice depicts the basic unit circuit, which includes a first level filtering, an equality full or partial comparison stage and final recording in the counter unit; the process is activated by event generation as indicated by event trigger case A. Alternatively, the left part of the slice can be utilized as a filter stage to access the counters, while the counters can work independently and log preconfigured events; the vertical shadowed rectangle indicate the counters activated by event generation (case B). The partitioning and modularity of this scheme allows for protection against illegal accesses.

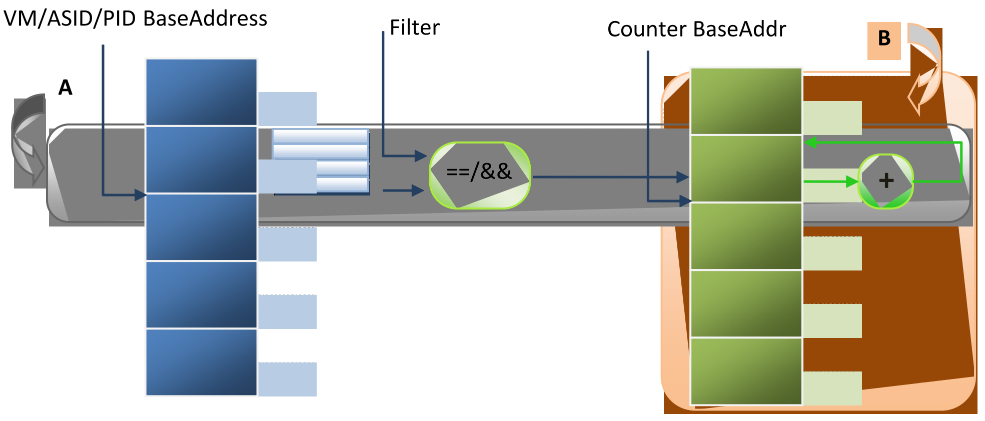


Figure 2: Organization of the multi-counter block Monitor

2.2. NoC Monitoring

In the context of integrating monitoring capabilities for the system NoC in VIRTICAL, one of the primary objectives is to develop advanced monitoring components in close synergy with network interfaces bringing a dynamic nature to these. Real-time collected statistics can assist, first, in optimizing provided QoS levels in terms of latency, throughput and jitter. Hypervisor and host's applications can provide improved resource management with the aid of run-time metrics such as throughput or packets slack. A packet's slack is the number of cycles the packet can be delayed in the network without affecting the application's execution time [1] . Second, based on run-time estimated metrics the system can adapt and improve the utilization of network resources and corresponding energy consumption through employing various reaction policies. By means of discovering potential congestion, adaptive routing mechanisms, such as those demonstrated in [2] , can be applied to support QoS traffic. Dynamic allocation of virtual channels and or queuing buffers, DVFS mechanisms, throttling and policing of packet injection rates are yet different alternatives to control NoC resources.

The developed monitoring probes provide an easy, fast and efficient infrastructure to jointly monitor NoC-based system resources and software applications running on top, and a seamless integration of the hardware monitor agents with the underlying NoC infrastructure in a non-invasive way.

Methodology

Monitoring a NoC infrastructure involves mainly two important aspects. In the viewpoint of supporting high-performance communication services, and particularly supporting guaranteed quality of service, the monitoring policies should be tailored to impose negligible interference to the system and its performance/latency characteristics. The second aspect of monitoring involves the location and amount of monitor information that needs to be maintained.

Monitoring mechanisms usually are required to provide throughput and latency statistics. In order to identify end-to-end events, such as request-replies in a NoC and corresponding latency or turnaround time mainly two techniques can be utilized. Packets are either tagged with timestamps and potentially with additional information, such as a network interface (NI) identifier and a sequence number, or new separate monitor packets are generated to provide similar information to the monitor at the destination NI. Both techniques are intrusive. It is essential that the monitor resources should be kept at a minimum to achieve low overhead.

The developed monitoring solutions are designed to differentiate in order to be employed on the basis of systems' constraints and requirements as follows:

- maintain monitor information locally (applied for statistic counters, and only if the sharing degree is low, i.e. if many processor threads desire to access these counters could cause potential traffic overloads)
- maintain monitor information in shared memory (appropriate for large amounts of traces through the monitor probes and if the sharing degree is high)

The developed monitor primitives can be utilized in various contexts at system level as we describe next.

Monitoring for Throughput accounting

One important metric that evaluates the quality of a network-on-chip is throughput. Bandwidth indicates the amount of data that can be put on the network in a given amount of time. The monitors that we designed can be spread over a NoC infrastructure to measure various variables that can be exploited to characterize the traffic over particular physical or virtual partitions of the NoC.

In particular, the developed counters are designed as performance counters to capture the number of packets departing a router in a predefined and programmable time interval and additionally can be employed to measure the buffer occupancy inside each router, so as to determine its level of congestion.

Furthermore, multiple performance counters can be integrated for a single router, or a network interface to address differentiation of the various traffic flows. Trading area overheads for supported counters per router challenges the use of block counters (with a lower granularity or sampling rate thereof).

Monitoring for Latency accounting

Latency is a difficult comparison criterion, because it depends on many application-specific factors. Depending on the application or the criticallity of a guest, minimum latency on a few critical paths can be more important to measure and ensure via particular policies than statistical latency over the entire traffic flows. The overall system–level SoC performance usually depends only on a few latency-sensitive data flows such as processor cache refills, while for most other flows only achievable bandwidth will matter. But even for the latter dataflows, latency does matter in the sense that high average latencies require intermediate storage buffers to maintain throughput, potentially leading to area overhead.

Latency can be measured as the round-trip delay of a read or write request performed by a master core. This can be done in software by the core itself, or by a monitor component in hardware. However, this entails the maintenance of a possibly large number of entries in a queue since a master can initiate multiple requests. We opted for distributed monitors that capture the latency of a packet as determined by the clock cycles form the time that the packet enters a router until the time that its first word departs from the same router. This difference is marked in the packet itself and is updated at each hop inside the netowrk-on-chip. Thus, when it finally exits the NoC a local monitor extracts the accumulated latency.

2.3. Monitoring RTL Implementation Results

The developed monitor hardware components are implemented in VHDL at Register Transfer Level and verified in an FPGA prototype to prove their feasibility and gain insight on the incurred cost.

Table 1 summarizes the implementation cost of indicative monitor configurations using a Virtex-4 VFX20ff672-10 device; the area of a MicroBlaze baseline core without local memory controllers or instruction and data caches is also depicted for comparison. Unless we integrate complex functions in hardware, such as compression or classification of events, the cost of integrating even multiple monitors with filtering capabilities is negligible.

Device Implementation cost of hardware monitor units			
Block	**Slices**	**RAMBs**	**Frequency**
MicroBlaze core v.7.30	1240	134	125
Counter with Event Filters	332	3	297
Switching Activity counter	148		387

The implementation results of the block counters for a Virtex4 xc4vfx20-11ff672 device are summarized next.

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	412	8544	4%
Number of Slice Flip Flops	612	17088	3%
Number of 4 input LUTs	549	17088	3%

Number of FIFO16/RAMB16s	2	68	2%
Performance Summary			
Minimum period:	3.232ns	Maximum Frequency:	309.406MHz

2.4. Monitoring RTL implementation validation at system-level

The developed hardware components are implemented in VHDL at Register Transfer Level and verified in an FPGA prototype to prove their feasibility and gain insight on the incurred cost. We have integrated our monitor models with the Hermes NoC [3] . This allows for direct validation and calibration of our monitor components. By deploying Hermes NoC, we have designed several different candidate NoC configurations and compared our monitor simulation estimates for these architectures with the real measurements. We investigate monitoring for two synthetic applications mapped on a four-by-four creditbased NoC. The packets consist of sixteen flits and the router buffers are matched to store sixteen flits. As figure 3 shows, four shared memories are connected at leaf nodes R00-R30, while each application consists of different traffic generators and occupies four tiles. Each traffic generator generates memory requests following exponential distributions, which range from 300 to 800 Mbps. Both applications access the shared memories using the NoC's XY-routing protocol, thus causing link sharing as well.

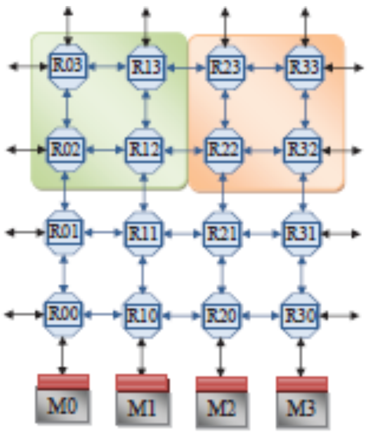


Figure 3: Traffic generation applications mapped onto two sets of cores and accessing the shared memories. The monitors capture latency for the two applications using the centralized monitor unit

The distributed monitors capture latency effects, which is a special field tagged inside each packet. The monitor is triggered when the latency exceeds fifty clock cycles, and notifies a hardware centralized manager when a critical threshold of one hundred clock cycles is surpassed. The type of application Ta (or identifier of VM is used interchangeably), the event type Te and the source node Sij form the tuple request {Ta,Te,Sij} are sent to the centralized event monitor.

One option is to employ time multiplexing of architectural event sampling to obtain all the values needed for latency calculation. We used point-to-point links to transfer monitor information to the monitor manager.

In this scenario only latency events are recorded, and additionally the interface of the centralized monitor combines incoming requests through "OR" operations. Hence, the tuple needs ten bits in total (two for the application and eight for the cores), while we could also include the monitor identifier in order to identify the congested memory block instead, or additionally to the source core. Figure 4 depicts the simulation results captured from the monitors for the generated traffic scenario. The middle graph of each scenario case depicts the events handled by the centralized monitor component as reported by the real prototype system. Finally, the bottom graph demonstrates the latency measured at the centralized monitor, including combining operation, incoming FIFO latency, recording in the internal context addressable memory and service delay.

Overall, the average delay achieved by the hardware monitors is almost fifteen clock cycles, including capturing, selection and transmission of the identified event. This clearly demonstrates the benefits of using our architecture in a multi-way high speed classification of monitor events.

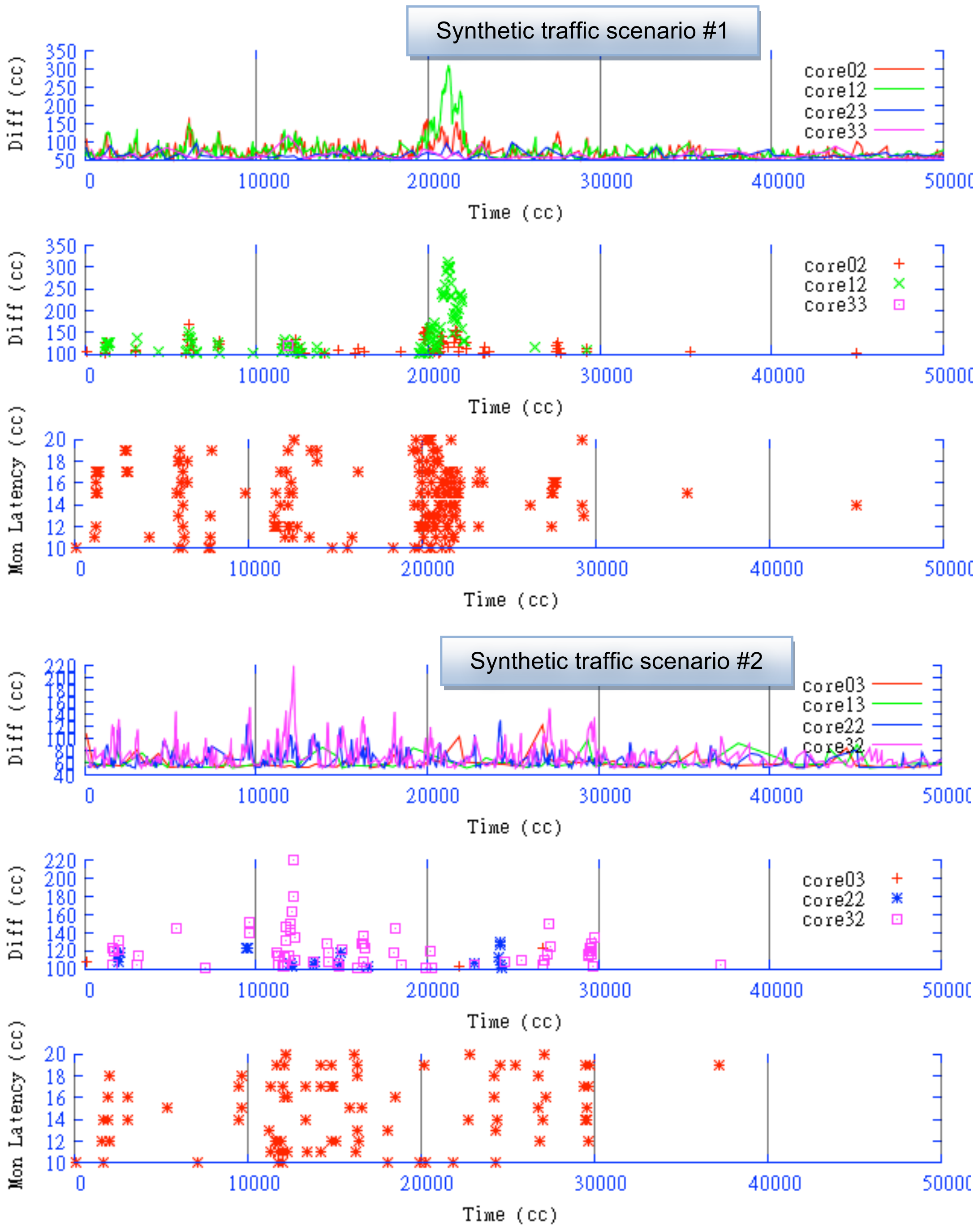


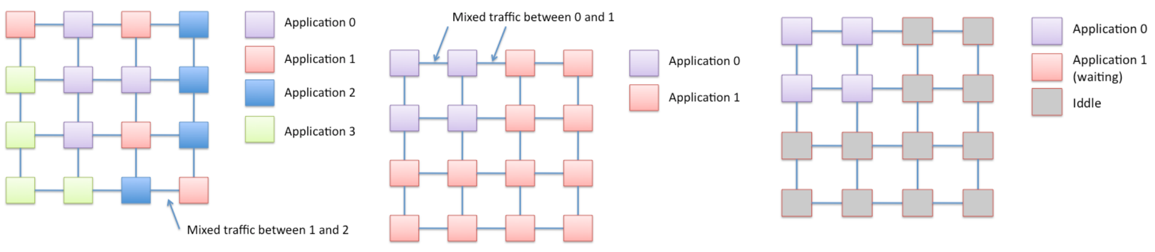
Figure 4: Latency captured by monitors for transaction across a 4×4 NoC to shared memories and monitoring latency for the two applications while using the centralized monitor unit

Through the usage of the monitoring block counters described in this section the latency for each core or task can be identified and maintained separately. Then, as proposed by Al Faruque [9] task mapping algorithms can be applied to optimize energy-performance metrics, which is nevertheless out of the scope of this report.

3. Partitioning Support

3.1. Preliminaries

One key aspect for the virtualization support aimed in vIrtical, is the possibility to provide partitioning support inside the GPPA. Multiple applications will be running on top of the system and the resources of the GPPA need to be partitioned in space in order to provide a perfect isolation of the communicating traffic between different application domains. Figure 5.a shows the case where four applications are using the GPPA resources without a partitioning support. Nodes with the same collor are assigned to the same application. As can be observed, the traffic generated by different applications collides in the NoC of the GPPA as the NoC paths are shared between nodes assigned to different applications.



(a) mixed application (b) smart allocation (c) low utilization

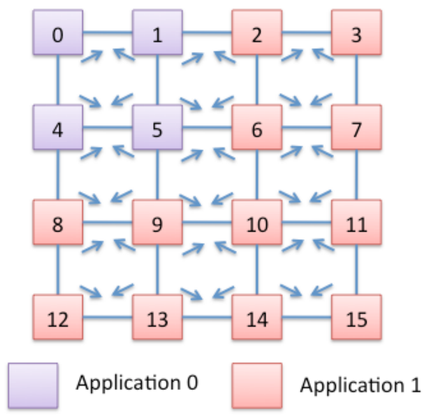
Figure 5. Application resources in GPPA lead to traffic collisions between applications.

Also, even if nodes are assigned smartly to applications, trying to avoid traffic conflicts, we can either come up with a conflicting case or with subobtimal assignment and low GPPA resource utilization. This is the case shown both in Figure 5.b and Figure 5.c, respectively. In both figures, the NoC uses the XY routing algorithm (messages are enforced to take X direction and then Y direction only) to avoid network deadlocks. In Figure 5.b we can see one partition is assigned 4 nodes wheras the other has the remaining nodes (12). The second partition will use XY routing and thus will have some communicating nodes with traffic crossing the other application domain. In Figure 5.c, to prevent the previous case the only mapped application is the first one, thus loosing GPPA resource utilization and affecting overall system performance.

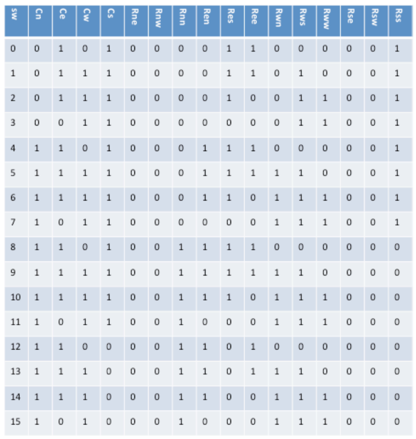
In vIrtical we apply the LBDR routing concept inside the GPPA, combined with a proper instantiation of the mechanism to support efficient partitioning of the resources. LBDR has been previously designed in the framework of the NaNoC project and aims at providing an scalable implementation of most deterministic routing algorithms for NoCs. It is based on three routing bits and one connectivity bit per output port of each NoC switch. The routing bits tell the switch whether messages can cross that link and then take the next one (three possible) at the next switch. Basically, those bits encode the so-called routing restrictions (the complement of routing restrictions, indeed). With the connectivity bit the logic only knows whether the output port exists or not. Figure 6 shows the switch IDs and the routing restrictions and the LBDR configuration bits for the case. In this case, the routing algorithm implemented in LBDR is XY.

As an example of routing bit, the bit Rne defined for switch 5 is set as 0. This means no message can be forwarded through the north port and at the next switch take the east port. As can be seen, there is a routing restriction (arrow) defined for that move. Also, the connectivity bits defined for switch 4 are all set to one except the Cw bit, which obviously is set to zero as there is no west link attached to switch 4. Notice that LBDR bits are encoded following a routing algorithm, in the previous case the XY routing algorithm. Other routing algorithms can be used, for instance the Segment-based routing algorithm, which provides much more flexibility.

The vIrtical project extends the use of the LBDR bits in order to support truly partitioning and maximum flexibility in partitioning definition. Indeed, in this work, we settle the basis for the definition of partitions and LBDR bit configurations. Also, the resources required for special partition configurations are shown. As a principle goal, the partitioning support must be maximized in the sense that all possible (and usable) partition configurations can in practice be set and used. Next, we detail the partitioning support opportunities with LBDR and applied to vIrtical.



(a) switch IDs and routing restrictions (arrows)



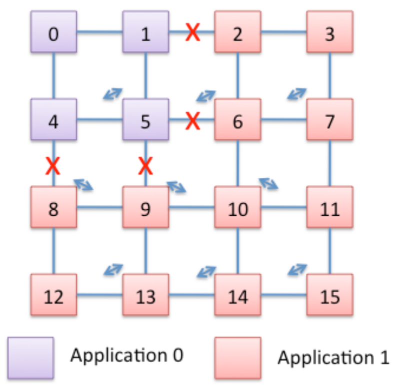
sw	Cn	Ce	Cw	Cs	Rne	Rnw	Rnn	Ren	Res	Ree	Rwn	Rws	Rww	Rse	Rsw	Rss
0	0	1	0	1	0	0	0	0	1	1	0	0	0	0	0	1
1	0	1	1	1	0	0	0	0	1	1	0	1	0	0	0	1
2	0	1	1	1	0	0	0	0	1	0	0	1	1	0	0	1
3	0	0	1	1	0	0	0	0	0	0	0	1	1	0	0	1
4	1	1	0	1	0	0	0	1	1	1	0	0	0	0	0	1
5	1	1	1	1	0	0	0	1	1	1	1	1	0	0	0	1
6	1	1	1	1	0	0	0	1	1	0	1	1	1	0	0	1
7	1	0	1	1	0	0	0	0	0	0	1	1	1	0	0	1
8	1	1	0	1	0	0	1	1	1	1	0	0	0	0	0	0
9	1	1	1	1	0	0	1	1	1	1	1	1	0	0	0	0
10	1	1	1	1	0	0	1	1	1	0	1	1	1	0	0	0
11	1	0	1	1	0	0	1	0	0	0	1	1	1	0	0	0
12	1	1	0	0	0	0	1	1	0	1	0	0	0	0	0	0
13	1	1	1	0	0	0	1	1	0	1	1	1	0	0	0	0
14	1	1	1	0	0	0	1	1	0	0	1	1	1	0	0	0
15	1	0	1	0	0	0	1	0	0	0	1	1	1	0	0	0

(b) routing and connectivity bits

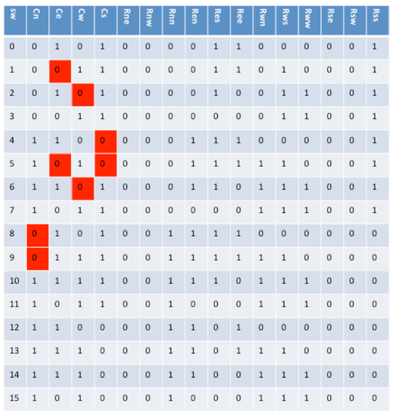
Figure 6. Routing restrictions, switch IDs, and LBDR bits

3.2. Basic Partitioning Support with LBDR

LBDR natively can provide partitioning support. This can be achieved by proper configuration of the LBDR connectivity bits. Figure 7 shows the case where the two previous partitions are configured with LBDR bits. As can be seen in the associated LBDR table, the highlighted connectivity bits of LBDR have been properly modified in order to avoid messages to escape from their domain. That is, messages are not allowed to leave their domain as the LBDR routing believes those links between domains do not exist. Indeed, partitions are configured by logicaly disabling links between partitions. The Cx bits involved are simply set to zero. Notice that messages can still progress within the domain as there are paths inside the domain. This is the case for the flow between switches 9 and 6 which can progress through switch 10.



(a) topology and routing algorithm (SR)



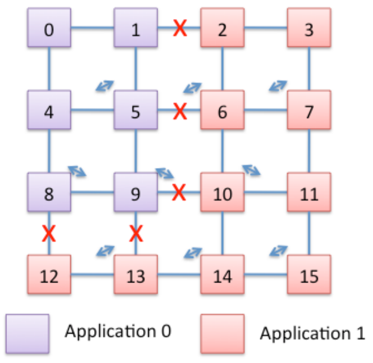
sw	Cn	Ce	Cw	Cs	Rne	Rnw	Rnn	Ren	Res	Ree	Rwn	Rws	Rww	Rse	Rsw	Rss
0	0	1	0	1	0	0	0	0	1	1	0	0	0	0	0	1
1	0	0	1	1	0	0	0	0	1	1	0	1	0	0	0	1
2	0	1	0	1	0	0	0	0	1	0	0	1	1	0	0	1
3	0	0	1	1	0	0	0	0	0	0	0	1	1	0	0	1
4	1	1	0	0	0	0	0	1	1	1	0	0	0	0	0	1
5	1	0	1	0	0	0	0	1	1	1	1	1	0	0	0	1
6	1	1	0	1	0	0	0	1	1	0	1	1	1	0	0	1
7	1	0	1	1	0	0	0	0	0	0	1	1	1	0	0	1
8	0	1	0	1	0	0	1	1	1	1	0	0	0	0	0	0
9	0	1	1	1	0	0	1	1	1	1	1	1	0	0	0	0
10	1	1	1	1	0	0	1	1	1	0	1	1	1	0	0	0
11	1	0	1	1	0	0	1	0	0	0	1	1	1	0	0	0
12	1	1	0	0	0	0	1	1	0	1	0	0	0	0	0	0
13	1	1	1	0	0	0	1	1	0	1	1	1	0	0	0	0
14	1	1	1	0	0	0	1	1	0	0	1	1	1	0	0	0
15	1	0	1	0	0	0	1	0	0	0	1	1	1	0	0	0

(b) LBDR bits

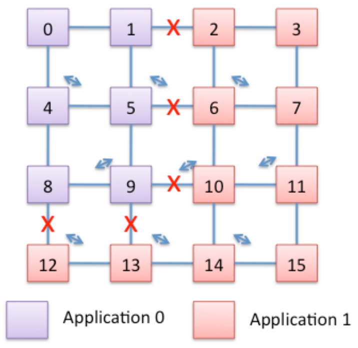
Figure 7. Two partitions defined with LBDR connectivity bits.

Also, is important to notice the simplicity and low overhead of this solution for partitioning support. Indeed, the LBDR mechanism does not require any single modification. In the past, the LBDR mechanism has been proved to be its overhead as large as the XY routing mechanism, and even to scale its overhead with switch radix and not with network size. Moreover, the routing algorithm implemented by LBDR does not need to be changed, not requiring any extra resource to guarantee deadlock freedom. Indeed, no virtual channel is needed. LBDR is the routing implementation technique chosen in vIrtical for the GPPA NoC.

One good mapping strategy should maximize the flexibility in defining partitions or domains. Indeed, the targetted partitioning mechanism in vIrtical must be ready for the allocation of tasks to resources in the GPPA in a way that GPPA resource utilization is maximiced. Thus, in principle, any domain shape should be possible to configure and use. The basic mechanism (just adapting connectivity bits), however, does not work in some scenarios. Take as an example Figure 8.a where the SR routing algorithm is used and coded in LBDR, and two partitions are mapped. The second partition has been adapted to fit the remaining resources of the GPPA. However, such partitioning configuration is invalid and has a severe problem. The path between switches 13 and 10 (or others) can not progress through the partition of Application 1 because there is a routing restriction at switch 14. The correct path to communicate nodes attached to the switches affected should bo outside the partition (through switch 9). However, the mapping performed prevents that path to be taken. Therefore, the match between the routing algorithm (represented by the routing restrictions) and the mapping configuration is invalid.



(a) incorrectly mapped



(b) correctly mapped

Figure 8. Two applications mapped with LBDR connectivity bits. SR routing.

Notice that the mapping algorithm, then, must conform somehow with the routing algorithm being used. Indeed, a different location of the routing restrictions would allow the mapping strategy to allocate the applications as done in Figure 8.a. This is the case shown in Figure 8.b. The routing algorithm is different but the mapping is the same.

To solve the previous problem we can take two directions. The first one is to simply restrict the mapping strategy to define only mappings that are in accordance to the underlying routing algorithm. This is the case shown in Figure 8.b. The other direction is to reconfigure the routing algorithm and adapt it to the desired mappings. This would mean moving somehow from Figure 8.a to Figure 8.b. The vIrtical project will be enabled to deal with both solutions. However, they have different impact on performance and mapping efficiency. In this report we provide a summary of both solutions in the next sections.

3.3. Mapping Strategy without Changing the Routing Algorithm

In this solution, the mapping algorithm (to be implemented in the vIrtical hypervisor) will be aware of the underlying routing algorithm and will trigger only mapping solutions compatible with the algorithm, thus, not deriving incompatible mappings with the underlying routing algorithm. The mapping algorithm will be applicable to any routing algorithm instantiated with LBDR bits. Next we show in pseudocode the algorithm.

```
Function MappingRequest(num_resources)
      ids: array (n x m)
      num_free: integer
      next_id: integer

      if (num_resources>num_free) return -1
      if (square_shape_available(num_resources, ids)) return next_id
      if(rectangular_shape_available(num_resources,ids))return next_id
      if compatibility(num_resources, ids) return next_id
      reconfigure_routing()
      return MappingRequest(num_resources)
endFunction
```

The algorithm defines basic shapes of the partitions that can be defined. The basic partitions are: square, rectangular, p-shape, d-shape, q-shape, and b-shape. Figure 9 shows the 6 allowed shapes. The algorithm keeps an array of IDs as large as the network inside the GPPA (*ids*). For each router, the *id* identifies the partition the router belongs to. It also keeps the number of free resources (*num_free*).

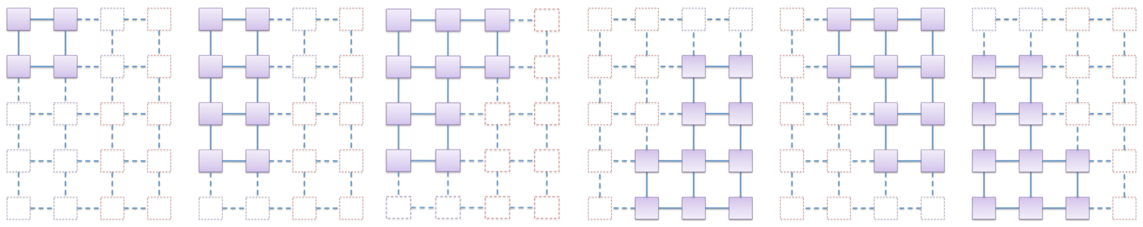

(a) square (b) rectangular (c) p-shape (d) d-shape (e) q-shape (f) b-shape

Figure 9. Partition shapes allowed by the partitioning algorithm.

The algorithm receives a new event, being a new task to be mapped, in which case it also receives the number of resources to be assigned (*`num_resources`*). If the number of free resources is lower than the number of requested ones, the mapping request is rejected (the algorithm returns *`-1`*). Otherwise, the algorithm maps a suitable mapping on top of the free resources. Priority is given to shapes in the following order: square, rectangular, x-shape. For square and rectangular the algorithm does not check compatibility with the routing algorithm. Indeed, those shapes do not need any message to leave the region to keep connectivity. This is because minimal routing is always enforced by the LBDR version used in this project. Notice that a different variation would be required in the case of using LBDR with deroutes (which enable non-minimal paths).

For the x-shape regions, if selected, then the algorithm checks for compatibility. If the *`compatibility`* function succeds (or the shape is square or rectangular), then the resources are assigned to the new domain (*`ids`* are updated) and the function returns the new *ID* for the domain. If not, the algorithm returns `-1` and the request is rejected.

The *`compatibility`* function is straightforward and requires only two checks to validate compatibility between the shape and the routing algorithm. Figure X7 shows the check for a p-shape. If the switch located at internal intersection of both rectangles (the critical switch labeled as C in the figure) has a routing restriction between the two links defining the boundary of the region, then the shape is not compatible. This can be easily checked by the proper routing bits at neighbour switches A and B in the figure. In case of any of those two bits are zero, then, in that case, there are no valid minimal paths for some pair of end nodes (indeed between A and B) through the partition. For the remaining shapes a similar check is performed. Indeed, all the shapes allowed are rotated versions of the p-shape.

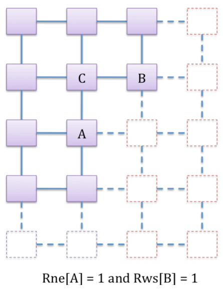


Figure 10. Condition to allow a p-shape to be formed.

Notice that this strategy may endup in configurations where enough resources are available but the underlying routing algorithm and the allowed shapes do not permit to achieve the mapping. This is the case shown in Figure X7. The algorithm is called for a new mapping with 3 resources (available) but simply the only possible shape (d-shape) is not compatible.

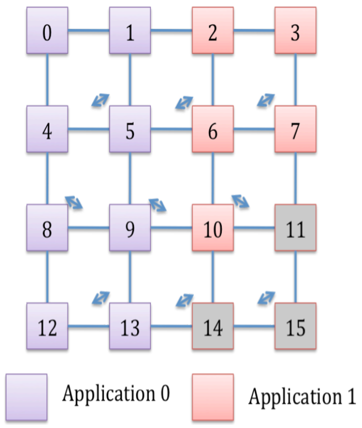


Figure 11. Incompatible third mapping on the GPPA.

One possible solution for those shortcommings could be the remapping of the partitions to make room for new compatible partitions. However, this would induce that already running threads in some nodes should be migrated, with the associated overhead. If more flexibility is required, in this project the solution conceived is to modify and adapt the routing algorithm to the mapped partitions (just the contrary to adapt partitions to the routing algorithm). This is shown in the next section.

3.4. Mapping Strategy with Changing Routing Algorithm

The idea of this solution is quite simple and is indeed, an extension of the previous one. Indeed, when the mapping algorithm fails to map a new partition, then, before quiting, it tries to adapt the underlying routing algorithm. This is achieved by calling the function reconfigure_routing. This function drives a total change of the routing algorithm driven mainly by the currently mapped shapes. For instance, if the previous d-shape is required to allocate a new partition, the function is called and the underlying routing algorithm is changed to the one shown in Figure 12. Now, the d-shape is allowed to be allocated, thus the algorithm is called again.

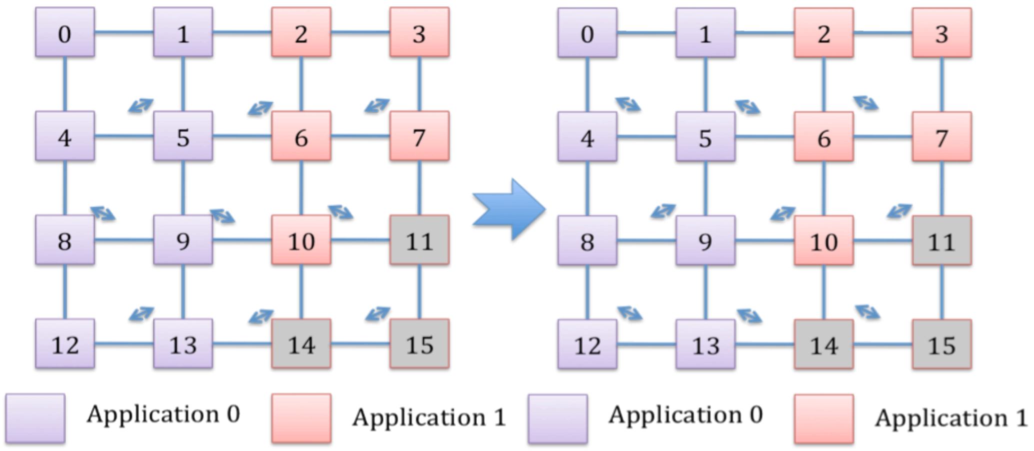


Figure 12. Changing the routing algorithm to allow a new d-shape partition.

Notice that, the routing algorithms shown so far perform the so-called *zig-zag* strategy when placing routing restrictions. All these variants belong to the same routing filosophy embedded in the SR routing algorithm. This zig-zag strategy is the most flexible one and will be used as the reference routing algorithm. Indeed, it allows to map partitions with all the shapes allowed by the mapping algorithm.

Notice that changing the underlying routing algoritihm can lead to a severe desaster for the whole system. This is because during the change some deadlocks can arise blocking the network and perpetuating the situation until the system is rebooted. To avoid such unwanted probability, there are two alternatives. First, the network can be drained, then the new routing algorithm is settled, and then the traffic is resumed (which is called static reconfiguration). Second, a dynamic reconfiguration strategy that simply avoids this race condition from happening is used. This will be described below in this deliverable. In vIrtical, whenever the routing algorithm needs to be changed (for any reason) the reconfiguration strategy will be invoked and will guarantee the smooth transition between the previous routing algorithm and the new one with out any traffic being significantly stopped.

3.5. Support for Global Network Traffic

Previously it was described the partitioning strategy for isolation in different partitions. However, the GPPA nodes have shared resources they need to be accessed from time to time. This is the case of the L2 memory and the NI connecting to the external system NoC. In such cases, there is a need to send messages from local nodes in a partition to external resources. The previous mapping strategy simply does not allow this to happen.

To solve this problem, the LBDR mechanism has been enhanced with more connectivity bits. Indeed, two connectivity bits per output port are implemented instead of one. One is referred to as *local* and the other as *global*. The local one is used by messages generated within the partition (messages sent from nodes to nodes). The global one is used by messages addressed to shared resources or generated in those shared resources and with destinations being nodes in a partition. With these bits, the LBDR mechanism is changed appropiately. Indeed, a multiplexer is added to the routing logic. The multiplexer simply takes the connectivity bit to check depending on the nature of the message. Messages, thus, need to encode in their header an additional bit that will instruct the switch its nature. It is a intra-partition message, or is a inter-partition message. Figure 13 shows the LBDR logic with the new components highlighted in red.

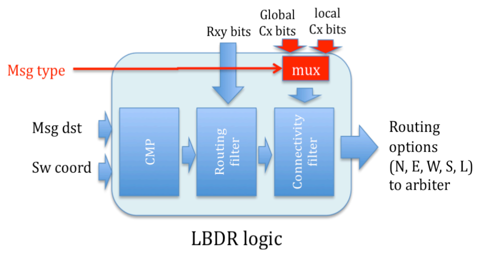


Figure 13. New logic to allow global and local traffic in the GPPA.

Notice that this change is orthogonal to the routing algorithm and does not compromise its deadlock-freedom property. Indeed, the routing algorithm is applied to the whole network and both inter- and intra- messages respect the same routing restrictions (routing restrictions are the same for all types of messages).

3.6. Possible Extension to Multiple Routing Algorithms

There is one possible new scenario when global and local traffic is allowed. This comes from the fact that one can think each partition could have its own routing algorithm. This is the case shown in Figure 14. Different SR instances are implemented on each partition, locally deadlock-free but globaly no. Indeed, this is just the addition of unrelated routing algorithms. If global traffic is not going to be present, then this solution is deadlock-free and probably each domain could have its optimum routing algorithm. However, if global traffic is to be present, then a different solution is needed.

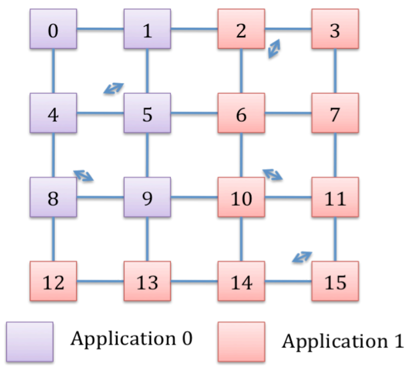


Figure 14. Example of local unrelated routing on each partition.

For this case, when different routing algorithms are defined in each partition, in order to allow global traffic, two virtual channels are required. One is for the local traffic and the other one is for the global traffic. Message differentiation can be achieved with the previous bit defined for the message header. However, there is an additional problem. Global traffic must have its own valid routing algorithm. Thus, at each switch there is a need of having two different routing algorithm implementations. When being implemented in LBDR this means the routing bits need to be duplicated and a multiplexor used, similarly as how it was done for the connectivity bits. Figure 15 shows the implementation.

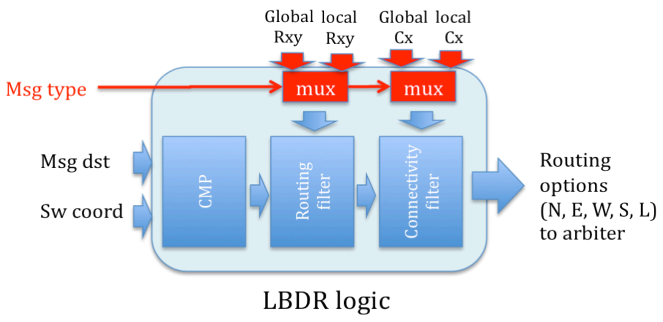


Figure 15. LBDR modifications to allow partition- and GPP-level different algorithms.

Notice that this solution doubles the overhead of LBDR (as double number of bits is needed). Although LBDR cost is low, it should be analyzed the benefits of this proposal. However, this depends on the needs of the local applications running inside the partitions to use a different routing algorithm.

4. Dynamic Reconfiguration

To address the new functionalities, the NoC must be enriched with an efficient reconfiguration process which enables the smooth and transparent transition between system configurations. For instance, Figure 16 shows two different configurations of a multicore system over time. In the first one (configuration A) different applications are mapped to the NoC nodes and execute concurrently, while other resources are powered down. Later, the resource manager may trigger a chip reconfiguration to power on unused resources and thus activate a new application (configuration B).

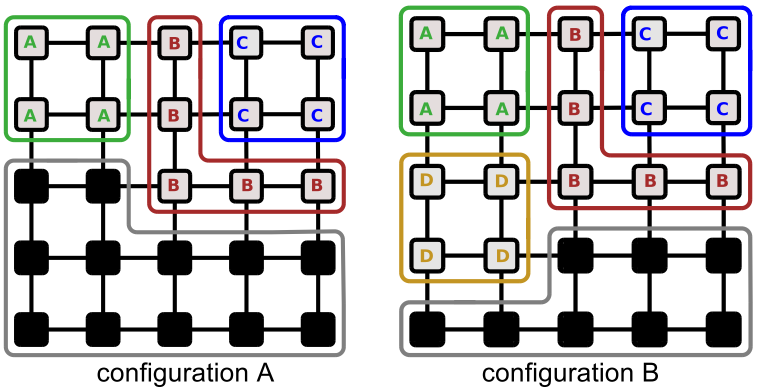


Figure 16. Two NoC configurations where the routing algorithm needs to be adapted.

The transition between configurations needs a careful design of the NoC routing algorithm, which establishes the paths for every packet in the network. At each configuration a different routing algorithm is needed. In both cases, the algorithm must be deadlock-free (should not introduce cycles in its channel dependency graph). However, in the transition between configurations, both algorithms can induce extra dependencies that lead to deadlock.

Therefore, in order to migrate from one configuration to the other, one possible approach is to drain the network, then changing the routing algorithm to the new one and finally resuming traffic injection with the new algorithm. This is the case of the so called traditional static reconfiguration (TSR). In this case system performance is likely to be heavily impacted by the reconfiguration process due to the temporarily low resource utilization. Alternatively, the network can be dynamically reconfigured, in the sense that traffic is not stopped during the reconfiguration process, but an effort is needed to avoid deadlock situations. This is typically achieved by devoting extra resources to the network. We refer to this case as the dynamic reconfiguration.

In this work we advance the state-of-the-art in reconfiguration frameworks for NoC-based systems. However, instead of designing a brand new reconfiguration mechanism, we recognize the large amount of bibliography and proposals made for reconfiguration mechanisms in high-performance off-chip networks. In this sense, we pick the approach that better suits the NoC domain and the tight resource budgets of the on-chip environment.

The Overlapping Static Reconfiguration process (OSR) in [11] enables a transparent system reconfiguration process. However, in [11] only the protocol was described while at the same time highlighting the key architectural requirements to properly support it (namely virtual channels, routing tables, event notification, involvement of end-nodes in the reconfiguration process). Unfortunately, no practical implementation insights were provided, thus raising the reader's skepticism on the applicability of OSR to an on-chip setting.

Here we report the implementation of the native OSR protocol in an on-chip network, proving that the needed network over-provisioning is such to make the protocol not viable in practice. As a consequence, we target the modification of OSR to better match the requirements of the resource-constrained NoC setting, thus resulting into the OSR-Lite framework. Such modifications concern both selected protocol features (without giving up the goodness of the underlying idea) and relevant implementation techniques.

With OSR-Lite in place, it is possible to reconfigure a whole 64-node network in a few hundreds of cycles, enabling the entire and transparent transition between any pair of independent and unrelated configurations. Moreover, this is achieved with no impact on network latency and with no impact on switch delay. The reconfiguration performance of OSR-Lite makes it the enabling tool for planned reconfigurations in multicore systems. The following specific scenarios can be therefore materialized by the outcome of this work:

- **Virtualization of the system**. Our method enables the runtime division of the entire network into sets of virtual regions for assignment to different applications running concurrently. This is the primary goal in vIrtical.
- **Power management**. The reconfiguration mechanism can be exploited for powering down unused resources; such functionality becomes compulsory to keep power consumption levels to reasonable bounds.
- **Reliability**. When a NoC is augmented with transient fault tolerance, then this kind of faults can be tolerated without any loss of information. However, intermittent faults are likely to be an indicator of the gradual onset of a permanent fault (typically, a wear-out fault). In this case, OSR-Lite can be used to reconfigure the network so to exclude the affected link/switch component, before the permanent fault shows up and causes packet loss.

4.1. Previous Work on Reconfiguration

During the last two decades, a large number of proposals have been presented about resilient routing for both off-chip and on-chip networks. These approaches are either nonreconfigurable fault-tolerant routing strategies which tolerate a limited number of faults [12, 13, 14, 15], or reconfigurable routing mechanisms that allow unlimited changes to the network. We focus on schemes of the second category, in particular on those based on reconfiguration processes that consider such changes to the network structure to obtain new routing paths replacing the previous ones.

In off-chip networks, such as those used in clusters, during a reconfiguration process, the topology resulting from the connection/disconnection or failure of network components is discovered by a central node, which runs the reconfiguration algorithm in software. The management software computes new routing tables and distributes them to each node. Detecting the new topology and communicating the new routing tables can be completed with or without traffic into the network. Static reconfiguration first stops and drains all user traffic from the network before completing the reconfiguration process [16, 17]. This reconfiguration method is unable to provide real-time and quality-of-service support needed by some applications. On the contrary, dynamic reconfiguration updates routing tables without stopping user traffic. In this case, the main challenge is to guarantee deadlock freedom as old and new routing functions are simultaneously active [18, 19, 20, 21, 22, 23, 24, 11].

In the context of networks on chip, new techniques have been proposed and other retain some features of the above. The Vicis NoC architecture [25] uses the turn routing model during fault-free operation, and a heuristic solution that makes exceptions to that routing model to maximize connectivity. Reconfiguration process rewrites the routing tables based on the information from built-in-self-test units in each router. When large number of faults occur, exceptions sometimes result in deadlocked routing paths.

A reconfigurable fault-tolerant deflection routing algorithm based on reinforcement learning for NoC has been proposed in [26]. The algorithm reconfigures the routing tables through reinforcement learning based on 2-hop fault information. In [27], a reconfigurable routing algorithm for a 2D-mesh NoC is presented. This algorithm introduces low hardware cost but can only be used in one faulty router or regular region topology. Other proposals can deal with irregular fault regions. A mechanism to tolerate failures in networks for parallel computers is described in [28]. It tolerates any number of failures regardless of their spatial and temporal distributions. Immunet is limited by the network connectivity and results in high area overhead because it requires three routing tables per router. In [29], a region-based routing has been proposed to handle irregular networks. This algorithm groups destinations into regions to make routing decision. However, it does not provide a reconfiguration method to migrate from one configuration to another.

Finally, [30] presented Ariadne, an agnostic recocfiguration algorithm for NoCs, capable of circumventing large numbers of simultaneous faults, and able to handle unreliable future silicon technologies. Ariadne utilizes up*/down* for high performance and deadlock-free routing in irregular network topologies that result from large numbers of faults.

Ariadne is implemented in a fully distributed mode. Thus it results in very simple hardware and low complexity although it comes with suboptimal solutions for lack of global view. The up*/down* routing will not perform optimally under certain configurations, specially in the absence of failures (in a 2D mesh). In addition, up*/down* routing is encoded in routing tables at switches. Unfortunately, the Ariadne latency badly scales with network size (the configuration latency increases with the square of the nodes number). This latter property has a severe impact on the network performance especially because Ariadne does not guarantee a transparent transition between configurations. The flits have to freeze in the network pipelines and the throughput drops to zero during reconfiguration. Even when the communication resumes, a high contention due to the fullness of injection queues strongly degraded the network performance for a long period.

As opposed to these solutions, OSR-Lite does not use routing tables at switches, allows coding any efficient routing algorithm (even DOR routing) and requires lightweight switch support to enable truly fast dynamic reconfiguration. Moreover its latency smoothly increases with network size, and the configuration transition is transparent, ultimately preserving the throughput of the system.

4.2. Native OSR technique

Typically, a routing algorithm is deadlock-free when its channel dependency graph (CDG) is acyclic (we do not consider fully adaptive routing algorithms). The CDG is set by representing the resources of the network by vertices (mainly the buffers associated with the ports of each switch) and the dependencies between two resources by arcs. There is a dependency between two resources r_1 and r_2 if a message can use r_1 and request r_2.

Two routing algorithms R_1 and R_2 are deadlock-free when they have an acyclic channel dependency graph. However, when using both algorithms at the same time new extra dependencies are induced potentially leading to deadlock. This can be seen in Figure 17 where a cycle is formed when using two routing algorithms (*XY* and *YX*) at the same time. During a reconfiguration process we refer to R_{old} as the old routing function and R_{new} as the new routing function. Similarly, packets routed with R_{old} will be referred to as old packets and packets routed with R_{new} will be referred to as new packets.

The native OSR method is based on the fact that those cycles are created only when old packets using R_{old} are routed after new messages using R_{new}. If old packets are guaranteed to never go behind new packets the extra dependencies do not occur in practice and then no deadlock can be formed. Indeed, in a static reconfiguration process the entire network is drained thus guaranteeing old packets will never go behind new ones.

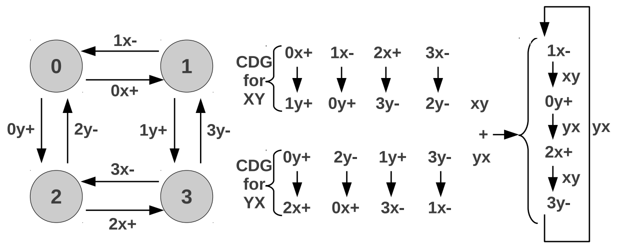


Figure 17. Channel dependency graph for two routing algorithms and their combination.

OSR is a static reconfiguration process but localized at link/router level, and not at network level. Indeed, it guarantees that new packets are only forwarded via links that have been drained from old packets. This is achieved by triggering a token that separates old packets from new packets. The token is triggered by all the end nodes and tokens advance through the network hop by hop. Indeed, tokens follow the CDG of the old routing function, draining the network from old packets. However, in contrast with static reconfiguration, the new packets can enter the network at routers where the token already passed. Figure 18 shows the complete native OSR mechanism, involving a central manager. In a first step, a reconfiguration action is triggered, either by the detection of a malfunctioning component or by a higher level manager in the system stack requiring a reconfiguration, e.g. a new application is admitted. In any case, when needed the central manager may receive event notifications through the network (step 1). Then, in step 2, the new algorithm for the new configuration is computed by the central manager. The resulting information is disseminated to all the switches in step 3. In step 4 the end nodes trigger the token and OSR spreads throughout the network (step 5).

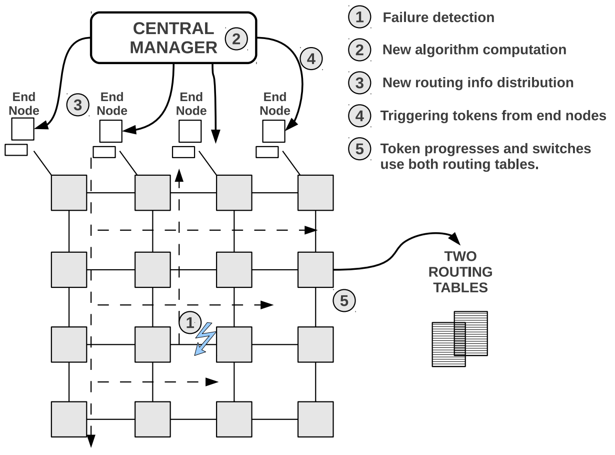

CENTRAL MANAGER
End Node
End Node
End Node
End Node
TWO ROUTING TABLES
1 Failure detection
2 New algorithm computation
3 New routing info distribution
4 Triggering tokens from end nodes
5 Token progresses and switches use both routing tables.

Figure 18. Reconfiguration steps performed in an OSR environment.

Figure 19 shows how tokens advance in a network. At a given output port, a token is triggered to the next downstream router indicating the output port has been drained from old packets. This is guaranteed when the token has been received through all the input ports of the switch that have old (R_{old}) output dependencies with the output port. These port dependencies can be extracted from the R_{old} routing algorithm. However, how to perform this is not explained in [11], although it is key to obtaining an efficient implementation. Notice that the token divides two epochs in the network, the old epoch (when packets are routed with the R_{old} routing function) and the new epoch (when packets are routed with the R_{new} routing function).

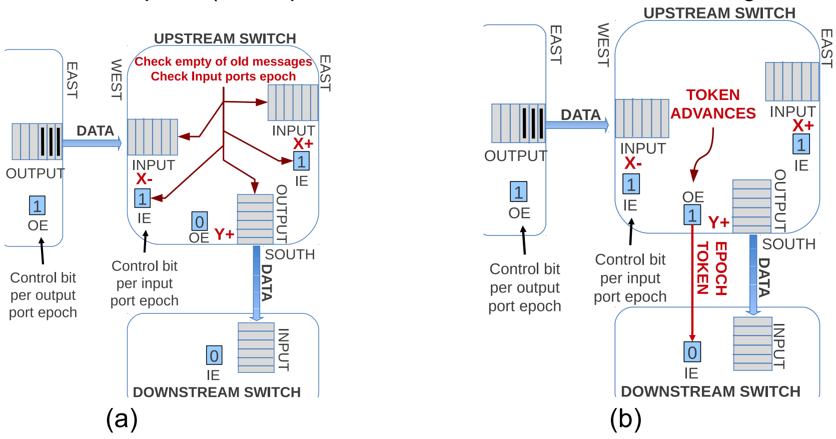


Figure 19. Token advance in a network: (a) check for absence of old messages and input ports epoch, (b) token signal propagation. The token separates old traffic from new traffic.

4.3. OSR-Lite

The OSR mechanism needs to be modified in order to better suit the NoC environment so to become an efficient and plausible mechanism for planned reconfigurations. Indeed, the main issues addressed in this work are the following:

- **Codification of the routing information**. During the reconfiguration process both routing algorithms coexist at the same time at routers. This means resources need to be sized for both algorithms. In OSR, routing tables were used to store the routing info. In NoCs, however, routing tables are an expensive resource in terms of access time, area, and power consumption. Therefore, hosting two routing tables per switch input port does not appear to be a cost-effective solution for OSR-Lite.

- **Control virtual channel (VC) used in OSR**. Different actions (sending routing information to routers, triggering the reconfiguration process) are performed during the OSR reconfiguration which imply the exchange of information between a central manager and the routers or the endnodes. In [11] this was implemented by means of a control VC. Unfortunately, using VCs only for that purpose has a large impact on router implementation (wseen later) and is not fully justified in an on-chip.

- **Reliable control VC assumed in OSR**. A different (spanning-tree) algorithm is assumed in OSR to effectively route control packets through the control VC.

- **Involvement of end nodes in the reconfiguration process**. In OSR the end nodes were notified to trigger the reconfiguration. This is done by end nodes injecting the token directly as a new packet. In NoCs, reaching the end nodes via dedicated packets from the central manager would be a time-consuming course of action. In order to cut down on the reconfiguration latency, involving only switches and not endnodes in the reconfiguration would be an appealing property in a NoC setting.

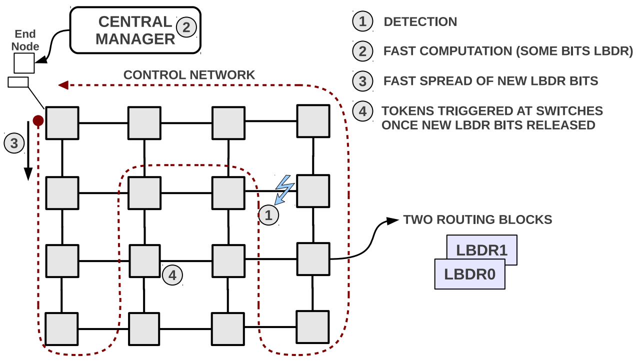

CENTRAL MANAGER 2
End Node
CONTROL NETWORK
1 DETECTION
2 FAST COMPUTATION (SOME BITS LBDR)
3 FAST SPREAD OF NEW LBDR BITS
4 TOKENS TRIGGERED AT SWITCHES ONCE NEW LBDR BITS RELEASED
3
1
4
TWO ROUTING BLOCKS
LBDR1
LBDR0

Figure 20. Reconfiguration steps performed in an OSR-Lite environment.

In order to address all these issues, we propose the OSR-Lite approach. Figure 20 shows all the steps and the main modifications performed. In particular, we exploit a control network through which routers can inform about expected topology changes (e.g., an output link is having frequent transient failures and is going to fail soon, or a region of the NoC is overheated and needs to be powered down). The control network collects all the notification events and sends them to a central manager (step 1). If the reconfiguration is instead initiated by a resource manager in the context of power management or virtualization strategies, step 1 can be skipped. The central manager then computes the new configuration (step 2) and disseminates the new routing information to the switches (step 3). Then, every switch starts the OSR-Lite reconfiguration process in step 4. Notice that end nodes are not involved in the reconfiguration process.

The control network can be used also in step 3 for routing bit dissemination to the switches. In previous work in [31] we have presented the design of a dual network for switch-to-global manager bidirectional signaling, thus offloading critical control tasks from the main data network. In that work, the dual network was used to notify diagnosis information to the manager following the main NoC testing phase, and to notify configuration bits of the routing mechanism to the switches. The same network could be reused for other purposes, such as congestion management, deadlock recovery and software debugging. In [31] it is showed to be a cost-effective solution for control signaling, which can be easily and effectively made reliable through a combination of fault-tolerant and online testing strategies. For this reason, this work relies on such a fault-tolerant dual network to convey control information of the reconfiguration process. Furthermore, [31] also reports an efficient computation algorithm that comes up with the routing configuration bits of a new network partitioning or topology shape. This is the algorithm the controller runs in step 2. Given that the control network and the computation algorithm are covered by previous work, from now on we focus on the core reconfiguration process of the network and on the microarchitectural support for that. The reader should keep in mind that all these mechanisms will work together in the complete reconfiguration framework. In the next section we describe the router implementation in more detail.

4.4. OSR-Lite implementation

Without lack of generality, we use the xpipesLite switch architecture [32] to prove viability of our OSR-Lite mechanism. The switch implements both input and output buffering and relies on wormhole switching. The crossing latency is 1 cycle in the link and 1 cycle in the switch itself. The switch relies on a stall/go flow control protocol. It requires two control wires: one going forward and flagging data availability ("valid") and one going backward and signaling either a condition of buffer filled ("stall") or of buffer free ("go"). We assume the following parameter values in the architecture: 32 bit flit width, 6 flit output buffers and 2 flit input buffers. To note that different flit width and input/output buffer depth could be assumed while preserving the OSR-Lite mechanism implementation.

The switch architecture is extremely modular. A port-arbiter, a crossbar multiplexer and an output buffer are instantiated for each output port, while a routing module is cascaded to the buffer stage of each input port. We implement logic-based distributed routing (LBDR) [33] as described previously in this report. LBDR bits are computed by a central NoC manager and disseminated to the switch input ports through the dual control network. Indeed, two sets of LBDR bits are allocated at each router for OSR-Lite. Upon receiving the new routing bits, a router triggers the reconfiguration process by auto-generating initial tokens at its local input port (port connected to an end node) and processing the tokens accordingly.

The logic enabling the OSR-Lite mechanism was integrated into the above mentioned baseline switch taking care to preserve its modularity together with its performance. Thus, the OSR-Lite logic was designed in new modules plugged into the switch without affecting the existing blocks. Moreover, the new modules were instantiated for each switch port following the modularity of the baseline blocks (the OSR-Lite mechanism can be extended for switches of every arity by means of simple logic replication).

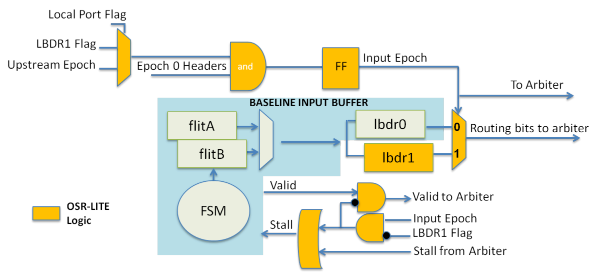


Figure 21. Switch input buffer enhanced with the OSR-Lite logic and a new set of routing mechanism.

OSR-Lite at the Input Ports

As a first step, the baseline switch was enhanced with a second routing logic unit (LBDR1) collecting the new routing info coming from the central manager. This unit is connected to the input buffer as the baseline LBDR0 block (see Figure 21) although is used exclusively for routing packets in the new epoch (new packets). The switch arbiters need to select the routing info from the appropriate routing logic block (either LBDR0 or LBDR1). This is obtained from a multiplexer configured by the current epoch of the input port (in a flip-flop). In order to reduce the reconfiguration latency, the input port evolves to the new epoch as soon as there are no stored header flits at the input port with the epoch bit set to zero (*Epoch 0 headers* signal) and the token has been received from the upstream switch (*upstream epoch* signal). Notice that in the case of the ports connected to end node (*local* port; *local port* flag), the token is assumed to arrive with the arrival of the new configuration bits (*LBDR1 flag*). In this case, the header flits located in the buffers are considered of the new epoch when the new configuration bits have arrived and the routing mechanism (LBDR1) is set. Notice that local ports do not introduce dependencies between channels that may lead to deadlocks, therefore is safe to assume all the injected flits as belonging to the new routing function. To notice that the token propagation will always start from local ports at switches, not involving end nodes.

The number of flit headers to be routed by LBDR0 and stored in the buffer is detected by a 2 bits counter monitoring the incoming and outgoing headers of the input buffer module. The counter increases its value when a header is accepted and the incoming token is low and decreases its value when a header is sent. In order to preserve the max performance of the baseline switch, sequential logic stages were exploited to avoid impacting the critical path in the OSR-Lite mechanism.

Notice that the implementation prevents possible race conditions from occurring. For instance, a token may be received from the upstream switch before the new routing bits are received. In that case, the header flits in the input buffers are stalled and declared not valid to the internal switch logic until LBDR1 is set.

OSR-Lite at the Arbiters

OSR-Lite requires a lightweight new module plugged around the baseline arbiters. The logic is reported in Figure 22. Basically, a set of *AND/OR* logic blocks together with a set of *EXOR* blocks allow the arbiter to process an incoming header exclusively when the epoch of the switch input port is the same as the one of the destination output port. On the contrary, a packet residing in an input port with the new epoch is stalled until the output port evolves to the new epoch (guaranteeing old packets go first and then new packets).

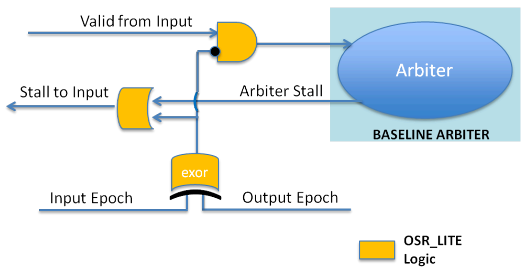


Figure 22. Switch arbiter enhanced with the OSR-Lite logic.

OSR-Lite at the Output Ports
Concerning the output port, an output port evolves to the new epoch when all the input ports with output dependencies to this output port have evolved to the new epoch. In order to efficiently deal with the dependencies, OSR-Lite takes profit of the routing bits used in LBDR. Routing bits indicate the routing restrictions that exist at neighboring switches. Therefore, they can be seen also as channel dependencies. If the R_{xy} bit is set it means that there is a link dependency between the output port *x* and the output port *y* at the next switch. On the contrary, if the bit is reset it means there is no dependency and in that case we can safely assume no packets will come through the port *x* requesting output port *y*. Therefore, the output port needs to receive both the epochs of the input ports and the routing restrictions located at the neighboring switches. The mechanism is enabled by a set of *OR* blocks (each of them belonging to a different input port) followed by an *AND* block, as represented in Figure 23.

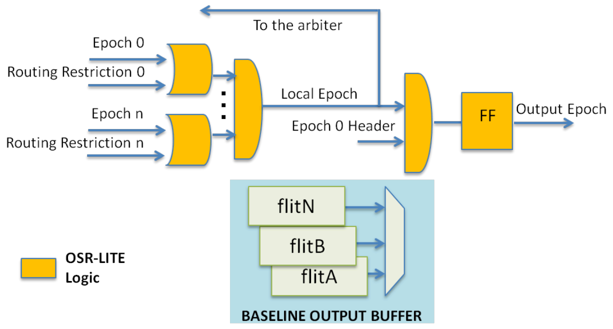


Figure 23. Switch output buffer enhanced with the OSR-Lite logic.

In contrast with the baseline OSR technique (where the routing restriction information was saved in the routing table), the OSR-Lite mechanism needs to obtain channel dependencies from the routing logic located at neighbor switches. As a result, three additional routing bits are sent by the LBDR0 logic of the upstream switch together with the token bit. To note that LBDR0 received its routing bits information through the control network in an earlier configuration stage.
In addition, the input port needs to send the incoming routing restriction signals to the appropriate output ports. Thus every link is extended by 4 additional wires (i.e. 1 token wire + 3 routing restriction wires). See Figure Figure 24.

Finally, the token is sent by the output port to the downstream switch when all the input ports with dependencies with the output port have evolved to the new epoch, meaning all these input ports have drained all the old packets from their buffers (see the *Local Epoch* signal in Figure 23).

Once the network has completely migrated to Epoch 1, the central manager can safely fill LBDR0 bits with a copy of LBDR1 bits, and instruct all the switches to safely swap to Epoch0 again. This allows for the system to be ready in few cycles for a new reconfiguration process.

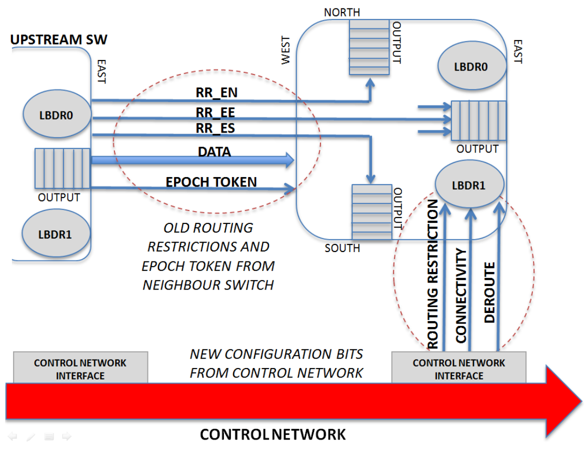

UPSTREAM SW
EAST
LBDR0
OUTPUT
LBDR1
RR_EN
RR_EE
RR_ES
DATA
EPOCH TOKEN
OLD ROUTING RESTRICTIONS AND EPOCH TOKEN FROM NEIGHBOUR SWITCH
NORTH
WEST
OUTPUT
EAST
LBDR0
OUTPUT
LBDR1
SOUTH
ROUTING RESTRICTION
CONNECTIVITY
DEROUTE
CONTROL NETWORK INTERFACE
NEW CONFIGURATION BITS FROM CONTROL NETWORK
CONTROL NETWORK INTERFACE
CONTROL NETWORK

Figure 24. Configuration information from neighbor switches and control network.

4.5. System-Level Evaluation

In this section, we evaluate OSR-Lite. First, we show how the OSR-Lite propagates over the network. Then, we evaluate the reconfiguration time overhead under different injection rates using synthetic traffic. Moreover, we compare the proposed reconfiguration with a static reconfiguration process in terms of network latency. Finally, we provide performance results by running real applications in a full system simulator environment.

Propagation

In order to simulate the reconfiguration process, we have modeled the OSR-Lite scheme in our event-driven cycle-accurate network simulator. A 8 x 8 mesh is used with wormhole switching (although the proposed method also works for virtual cut-through switching). Flit size is set to 4 byte and messages are 5-flit long. For the transient state, 50K messages are assumed and results are collected after 50K messages are received.

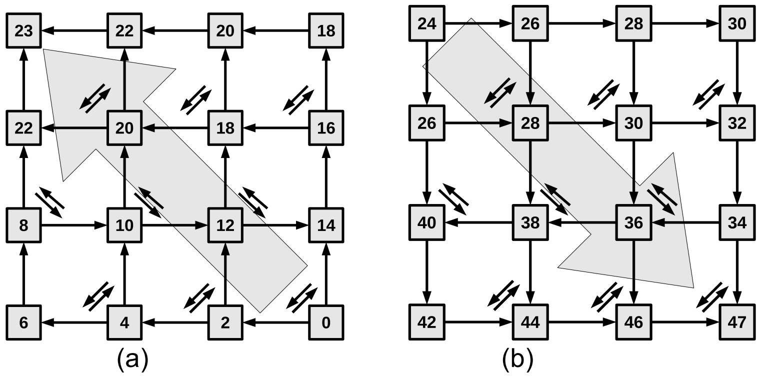


Figure 25. SR-Lite propagation over a 4 x 4 2D mesh topology: (a) scrolling up, and (b) scrolling down.

Figure Figure 25 shows how OSR-Lite tokens propagate over a mesh when there is no traffic traveling through the network. The diagonal arrows represent the bidirectional restrictions imposed by the routing algorithm (Segment-Based routing [34] in this case). In this figure, the numbers inside the switches represent the cycle when the token signal is propagated to its neighbors. Moreover, the arrows among switches depict the direction of the token signal propagations. As we can see, the token signals propagate among switches throughout the network in the order of the routing channel dependency graph, where Figure 25.(a) follows a scrolling up zig-zag direction, and Figure 25.(b) follows a scrolling down zig-zag direction.

When no messages are traveling through the network and a regular 2D mesh is considered then the number of clock cycles required for the OSR-Lite reconfiguration process is modeled by the following formula:

$$\text{PropagationTime} = (4xDx(D-1))-1$$

where D represents the mesh dimension. As we can see, it is a very fast process as the protocol uses only 223 cycles when a 8 x 8 mesh is considered. The high speed of the OSR-Lite reconfiguration process allows to perform frequent planned reconfigurations without affecting the integrity of the system operations. However, when there are messages traveling through the network the switches must drain the input queues of old messages before propagating the token signal as explained previously. This fact delays the OSR-Lite propagation depending on the network load. In the following, we analyze this effect taking into account different injection rates.

Time Overhead

In order to analyze the impact of the network load over the OSR-Lite reconfiguration framework, we have performed different simulations varying the injection rate. For each rate, we assume a constant packet generation rate for all end nodes. Moreover, in order to ensure that start-up instabilities do not affect our evaluation results, reconfiguration is not invoked until the network is completely stabilized. Figure Figure 26.(a) shows the performance obtained in a 8 x 8 2D mesh network under uniform traffic when no reconfiguration process is triggered. The figure indicates the three network injection rates that are used in the simulations. In what follows, the three rates are referred to as Low, Medium, and High, respectively.

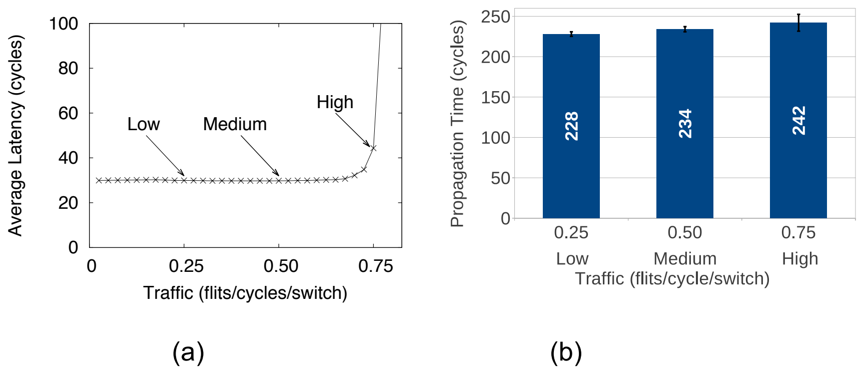


Figure 26. (a) Average message latency at different injection rates for SR routing on 8 x 8 2D mesh (b) OSR-Lite propagation over a 8 x 8 2D mesh topology at different injection rates.

Figure 26.(b) shows the number of cycles involved in the propagation of the OSR-Lite process taking into account the three different injection rates. Each bar depicts the mean of 30 simulations varying the seed. Moreover, we show the error bars that represent the 95% confidence interval. As we can see, the propagation time does not exceed 242 cycles for the High injection rate. Moreover, the difference between both the minimum and the maximum network loads is only 14 cycles, and therefore, the network traffic condition has a minimal effect on the OSR-Lite token propagation.

Finally, the contribution in terms of cycles for event notification (A), algorithm computation (B), configuration bits delivery (C) and OSR-Lite propagation (D) should be taken into account to determine the total latency for a complete reconfiguration process. In particular, (A) and (C) latencies depend on the position of the components to reconfigure with respect to the central manager. On the other hand, (B) and (D) latencies are related to the number of components to reconfigure and the traffic injection rate respectively. As an example, when we consider a 8 x 8 mesh then at most 66 cycles are required to cross the control network. Moreover, if 7 switches need to be reconfigured (i.e. the scenario of Figure 16) then 195 cycles are required by the computation algorithm in [31]. Finally, 242 cycles are spent by (D) in a High injection rate scenario. Summing up, the total amount of cycles for a complete reconfiguration process is the following:

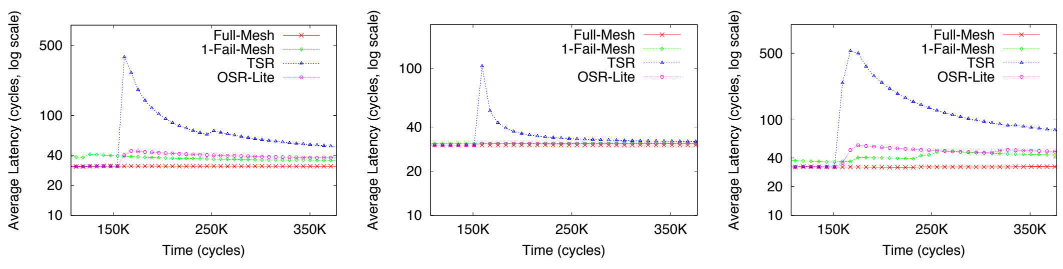
```
66(A) + 196(B) + 66(C) + 242(D) = 569 cycles
```

For dissemination of new LBDR bits to the switches, the dual network has to carry 17 bits per switch. However, not all switches need to be reconfigured, since the algorithm in [31] is able to evolve a system configuration into a new one while updating the minimum amount of LBDR configuration bits.

Comparison

In this section we compare the OSR-Lite protocol and the traditional static reconfiguration process (TSR). Figure 27 represent the average network latency respectively under hotspot traffic and uniform traffic with Medium and High injection rates, where both reconfiguration processes (OSR-Lite and TSR) are invoked after 150K cycles. Moreover, we have plotted two additional lines: the average message latency for the full mesh (Full-Mesh), and the average message latency for the mesh which has one link disabled from the beginning of the simulation (1-Fail-Mesh). Notice the y-axis is in logarithmic scale. Moreover, we have selected a random link in the 8 x 8 mesh as faulty. Under hotspot traffic pattern, 5 nodes are randomly chosen as hot spots which receive an extra proportion of traffic (30%) in addition to the regular uniform traffic.

The first observation is that both Full-Mesh and 1-Fail-Mesh obtain a different message latency. This is normal because the 1-Fail-Mesh suffers a latency degradation due to the disabled link. On the other hand, the two reconfiguration processes (OSR-Lite and TSR) start at the same time at the 150K cycle. At this point, the reconfiguration process moves from the Full-Mesh to the 1-Fail-Mesh topology. This effect can be estimated by the figures as the latency evolves from the latency obtained for the Full-Mesh to the latency obtained for the 1-Fail-Mesh. However, an important result based on the figures is that OSR-Lite performs the reconfiguration without degrading the obtained performance. In this case, the obtained latency grows up to the 1-Fail-Mesh line. Therefore, the latency is always near the maximum obtained with the 1-Fail-Mesh topology. In the TSR case, on the contrary, the latency is degraded due to the reconfiguration process overhead (need to drain the network). In the three cases, the latency grows above the 1-Fail-Mesh latency until it stabilizes. Specifically, in the Figure 27.(c) the latency of the TSR line grows to more than 500 cycles, and then stabilizes after 350K cycles. In this period of time, the TSR reconfiguration is degrading the obtained latency more than the link failure degradation produces. On the other hand, the OSR-Lite latency is upper bounded by the 1-Fail-Mesh latency.



(a) (b) (c)

Figure 27. Average message latency with (a) hotspot traffic and uniform traffic ((b) medium network load and (c) high network load).

Interestingly, the hotspot traffic and the uniform traffic with a High load have similar reconfiguration performance. Then, we can observe that the OSR-Lite has no impact on the message generation while the TSR process does. In fact, the TSR process increases considerably the obtained latency for all the cases. The main reason is that TSR queues all the messages at end nodes during reconfiguration while that need disappears in the OSR-Lite scheme.

Performance with Real Applications
In the following, we present performance results when real applications are used. In this case, we use a full-system simulator based on Simics-GEMS [35,36]. Regarding the on-chip network, we have used the same configuration as the previously detailed. Messages are 8-flit long for control messages and 72-flit long for data messages. As workload, we have used the PARSEC v2.1 benchmark suite [37]. Although we have used all the applications from the PARSEC v2.1 benchmark, due to the lack of space, we only show results for two applications: Blackscholes, and Streamcluster. In all cases, Simsmall input set has been used.

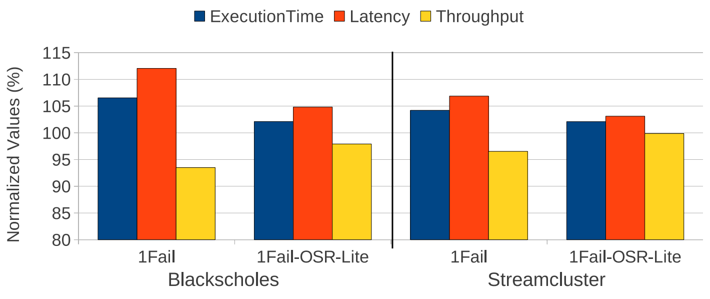


Figure 28. Performance with real applications.

Figure 28 shows the execution time, the network latency, and the network throughput. All the results are shown in normalized terms with respect to the results of a fully connected 4 x 4 mesh without link failures. The x-axis depicts the 1Fail topology (a 4 x 4 mesh topology with 1 link failure), and 1Fail-OSR-Lite that represents the same topology as 1Fail but, in this case, the OSR-Lite reconfiguration process is triggered in the middle of the application execution. Therefore, for this latter case, we pass from the fully connected topology to the 1Fail topology using the OSR-Lite reconfiguration. These results are shown for both Blackscholes, and Streamcluster PARSEC applications.

As we can see, the performance degradation is minimal for both cases (1Fail and 1Fail-OSR). Regarding the 1Fail-OSR case, obviously the degradation is lower because the link failure only affects half of the application execution, and the execution time degradation does not exceeds 3% of execution time overhead in any application.

4.6. Synthesis results

The implementation of a switch enhanced with the OSR-Lite mechanism has been compared in terms of area and routing delay with a switch based on the native OSR mechanism described previously and the baseline xpipesLite switch architecture [32]. The evaluation will demonstrate the infeasibility of the native OSR mechanism for an on-chip setting because of the need for VCs and the low scalability of routing tables.

For the experiments, an industrial memory compiler for a 40nm process technology was used to generate the memory macros required by the routing tables of the OSR mechanism. The switches together with their reconfiguration mechanisms were synthesized for the same 40nm industrial library.

Area Comparison
The description of the OSR mechanism in [11] focuses on the protocol details and it lacks of practical implementation details. Thus we exploited the information provided in [11] to model the OSR mechanism at RTL level and evaluate this latter solution in an on-chip constrained system. Especially, the OSR mechanism relies on 1 data VC supported by an additional control VC, and it adopts routing tables. As a result, we implemented the OSR mechanism into a 5 x 5 switch augmented with VCs by following the design techniques for area efficiency in [38] and we enhanced the switch with the 40nm memory macros to model the routing tables.

The 8 x 8 mesh topology is considered. Thus, 64 end-nodes are the total number of destinations in the system. When routing tables are used for distributed routing, each switch input port has a memory module with a number of words equal to the amount of destinations. Every word is composed of 3 bits, matching the switch radix. Given a destination ID, the switch selects the target output port based on look-up table. The minimum word width that the memory compiler, at the 40nm technology node, can generate is 4 bits. As a result, above all the available memory cuts, a single-port low-power RAM with 64 words of 4 bits was the memory cut showing the lowest routing delay and area footprint.

Finally, Figure Figure 29 shows the area footprint of this latter solution (the *OSR SW*) with respect to a baseline switch and our proposed solution (*OSR-LITE SW*). In particular, the OSR-Lite area overhead takes into account also the contribution of the control network carrying the information from the global manger to the routing mechanisms. For this purpose, we exploited the fault-tolerant control network proposed in [tecs].

The OSR-Lite reconfiguration mechanism requires a 14% of area overhead with respect to the baseline switch. This result is mainly due to the additional LBDR routing mechanism (+12%) contribute. On the other hand, the area overhead of the remaining reconfiguration logic is negligible when integrated into the switch.

Interestingly the OSR-Lite switch outperforms the baseline OSR switch: this latter requires approximately two times larger area than the counterpart solution. This result is mainly due to the severe area penalty introduced by the VCs and the 65% area saving achieved by the LBDR mechanism with respect to the routing table.

As a last consideration, the routing mechanism of the OSR-Lite solution scales with network size. In fact, while the memory macro suffers from increasing area and delay penalties, the logic complexity of the distributed routing algorithms does not depend on the number of destinations, hence it stays constant. Indeed, the distributed routing algorithms just grow with the switch radix.

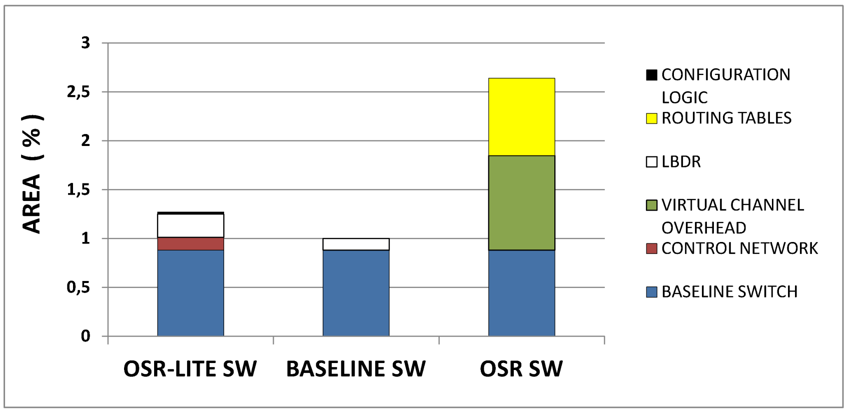


Figure 29. 5x5 switch (a) area comparison.

Routing Delay Comparison
In order to evaluate the effects of the OSR-Lite mechanism on the switch routing delay, we performed the 5 x 5 switch synthesis for maximum performance. The same experiment was repeated for both the baseline switch and the switch augmented with the baseline OSR mechanism. The OSR-lite switch and the baseline switch achieved a similar maximum operating speed of 750 MHz. The reconfiguration scheme was designed to avoid long critical path and preserve the baseline switch performance. Our OSR-Lite-enabled switch is thus capable of an at-speed reconfiguration.

On the other hand, the OSR switch is the 35% slower than our proposed solution. This result is mainly due to the intrinsic complexity added by the VC logic and the delay required to access the 64 words RAM routing tables.

5. Quality of service

One of the key features for effective virtualization in embedded systems is that of **providing APIs to application developers** in both the open and embedded virtualized environment to allow them to **negotiate with the hypervisor in terms of QoS/SLA**. This goal will be pursued within our project by considering a vertically integrated approach where suitable high-level constructs will be designed and implemented within a widely adopted and understood **parallel programming model such as OpenMP**. The compiler and runtime systems will be in charge of actually interacting with the hypervisor to ensure that the desired goal is achieved with the maximum efficiency in terms of different metrics (QoS, energy). In order to complete the picture, the link to the hardware is needed. In this project, a NoC infrastructure natively conceived for best-effort communications will be progressively augmented to also serve QoS traffic flows. As a guiding philosophy, QoS guarantees will be provided in terms of prioritized traffic (with reconfiguration capability of priorities to match the requirements of multiple use cases) and even of more aggressive circuit switching techniques.
As a main innovation with respect to state-of-the-art QoS NoC frameworks, vIrtical fosters two main approaches:
1- QoS for NoCs is impractically implemented in hardware only, due to the large and hard-to determine number of possible use cases at run-time. A proper mix of hardware facilities and software controlled management policies is vital to achieving efficient results.
2- In this project we are going to offer runtime QoS differentiated services by leveraging runtime reconfiguration of the NoC backbone. It is worth observing that virtualization and QoS are two tightly interrelated requirements for embedded systems. In this direction, the extension framework of NoCs for QoS will be synergic with the routing mechanism extensions for network partitioning and isolation illustrated above.
Given that, this section of the deliverable reports on a **soft-QoS package** made available by vIrtical to deliver QoS guarantees to network packets/flows, and that is structured into:

- *packet-level soft-QoS through priority-class round robin;*
- *flow-level soft-QoS through message-class VC allocation;*
- *full bandwidth reservation (circuit-switching).*

The implementation of the proper hardware support to materialize the soft-QoS package cannot overlook the highly dynamic landscape of the GPPA environment, where network partitions are allocated/deallocated at runtime. The deliverable therefore captures the fundamental interdependencies between the runtime reconfiguration support of the GPPA NoC and the need to preserve QoS provisions of running communication flows at any time.
In the future, the above items will be the tuning knobs in the hardware for the QoS requirements that the programming model will require through the hypervisor.

5.1. Packet-level soft-QoS through priority-class round robin.

One of most commonly used arbitration policies in NoC switches are Fixed-Priority and Round-Robin. The first is unbalanced by definition, indeed the contention is won always by packets from input ports that have the higher fixed priority. The second operates on the principle that a request which was just served should have the lowest priority on the next round of arbitration. Applying it to a single switch, we get that every packet is equally treated but this loses its validity considering a larger scenario, like a whole NoC or simply a part of it.

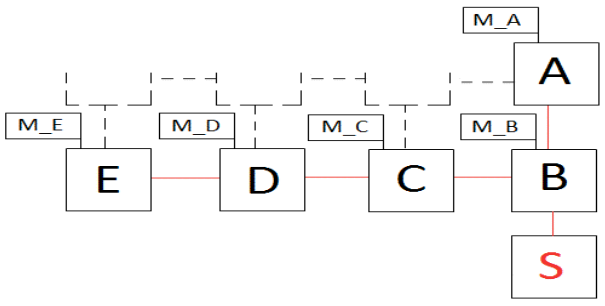


Figure 30: Round Robin policy loses its fairness scheduling switch_B traffic.

In Figure 30 many masters (M_A, M_B, M_C, M_D, M_E) want to approach the same slave resource (S). It's obvious that a Round Robin policy can't provide an equal sharing of the resource, because packets from north input port (from switch A) get the 33% of grants, the same for packets from M_B, and the remaining

33% is (unequally) divided between M_C, M_D, and M_E, all arriving from west input port, losing fairness. The above issues can be solved by combining the two arbitration policies into a hybrid policy. By construction, round robin corrects the unbalancing property of fixed priority, while fixed priority can offset the global imbalances that arise in a round-robin NoC. Weighted round robin is an example of such hybridized arbitration policies: thanks to weights, the available bandwidth is not equally split among contenders, but an heterogeneous bandwidth allocation is performed. We did not find this to be a good match with the requirements of a GPPA, where differentiated bandwidth assignments to switch ports are a bit innatural. In contrast, in a GPPA it is possible to identify packet types and traffic flows among which a priority ranking should be established. Within each service level, there is then no reason for establishing further priority rankings, therefore round-robin can be applied. The reason for establishing priority rankings in a GPPA can be manyfold. On one hand, we may have the need for functional differentiation of service classes. In this direction, instruction cache-line refills should be prioritized over data transfers not to stop execution, especially in an architecture where we have long physical paths to achieve the centralized L2, while data is predictably stored in local scratchpad memories. On the other hand, some inter-core or inter-cluster communications might be more critical than others based on the software knowledge of their exact meaning in the context of the application. Finally, priorities may be a way of fixing topology-dependent unbalancing effects, such as the case presented in Figure 30, by enhancing the priority of poorly served packet flows. At the same time, control packets associated with platform management (e.g., reconfiguration directives) clearly deserve a dedicated service class.
In vIrtical, a **priority-class round robin arbitration policy** was therefore developed and used, consisting of two-step arbitration: on the first step, requests are simply filtered based on their priority level. Then, round-robin is applied in case multiple concurrent requests exhibit the same higher priority level. The behavior is shown in the figures below.

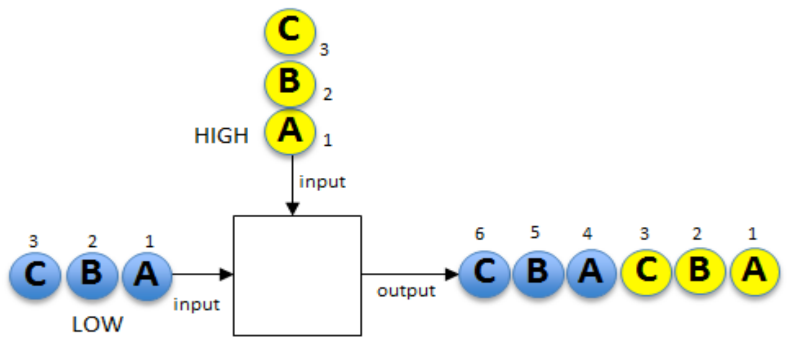


Figure 31: Arbitration in case of packets of different priority levels.

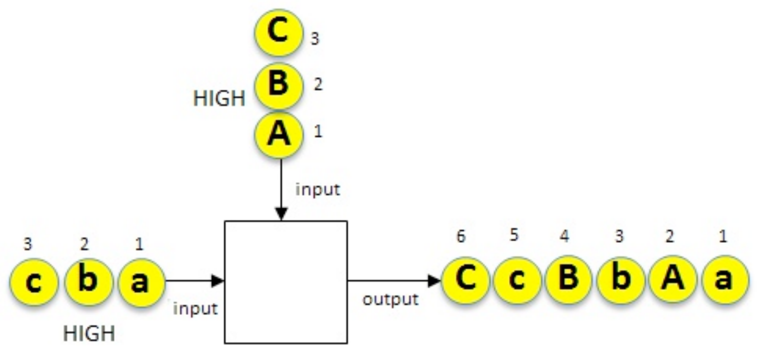


Figure 32: Arbitration fairness within the same priority level.

As shown in Figure 31, if the contention is between packets of different priority levels, the allocator gives the grant at packets with the highest priority, so they reach the output port first. The lower priority packets, instead, are stalled until the end of high priority traffic.
Figure 32 shows the case the traffic packets are at the same priority. In this case the allocator gives grant according to the classic RoundRobin policy that, in its basic form, is a simple scheduling, allowing each requestor an equal share of the access in a limited processing resource in a circular order.

From an implementation standpoint, to ensure the correct operation, we extend a Round-Robin based baseline arbiter replicating all the state registers of the arbiter on a per-priority basis. We also add a specific

algorithm to detect the priority of incoming packets, selecting the proper range of bits (3 bits needed to create 8 levels) out of the header flit of the packet, and furthermore, a state variable that stores the current priority level. In this way the requests, received by an arbiter, are filtered by the priority levels, serving first the highest level. So using this sort of priority detector, the contention for the grant is only about requests of the highest priority packets, then decreasing to lower levels. When there are no requests of higher priority, the contention passes to lower level requests.
Still using the state variable, the arbiter updates only the future value of new pending requests for a specific priority level, ensuring the maintenance of the classic RoundRobin circular order, but specifically for each level. Every *request* and *pending request* vector is replicated once for each priority level.
In order to set priority traffic in the system, we enrich the information of a packet adding a 3-bit QoS field embedded in each header, this way enabling up to 8-level priority schemes to classify different types of traffic within the system. Thus the level 0 has lowest priority and level 7 is the most prioritized (see Table 2).

PRIORITY LEVELs	Bits encoding
0 (lowest)	3 ‘b000
1	3 ‘b001
2	3 ‘b010
3	3 ‘b011
4	3 ‘b100
5	3 ‘b101
6	3 ‘b110
7 (highest)	3 ‘b111

Table 2: QoS encoding on header packets.

Using this priority assignement with 8 levels, we intend to serve both user-specified and architecture/topology-dictated priorities. The former ones are associated to the QoS requirements of the application (e.g., achieving a given processing rate), while the latter ones are associated to functional requirements (e.g., a low-latency platform configuration) or to structural imbalances (e.g., topological imbalances). In the next subsection, some experimental results are presented.

5.1.1. **Validation of the arbitration policy**

Now we present some functional validation tests of the priority-class round robin arbitation. The first experiment refers to arbitration inside a single network switch. Experiments were performed with an RTL-equivalent cycle- and signal-accurate SystemC simulation environment.

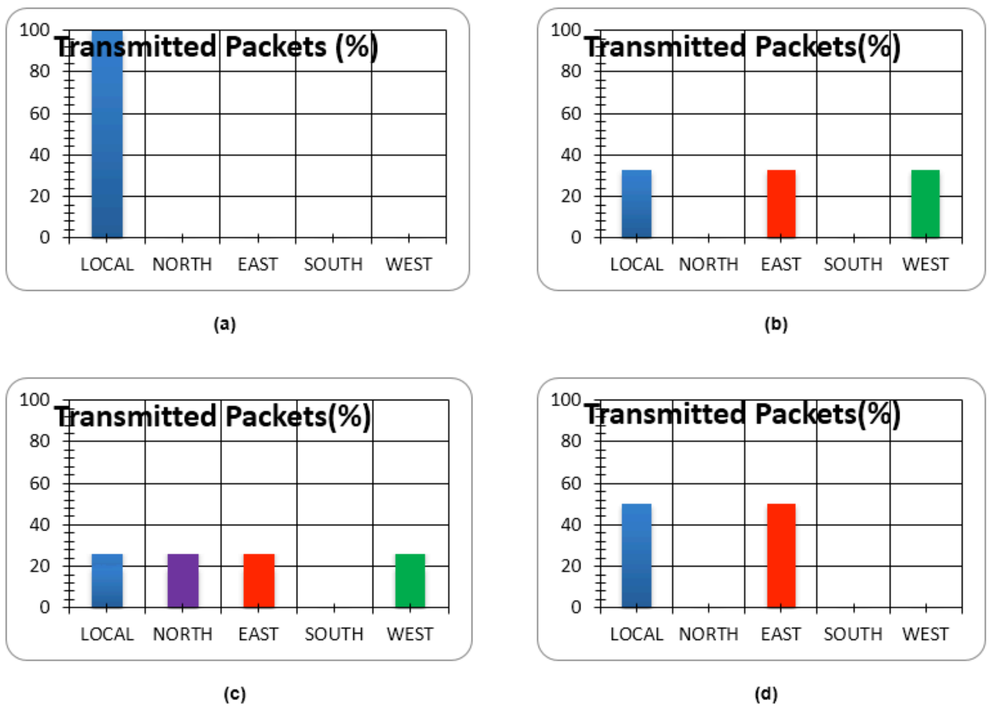


Figure 33: (a), (b), (c), (d): validation tests about the accurancy of a switch scheduling algorithm with priority class round-robin policy. The target output port is south (S), and the actual transmission rate at conflicting input ports is reported.

We consider a 5x5 switch where N, E, S, W, L are identifiers of north, east, south, west and local input/output port, supposing N, E, W and L are transmitting packets that want to reach the S output port. Based on packet priority we can have the following cases:

- Figure 33 (a): local input port is transmitting high prioritized traffic in a continuous way ("continuous" means that there is no idle time between a packet and the next one). So, even though north, east and west port want to transmit their packet throught the south port, the arbiter contention is won by prioritized traffic and the output port is reached only by traffic from local input port.

- Figure 33 (b): we suppose local, east and west input port have the same high priority level, and continuous traffic (north has lower priority packets). So the arbiter gives grant in a circular way to these ports, according to classic R-R policy, simply filtering the port with low priority traffic.

- Figure 33 (c): if all the input ports (N, E, W, L) have traffic at high priority level the whole arbitration behavior is aligned to a baseline Round-Robin scheduling and there is fairness about getting the grant between all input ports that are transmitting.

- Figure 33 (d): this is a particular case of situation shown in Figure 33 (a). In this case we consider local input port with high priority traffic and east port with a lower one (north and west port are not transmitting). If the idle time between packets of the higher level is enough to allow the trasmission of a packet of lower level, the situation is balanced as shown in the figure. In particular, considering packets composed by 3 flits-per-packet (header, payload and tail), the situation proposed is obtained considering idle time (*x*) $1 \leq x \geq 6$. If there is no idle time the situation is the same of Figure 33 (a), else if $x > 6$ the graph will be unbalanced in favour of traffic from the east input port.

Now we consider a larger scenario, a whole 4x4 NoC composed by 16 5x5 switches (north, east, south and west ports plus the local one connected to the master for switches between 0 to 14, and to the slave for switch 15), as shown in Figure 34. This scenario resembles the instruction cache refill network in the GPPA.

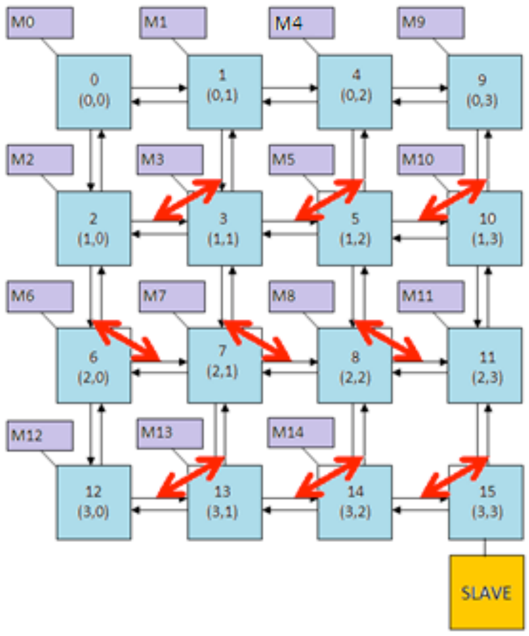


Figure 34: **A 4x4 NoC composed by 5x5 switches. Red arrows indicate the routing restrictions provided by the routing bits (RBITS) of the LBDR mechanism.**

We consider fifteen masters and one slave, connected to the local port of switch 15. All the masters inject traffic in the network, asking to reach the slave. As already argued in Figure 30, in this topology there is an unbalanced allocation of bandwidth because of the switch position in the topology itself: for instance, it's obvious that traffic injected by master 11 e 14 is privileged over any other, because of their position closer to the final destination.

Considering every master is injecting packets with the same priority level, with idle time between a packet and the next one of 25 clock cycles, stopping the simulation after 1000 packets arrived at slave 15, the results we obtain are shown in Figure 35.

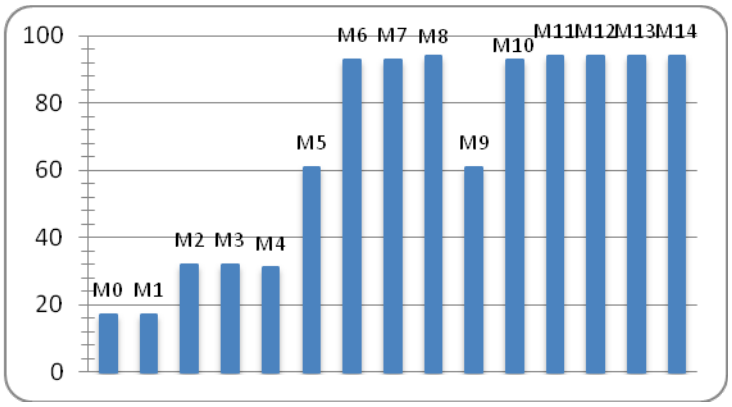


Figure 35: the graph shows how many packets have access to slave_15 after 1000 packets arrived, specifying on the x-axis their source master.

As we can see, this is a very unbalanced situation, and in the GPPA this would mean that some cores/clusters get more instructions to execute than others, causing a general execution misalignment.In order to restore fairness, we inject a prioritized traffic, increasing the packet priority of masters that are discriminated by their position. When we consider increasing priority as a function of the distance from the destination, then we get the fair bandwidth allocation of Figure 36. It should be observed that the achieved fairness also depends on injection rate. Clearly, high-priority nodes with high injection rates end up starving the others. However, this issue is mitigated by the traffic types that are served by the NoC in a GPPA. In fact, traffic flows concern cache-line refills and inter-cluster communications, that are relatively unfrequent events. For this reason, at this time we did not consider it worth taking the proper course of action against starvation. Later, upon actual GPPA implementation, this assumption will be validated and the architecture improved if needed (for instance, through virtual channels).
Finally, it is worth recalling that the large number of priorities we implemented (8) is such to suffice both for this kind of topology effects for the typical scale of a GPPA, but also to get some performance differentiation based on application criticality, and to get special service classes for control messages.

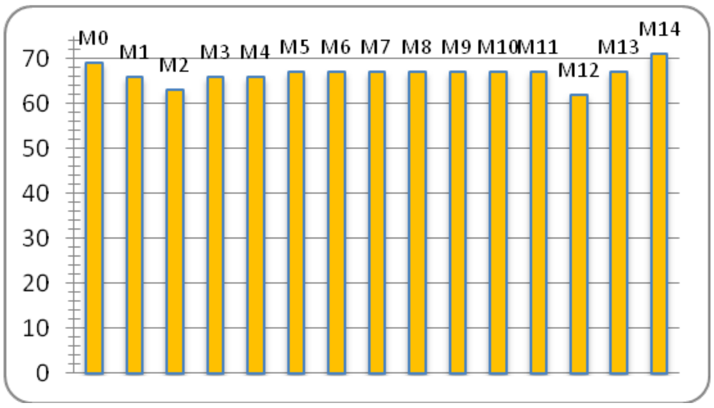


Figure 36: Fair bandwidth allocation through priority assignment.

5.1.2. **Implementation overhead**

Finally, in this section, we present the experimental measurements we performed on two 5x5 switches. The first switch makes use of an arbiter without any priority-class round robin support, while the second one implements the service classes. The reader should recall that the xpipesLite switch is considered as the baseline architecture, and that in this VC-less architecture one allocator is instantiated for each output port of the switch.
The measurements have been performed by synthesizing (with Design Compiler) both switches at max performance. We used a low-power, standard Vth 40nm industrial technology library (Vdd=1.2V) and we measured the area and the critical path delay.

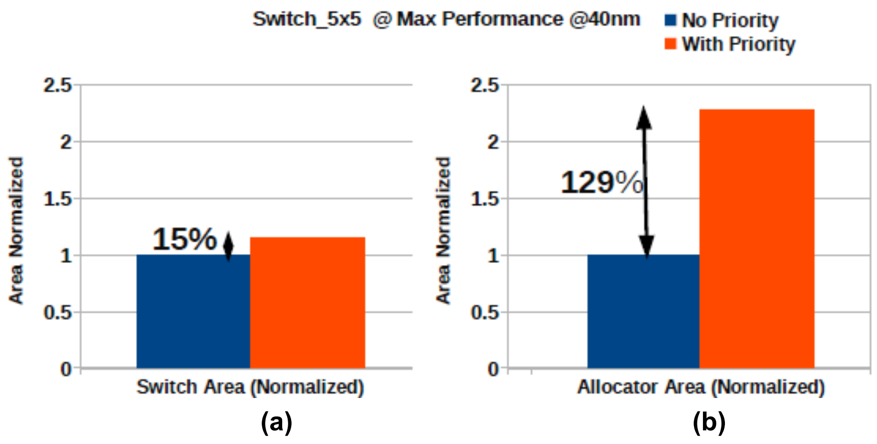


Figure 37 (a) Normalized area overhead at the switch level; (b) Normalized area overhead at the allocator level.

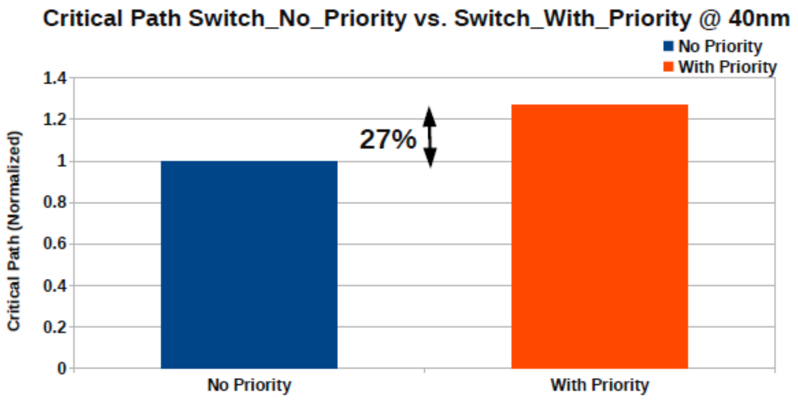


Figure 38: Normalized critical path delay overhead at the switch level.

Figure 37 shows normalized area overhead of switches (left side) and arbiters (right side). From Figure 37, it appears that the total area of the switch with priority-class round robin is 15% larger than that of the switch without any QoS support. This increase is due to the increased control logic implemented in the allocator, to the large number of supported priorities, and mainly to the need of replicating the allocator state on a per-priority basis. When we consider the allocator-level report, this overead is tangible. On the other hand, it is worth recalling that the xpipesLite switch used for comparison is the most lightweight switch one can ever find in a NoC, with just simple network functionality. Should the switch complexity grow (e.g., fault-tolerance, testing, runtime reconfiguration), then this overead would be rapidly absorbed. Also, with respect to the GPPA as a whole, it is reasonable to expect that the NoC has a marginal impact over the total area footprint.
As regards the critical path delay (see Figure 38), the switch with priorities is 27% slower than the reference one, since the allocator extensions go on the critical path. Overall, we could say that most of the overhead comes from the decision to enforce fairness among packets of the same priority class, not from the support of priorities themselves.

5.2. Flow-level soft-QoS through message-class VC allocation

As described in section 3.5, global network traffic in the GPPA (e.g.,for instruction cache line refills) could be accommodated in the on-chip network without any virtual channel, provided the routing algorithm for the local partitions and for the global communication does not change (only connectivity bits change). From a QoS perspective, this architecture design point can be extended with virtual channels just at the same for specific performance optimization goals. In practice, in vIrtical we will consider two virtual channels in order to reduce the interaction of intra-partition and L2 global traffic to NoC links only. One virtual channel (VC0) is used for intra-partition communications, while the second one (VC1) is used for L2 signaling. VC1 will also be used by the master port of the top-level NoC (the STMicroelectronics' STNoC) to write code and data to the L2 of the GPPA and/or data to the culster scratchpad memories. Some solutions in the high-performance computing domain (e.g., the Tilera manycore processor) opt for the extreme approach of full network replication, with each network dedicated to a specific traffic class. We do not go that far, since GPPAs lie at the boundary between SoCs and chip multiprocessors, and resource budgets are still constrained in this domain. In this context, we prefer to retain unified network links between the two virtual networks, while optimizing for their bandwidth exploitation through the virtual channel solution.

A further increase in the number of virtual channels may be due to deadlock avoidance issues. In fact, traffic toward the L2 consists of memory requests and memory responses, which is subject to protocol-dependent deadlock. The simplest workaround for this problem consists of having 1 virtual channel for memory requests, and another one for memory responses. For intra-partition communication, this is not the case, since it is in principle possible to have one-way communications only between clusters. **As an effect, in the GPPA we currently envision 3 virtual channels**:

- *1 for intra-partition communications*
- *1 for memory requests to the L2*
- *1 for memory responses from the L2*

5.2.1. Multiswitch virtual channel implementation

In vIrtical, we use a simplified yet efficient implementation of virtual channels.The conventional implementation style of virtual channels consists of multistage arbitration. Let us focus on the following extension of the xpipesLite switch taking this approach to implement virtual channels.

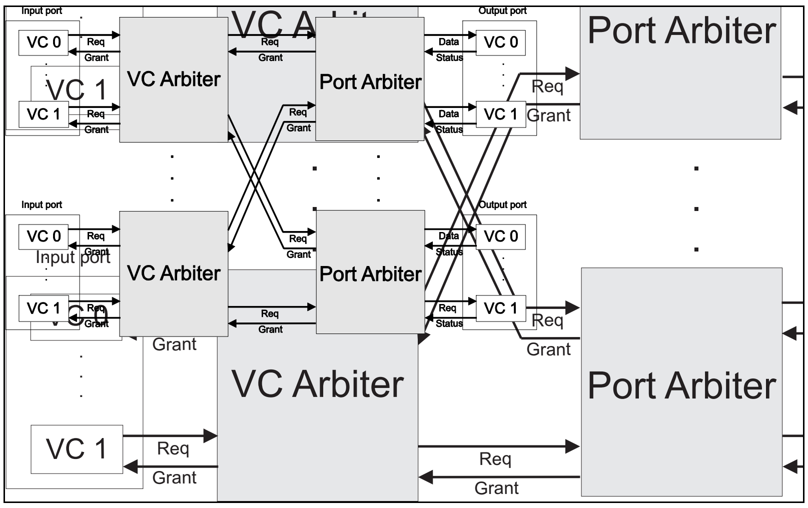


Figure 39: Multistage implementation of a virtual channel switch (2 VCs are showed).

For the sake of simplicity, the focus of this section is restricted to *statically allocated Virtual Channels (VC)* and to *deterministic routing algorithms*.The switch input port receives the virtual channel identifier (ID) together with the flit from the upstream switch. This ID is used to select

the virtual channel where arriving flits must be stored (Figure 39). Also, a stall signal is generated by each virtual channel and propagated upstream to the attached output port to notify availability of buffer space on a per-VC basis. Each virtual channel implements its own buffering space and a very simple LBDR decoding logic that computes the target output port. Switch allocation is performed immediately after the flit arrives, and the routing information is used to identify the intended switch output port. VCs are assigned nonspeculatively after switch allocation: the winning VC that is granted access to a given output port automatically reserves the VC with the same ID at that output port. This is because VCs are statically allocated. As will be clarified shortly hereafter, it can never occur that a VC is granted access to an output port and the intended VC at that port is occupied.
Switch allocation can be performed with a separable input-first allocator. Since allocation requires 2 stages of arbitration we call it the *multistage architecture*. One rule that is enforced during switch allocation is that a flit, either head or body flit, can only win the arbitration in the first stage if it requests an output VC that has free buffer space and is not in use by another input VC. In practice, the first stage arbiter filters the requests for nonfree output virtual channels. This way, it is not possible to waste a cycle by selecting a winner in switch allocation that will find its target virtual channel reserved or with no space. To provide fairness among all the input virtual channels, if the winner of the VC arbiter does not win the port (second-stage) arbitration, it receives the highest priority in the virtual channel arbiter. This guarantees that the last winner will be proposed again as soon as possible.

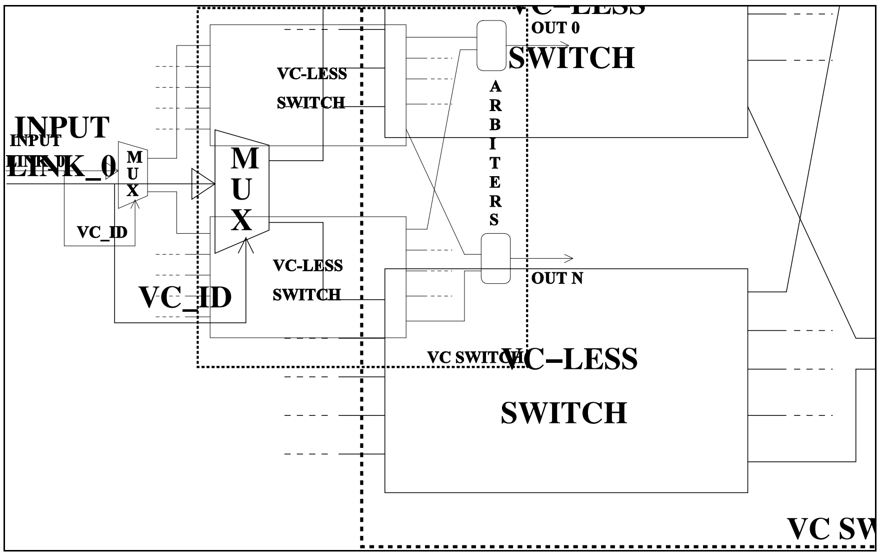


Figure 40: Multiswitch implementation of a virtual channel switch (2 VCs are showed).

An alternative VC switch architecture consists of replicating not just buffers per channel, but rather the entire baseline VC-less switch as many times as the intended number of virtual channels (Figure 40). Replicated switches then share the same physical input and output links, similar to what conventional VCs do, but with the main difference that in the new implementation VCs have their own access to a replicated crossbar and the first stage of arbitration can be finally removed. This solution will be denoted as the *multiswitch VC implementation*. The underlying principle is simple: instead of replicating buffering resources inside a switch, the idea is to replicate the baseline VC-less switch without impacting its internal critical path. Similar to the multistage architecture, also this solution requires an additional stage of *link arbitration* in order to multiplex the outputs of the baseline VC-less switches into the same physical output links connecting to downstream switches. As Figure 40 indicates, this stage is cascaded to the replicated VC-less switches it arbitrates on a flit-by-flit basis while the arbiters of the replicated switches keep arbitrating at the packet level. Interestingly, delay of this arbitration stage does not add up to that of the VC-less switches to determine the critical path, since they are separated by a retiming stage (the switch output buffers). In practice, *the critical path of the multiswitch architecture is the same of a VC-less switch*, since it does not make use of a multistage arbiter. However, one might argue that this comes at the cost of replicating more physical resources (e.g., the crossbars). At this point, a basic principle of logic synthesis comes into play and leads to opposite conclusions. When comparing the multistage with the multiswitch VC implementations, this latter has less functions on the critical path, hence potentially resulting in a more area/power-efficient gate-level netlist after logic synthesis. In fact, the multiswitch architecture certainly provides a higher maximum speed than the multistage one. However, if we require the two architectures to be aligned to the speed of the slowest one (the multistage), then combinational logic of the multiswitch design can be inferred with relaxed delay constraints and therefore thoroughly optimized for area and power. In practice, a different design point along the area-performance optimization curve can be inferred.

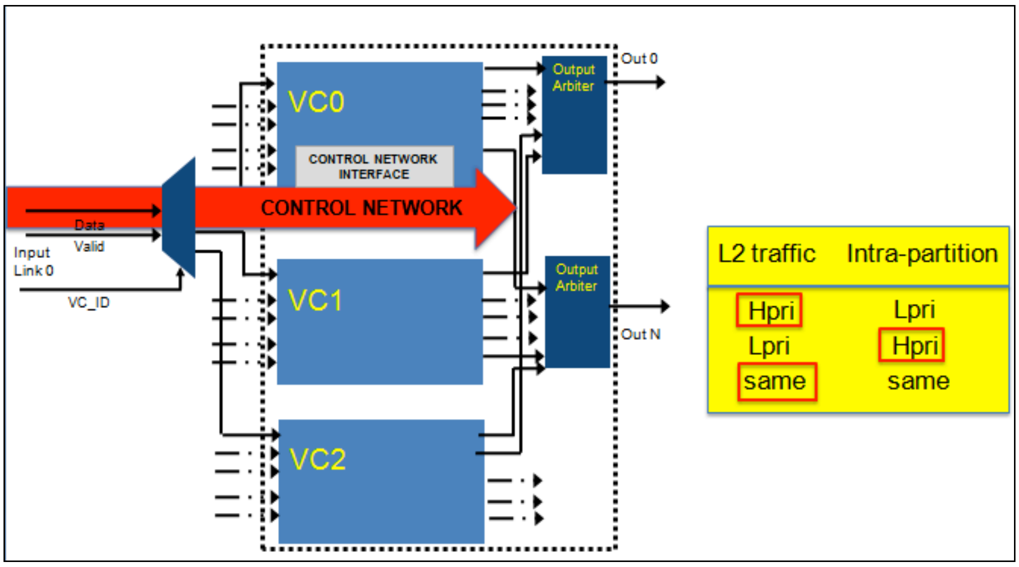


Figure 41: Multiswitch VC switch of the GPPA, with control network for selective virtual channel reconfiguration.

5.2.2. **Specialization for GPPA**

vIrtical will adopt the multiswitch implementation of a virtual channel architecture. When considering the distinctive features of the GPPA, we end up in the final architecture of Figure 41, which is a novel contribution of vIrtical. The figure shows a virtual channel switch, implemented by VC-less switch replication. At this point, it is worth recalling that VC0 is used for intra-partition messaging only. As such, this is the only virtual channel that needs to implement the runtime reconfiguration of the routing function. In fact, the number of partitions change over time. In contrast, the other virtual channels enable global traffic, hence its routing function does not undergo any runtime reconfiguration (unless we want to consider the possibility of dynamic rerouting to account for possible malfunctions that might show up at runtime). Given that, the figure shows a control network which is connected only to the top virtual channel: it brings the control signals of the OSR-Lite reconfiguration mechanism, selectively to the intended virtual channel/VC-less switch. As dictated by section 3.5, the three VC-less switches might have the same routing restrictions but different connectivity bits. ***However, in the more flexible scenario of section 3.6, the three switches might actually work with different routing restrictions as well***. In fact, the proposed architecture already routes local and global traffic across different virtual channels, hence preventing the occurrence of deadlock due to mixing up of packets with different routing algorithms in the same buffers. Whenever full bandwidth reservation needs to be delivered (see section 5.3), then the architecture in Figure 41 still holds provided VC0 has its own links in addition to buffers. In practice, we would need a multi-network solution, as illustrated shortly hereafter. VC1 and VC2 would then belong to a separate network than the one of VC0.

Finally, the output arbiters were set to implement the following arbitration policy:

- when conflicting packets from VC0 and VC1/VC2 have the same priority, VC1/VC2 is prioritized (to enable prioritized instruction cache line refills)

- when conflicting packets from VC0 and VC1/VC2 have different priorities, then the VC with the highest priority is prioritized.

Switch Type	Combinational Area	Sequential Area	Total Area	Path-Delay
VC Multi-Switch (@max-perf)	1.5	0.96	1.14	0.71
VC Multi-Switch (@relaxed)	1	1	1	1
VC Multi-Stage (@max-perf)	1.43	0.91	1.09	1

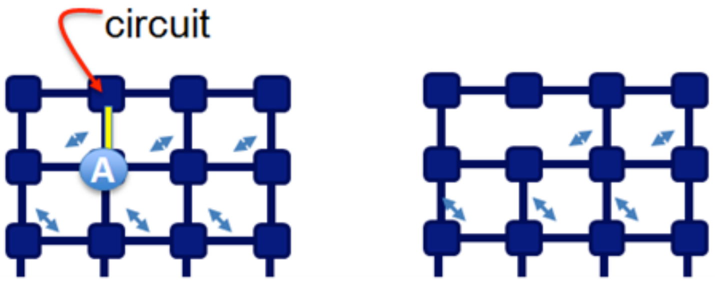
Table 3. Area and power overhead of VC implementation styles.

5.2.1. Implementation overhead

The issue is to determine whether the area savings achieved by logic synthesis are enough to compensate for the larger amount of hardware resources that are instantiated in the multiswitch architecture, especially the replicated crossbars. Please observe that the multistage and the multiswitch architectures can be designed to instantiate the same overall amount of buffering resources: N VC queues in the multistage switch are equivalent to a single queue in N replicated switches. Table 3 summarizes the area/critical-path of two 5x5 virtual channel switch (VC multiswitch and VC multistage) synthesized with the same 40nm technology library. The designs were synthesized at their maximum performance first, then the delay constraint was gradually relaxed, thus getting area/critical path results as illustrated in Table 3. From the column five of Table 3, it appears that the multiswitch architecture can achieve a higher speed (29%) than the multistage one since it implements less control functions on the critical path. Therefore, the physical synthesis tool can reduce the area of this design by 14% while relaxing its performance constraints. It is then possible to match the same maximum speed of the multistage architecture, while incurring a lower area, since the area scalability process for the internal combinational logic (e.g., the crossbar) is very effective. The area saved by the multiswitch amounts to roughly 9%. Most of the extra area of the multistage comes from its combinational logic (43% with respect to the multiswitch one). The second and the third columns show respectively the area breakdown of combinational and sequential logics of both switches at max performance. For high performance applications, the multiswitch VC results to be the best choice since it is able to run at higher frequency than the multistage one. Moreover, when the multiswitch is relaxed and runs at the same operative frequency of the multistage one, it leads to save approximately 9% of the area.

5.3. Full bandwidth reservation

Circuit switching is easier to implement in our target GPPA than in standards NoCs because no setup/teardown procedures are required: once a partition is configured, ***circuits inside that partition can be automatically pre-configured as well***, and run till completion of the partition. ***Given the implementation mechanism of circuits we will propose, circuits can be dynamically allocated or tore down at partition runtime exclusively by means of runtime reconfigurations of the routing function (OSR-Lite based, in our case).***



(a) **(b)**

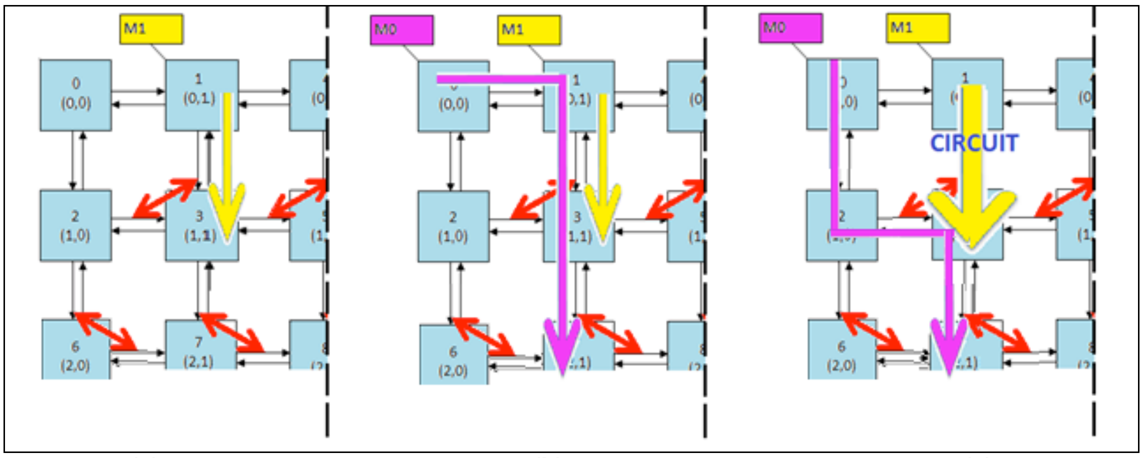
Figure 42: (a) visibility of A, to which a circuit is reserved, vs. (b) visibility of the other nodes.

Observing* Figure 42 *(a) and (b), the key philosophy we adopted to implement circuit switching emerges, that is viewing reserved links and buffers in the same way as broken, hence unusable components, except for the traffic flow which the circuit is dedicated to.
Viewing a circuit “as broken links” means that the connectivity bits of the adjacent switches are set to zero, so they don’t consider the path through the circuit reserved resource in their routing computation. Only the switch A can “view” the link as non-broken. In practice: packets issued by A and heading to the destination served by the circuit, should be routed across the circuit. All other packets, although heading to the same destination, should be routed elsewhere.
From an implementation standpoint, we get an overhead of **11 additional bits for each input port**, needed to expand the LBDR routing mechanism: if the destination the packet wants to reach is the circuit destination (coded by the 11 bits), the routing logic forces the output port expected by the circuit (coded by the 11 bits), else the packet is routed through other output ports because the link is seen as broken.
8 bits are used to encode a destination address (4 for the y-axis coordinate and 4 for the x-axis coordinate) and 3 bits to give the information about the output port the packet has to be sent (3 bits are enough because we are considering a 5x5 switch).
Please notice that, similarly to the case where a link or a switch sub-block is excluded from routing paths because it is defective, also in this case the insertion of a circuit affects the routing algorithm of the partition the circuit is inserted into. That is, routing paths should not go through the reserved circuit. If the partition is created on top of idle resources, the algorithm will be custom tailored to accommodate the circuit from the ground up. If a circuit is created at runtime in a running partition, then the algorithm should be reconfigured through the OSR-Lite process. Ultimately, this requires a selective modification of routing bits and connectivity bits, where thanks to OSR these changes can be operated without draining network traffic. **This brings to the key novelty of the vIrtical approach: the establishment of a circuit at runtime is equivalent to the runtime reconfiguration of the routing function.**
In terms of architectural implications, circuit support impacts the virtual channel architecture in Fig.41. In fact, circuits would be established across the input/output connections of VC0 switches (those used for inter-cluster/intra-partition communications). At the same time, corresponding inter-switch links would be reserved. However, such links are shared with virtual channels for global traffic (VC1 and VC2), which would experience blocking for the time of circuit reservation. To prevent this, the solution is twofold:

- **we may enforce runtime reconfiguration of the routing function of VC1 and VC2 as well, unlike Fig.41. This way, the link is seen as “broken” by all virtual channels, which would then implement the suitable course of rerouting action.**
- **we may design two different networks: one for intra-partition communication (i.e., VC0 would become a separate network by adding inter-switch links), and one for global traffic (which may still implement virtual channels for requests and responses). This way, full bandwidth reservation on one network would not affect the other network. At the same time, the global network would not need to support runtime reconfiguration of its routing function, thus following the philosophy of Fig.41.**

5.3.1. **Functional validation**

To validate our implementation we propose the test in the figures below.



(a) (b) (c)

Figure 43: Runtime reconfiguration to enforce a circuit. (a) Master M1 is active and is transmitting packets destined to slave connected to switch 3. (b) M0 starts to inject traffic with switch 7 as destination, so routing algorithm routes this traffic through the path of traffic from M1. (c) After a reconfiguration a circuit is enforced: traffic from M0 changes the routing path because it sees the link as reserved.

We consider a runtime reconfiguration to enforce a circuit. As shown in Figure 43 (a) master M1 starts to inject traffic to slave connected to switch 3. Figure 43 (b) shows when also M0 starts to inject traffic to slave 7 as destination: in this case the traffic is routed to the M1 traffic path and so participates to a Round-Robin scheduling (no packets priority is specified) and the grant is given in a circular way. After a runtime reconfiguration with the OSR-Lite mechanism, as shown in Figure 43 (c), a circuit is created between switch 1 and switch 3 and so the link reserved to M1 traffic because the other switches consider the link as reserved/broken (we set to zero specific connectivity bits of LBDR-bits to mimic the fault). At the same time, through OSR routing bits are selectively modified to accommodate the new routing paths resulting from the exclusión of the reserved circuit for most cores.

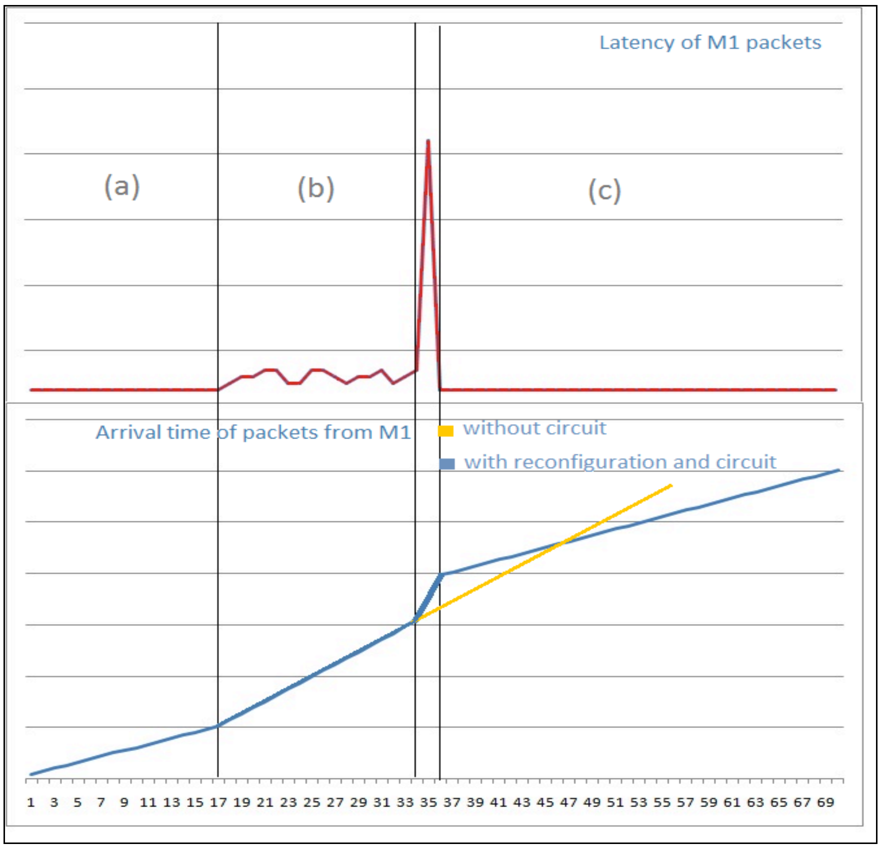


Figure 44: according to Figure 43 (a), (b), (c) we show in red the latency of M1 packets and in blue/yellow the arrival time of packets from M1 with a runtime reconfiguration to create the circuit and without the circuit.

The experimental results reported in Figure 44 show latency and arrival time of packets from master M1. We note that in (a) latency is minimal and constant and also the gradient of the arrival time curve, because there is only M1 injecting traffic. In (b), when also M0 starts to inject traffic, the gradient and the latency increase because of the contention (to reach the south output port in switch_1) between local port traffic (traffic from M1) and west port traffic (traffic from M0). In (c), after a runtime global reconfiguration, obtained by OSR-Lite, the graph shows a decrease of the gradient of the curve and a latency that returns to the level seen in (a) because the instauration of the circuit allows packets from M1 to go directly to slave of switch 3, without Round-Robin scheduling: the path is reserved by creating the circuit. Accordingly, the traffic from M0 is routed through another path, because the link is seen as broken.
Noteworthy, the time needed by the reconfiguration to create a circuit is absorbed very soon because the gradient of the curve decreases: in Figure 44 we can see in yellow the arrival time if there is no reconfiguration. It mantains a higher gradient, because of the arbiter contention, and intersects the blue curve, thus giving users the reference for convenient reconfiguration.

5.4. QoS-aware Runtime Reconfiguration

We consider now the reconfiguration process. As the reader may recall from the OSR-Lite mechanism, the reconfiguration process involves the whole network, and the whole traffic running over it. A switch, waiting for the token arrival, stalls the new traffic and only when the token arrives and gets forwarded through an output port, it proceeds to processing new packets through that output port. So if there is running traffic, a switch is to some extent affected by the reconfiguration process, though there are no changes to the configuration of its specified LBDR_bits. ***This questions the possibility of meeting QoS constraints in an operational environment where lots of runtime reconfigurations take place.*** To give an applicative example, we consider the figure below.

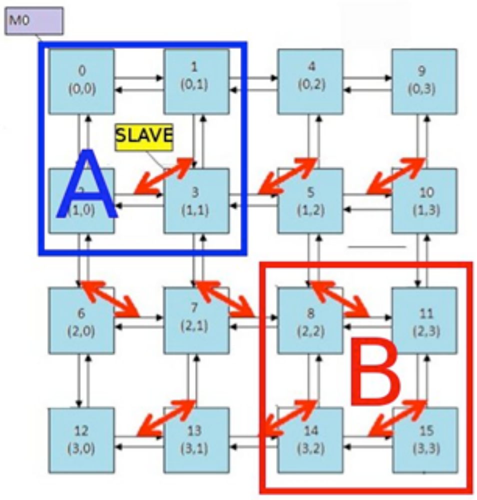


Figure 45: running traffic in a partition "A" of the network, involved by a reconfiguration process done to create another partition "B". Red arrows represent routing restrictions of the network

Figure 45 shows a 4x4 NoC where there is running traffic. In particular we consider a specified partition (partition "A"), composed by switches 0,1,2,3, where master M0 is injecting traffic in a continuous way to reach slave S3. If we want to create a new partition "B" (for example including switch 8,11,14,15), launching a new OSR-Lite reconfiguration process also the running traffic in the first partition will be affected to some extent. In principle, this should not be required because nothing changes in the LBDR_bits configuration of the running partition "A". Figure 46 shows how running traffic in partition "A" is affected by a global reconfiguration process vs. an ideal local process to partition “B”. This is the result of an RTL-equivalent SystemC simulation.

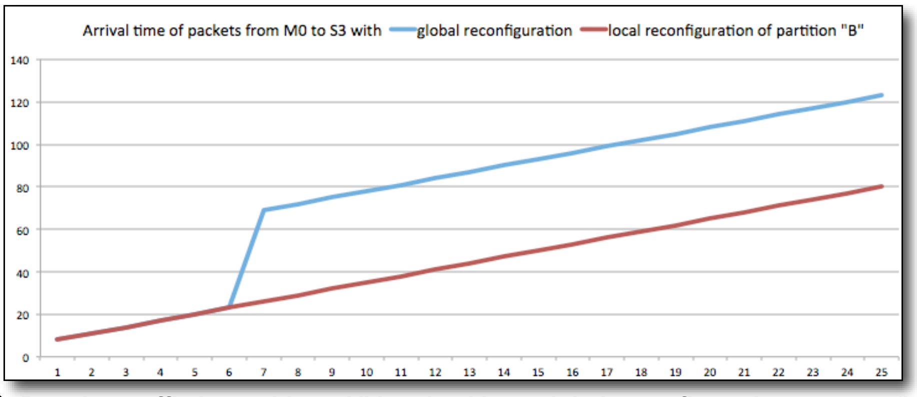
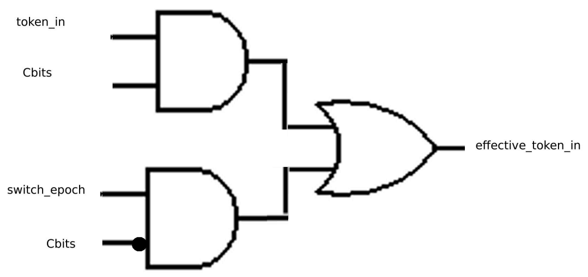
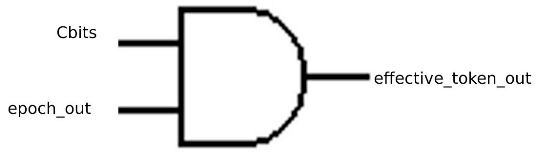


Figure 46: Running traffic in partition “A” involved by a global reconfiguration process (blue) vs local reconfiguration process (red) upon new partition creation. The plot shows the arrival time of packets to their destination. On the x-axis there is the packet identifier.

We can clearly see that arrival times are affected in a very significant, although localized, way because switches involved by the reconfiguration process have to wait for the token. In particular the reconfiguration process impacts switch_0 for about 46ns (considering 1ns = clock cycle).
The solution to this problem is **local reconfiguration** that, as we can see in Figure 46 (red curve), does not affect the existing partition and its running traffic. Indeed, the reconfiguration in this particular case should concern only switches of future partition "B" because it's here where routing configuration bits change.
The mechanism behind local reconfiguration we implement in our network is very simple. We give new LBDR_bits and LBDR_en (enable signal to inform the switch that LBDR configuration is changing) only to switches involved by the new partition but at this point an evolution of switch reconfiguration control logic becomes necessary since the token propagation must be limited to the new partition area.
The figure below shows the logic behind the performed modification.



(a)



(b)

Figure 47: improvements of logic behind (a) epoch_input, (b) epoch_output, to control token propagation in case of local reconfiguration.

In Figure 47 (b) we consider the simple way of constraining token propagation. Simply, considering *epoch_out* as the signal that represents the *token* propagating signal in the baseline solution, now we filter it with an AND gate and *Cbit* signal, that represents the connectivity bit that give the information about the presence of an active link between two interconnected switches: *effective_token_out* signal goes high (*effective_token_out* = 1) only if *epoch_out* = 1 and *Cbits* = 1, so there is an effective token propagation only if there's a valid link that connects the output port to another switch. In contrast, in a boundary switch, or when the output port is toward a partition edge, the token propagation is blocked.
In Figure 47 (a), instead, we consider part of the OSR-Lite logic that controls epoch transition for partition boundary switches. Considering *switch_epoch* the signal that goes high when the LBDR enable is sent (i.e., new LBDR bits received from the control bus), the key idea is to mask the *token_in* signal by an AND block with *Cbits*, and in the same way to filter *switch_epoch* with the negation of *Cbits*. So if a token arrives (*token_in* = 1) and input port is linked to another switch (*Cbits* = 1), the token passes. The OR block, instead, is useful to maintain the correct behavior of the logic: if a token arrives from an active link or there is a pending reconfiguration (*LBDR_en* enabled, so *switch_epoch* = 1) and the port has no connection (!*Cbits* = 1) the *effective_token_in* signal goes high. So the switch port sees an effective token if there is a valid *token_in* (from an active link) or if a reconfiguration is launched and the port is not actively connected.
The conclusion of updates of epoch_input and epoch_output is that the token propagation is effectively limited to the part of the network where we want to create a

new partition: a non-valid token in input is not considered, a non-valid token in ouput is absorbed by the partition edges, so local reconfiguration becomes possible.
These improvements allow also a local reconfiguration applied to an existing and running partition, ensuring that will be limited to its switches. This can be the case of a change of routing algorithm, of the dynamic setting of a circuit, of working around a faulty link. Local reconfiguration for a running partition is more efficient than a global one, as shown in the experimental test represented in figure below.

5.4.1. **Functional validation**

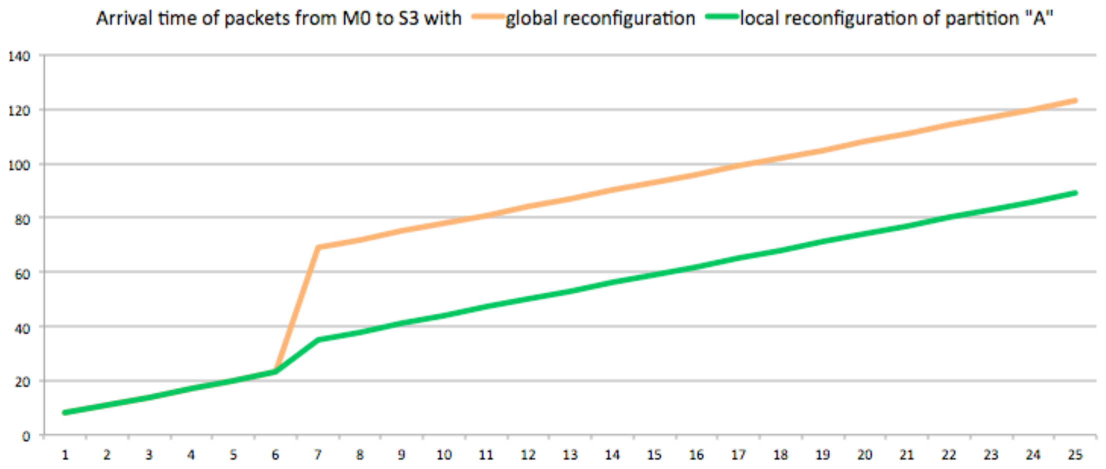


Figure 48: Running partition reconfiguration: running traffic involved by a global reconfiguration process (orange) vs local reconfiguration process (green); arrival time of the packets to their destination is showed.

Let us consider a scenario similar to Figure 45. If a new partition “B” is allocated, then with the proposed approach traffic inside “A” is not impacted. Therefore, let us consider the more interesting scenario where only partition A is running, and its routing function needs to be changed at runtime via an OSR-Lite reconfiguration. Figure 48 shows how arrival times of running packets in partition “A” are improved by reconfiguring the same running partition "A" locally: if we set a global reconfiguration process, traffic blocking is correlated with the reconfiguration of the whole network, while a local partition implies token propagation only inside the partition under reconfiguration. In particular, traffic at switch_0 is stalled for about 12ns during local reconfiguration, as opposed to 46ns during global reconfiguration, highlighting a very significative improvement of about 72%.

Considering the whole new outline, OSR-Lite allows not only global reconfiguration but it is also useful to set intra-partion reconfigurations, necessary to bypass a broken link, to create a circuit and to change the routing algorithm of a running and existing partition, without affecting other partitions, and minimizing the reconfiguration transient.
Moreover it can be used to create new partitions from idle resources, too, in case of loose synchronization with the software. In fact, assuming idle resources, there are two options. On one hand, the network manager might configure the routing mechanism of the network section involved by a new partition, and once finished trigger software execution on partition IP cores. In this case, token propagation is not strictly needed, since there is no ongoing traffic. On the other hand, there might be loose synchronization: IP cores try to inject traffic regardless of the state of the network. This latter, in turn, will prevent such traffic injection at its switches via backpressure until they progressively mígrate to the new epoch. In this case, not all the switches start collecting packets at the same time. OSR-Lite would be the indirect synchronization mechanism between network state and IP core execution in this scenario.

5.4.2. **Dual-network design for high-performance reconfiguration**

Until now, referring to OSR-Lite reconfiguration, we considered every LBDR_en provided in a synchronous way, i.e. at the same time for all the switches of the NoC. This was done since the focus was on functional validation.
In this subsection we relax that assumption and consider a more realistic setting, where control bits of the OSR-Lite reconfiguration process are brought by a dual network. Such a dual bus may be another 2D mesh superimposed to the main one, however this would be too much of an overhead. We more realistically envision a global ring topology connecting all the switches of the main NoC in a row. A global GPPA manager is one node of the ring, and also closes it.
The interesting issue that we intend to address here is that the ring may connect the switches of the main NoC based on different patterns. The issue to investigate is which of these patterns best matches the token propagation pattern of the OSR-Lite mechanism.

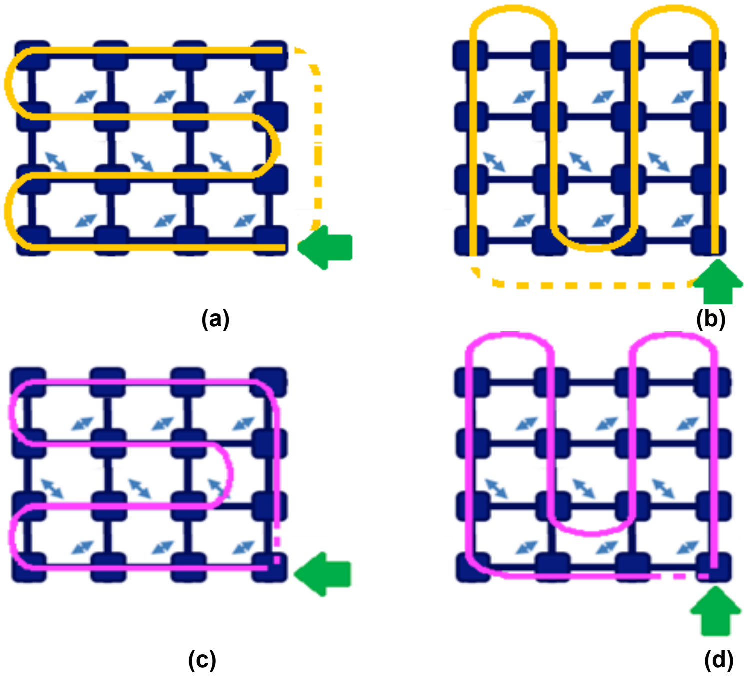


Figure 49: different paths to provide LBDR_en by dual-bus. (a),(b) are routing inefficient paths; (c), (d) are layout aware paths. Green arrows show the starting point and the direction of each path.

As Figure 49 shows, we provide the OSR control bits to the switches using four different paths. The yellow (a) and (b) paths are theoretically sound but in practice they end up in very inefficient implementation. In fact, the return path of the link would be most probably multicycle in 40nm technology (and below), due to the its long length. Pink paths (c) and (d) instead are layout aware paths.
Figure 50 shows the experimental results measuring the global reconfiguration time of the whole network (as though the whole network was a unique, global partition that gets reconfigured on-the-fly), according to different paths proposed in Figure 49. Let us recall that what changes in all cases is the order in which OSR control bits are fed to the switches of the main NoC.

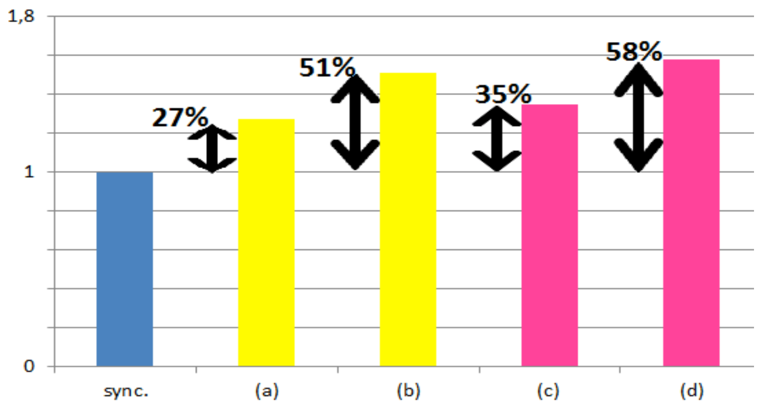


Figure 50: reconfiguration time with dual networks of kinds (a), (b), (c), (d) of Figure 49, compared and normalized with respect to the ideal, synchronous feeding of OSR control signals.

The best case is the synchronized one (75nsec considering a clock period of 1nsec), although irrealistic. This would imply the possibility of inferring a single cycle one-to-many connection between the global manager and all network switches. Yellow path (a) turns out to be the most efficient one, since it tries to match as much as possible the token propagation pattern across the network. This pattern is especially convenient for top left partitions, since LBDR_en signal that causes a new_epoch arrives closest to the arrival of all the token_in signals for those switches: new_epoch packets are stopped for a smaller time. In case yellow path (a) could not be physically implemented, then pink path (c) is the closest to it, both from a shape and hence from a performance viewpoint.

6. Conclusions

To optimize performance, predictability, and energy efficient operation of virtualized heterogeneous multicore systems for dynamic application workloads, new runtime techniques are required based on monitoring components deployed at critical system locations. This report describes a set of hardware monitoring components developed for a heterogeneous SoC architecture which enable the synergistic hardware and software extensions for dynamic system adaptation.

Innovative hardware-level enhancements and corresponding design methodologies at different system layers can advance virtualization technology by alleviating software overheads, leading to sophisticated hypervisor enhancements supporting not only basic global shared address translation, but also runtime system monitoring and control, dynamic power management, shareability, system-wide cache coherence and a well-defined memory consistency model, thereby ultimately exploiting the high performance capabilities of the underlying physical layer.

In addition to monitoring services, partitioning and reconfiguration support has been also reported. By partitioning, the hypervisor will be able to assign different sets of GPPA resources to running applications, thus providing the required isolation effect of a virtualized system. Also, runtime reconfiguration has been reported, in which the underlying NoC of the GPPA will be reconfigured with minimal impact on running applications.

Finally, this deliverable has reported on the successful development of a soft QoS package, including packet-level, and traffic flow-level QoS guarantees, up to the reservation of circuits. QoS provisions have been specifically conceived for the GPPA of the system. Packet-level provisions consider the differentitation of traffic types in the network, but also the specific requirements of control signaling or the traffic imbalances naturally found in topologies, thus justifying a priority ranking. Within the same priority class, round robin is arbitration is preserved. Also, conflicts between traffic to L2 and intra-partition traffic is limited to NoC links only by means of virtual channels, that at the same type deliver maximum link bandwidth exploitation. Finally, the synergy between the NoC architecture and the centralized software controller enables the reservation of circuits in the network without suffering from the overhead for path setup and teardown. Bandwidth reservation ends up being very similar to traffic rerouting around a faulty link. Finally, the interdependencies between runtime network reconfiguration and delivery of QoS over time has been considered. In this direction, local reconfiguration schemes are proposed, that tweak the token propagation mechanism of OSR-Lite. Also, the most suitable ring topology patterns are analysed to best match the token propagation pattern in the main network of the OSR-Lite runtime reconfiguration mechanism.

References

[1] R.Das, O. Mutlu, T. Moscibroda, C.R. Ras, “AE´RGIA: A Network-on-chip Exploiting Packet Latency Slack”, Micro, 2011, pp. 2-14

[2] L. Tedesco, F. Clermidy, and F. Moraes, “A monitoring and adaptive routing mechanism for QoS traffic on mesh NoC architectures.” In CODES+ISSS, 2009, pp. 109–118.

[3] F. Moraes, N. Calazans, A. Mello, L. M¨oller, and L. Ost, “Hermes: an infrastructure for low area overhead packet-switching networks on chip,” Integr. VLSI J., vol. 38, pp. 69–93, October 2004.

[4] M. Daneshtalab, M. Ebrahimi, P. Liljeberg, J. Plosila, H. Tenhunen, “Memory-Efficient On-Chip Network with Adaptive Interfaces”, IEEE Trans. On Comp.-Aid. Des. of Integ. Circ. and Syst., vol. 31, no. 1, Jan 2012, pp. 146-159.

[5] G. Kornaros, I. Papaefstathiou, A. Nikologiannis, N. Zervos, “A fully-programmable memory management system optimizing queue handling at multi gigabit rates”, Proceedings of the 40th annual Design Automation Conference, 2003, pp 54-59

[6] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, “Memory access scheduling,” in Proc. ISCA, 2000, pp. 128–138.

[7] F. Qin, C. Wang, Z. Li, H. seop Kim, Y. Zhou, and Y. Wu, “Lift: A low-overhead practical information flow tracking system for detecting security attacks”, In MICRO 39, pp 135-148, 2006

[8] N. Nethercote and J. Seward, “Valgrind: a framework for heavyweight dynamic binary instrumentation”, In PLDI, pp 89-100, 2007.

[9] D. Bruening, T. Garnett, and S. Amarasinghe, “An infrastructure for adaptive dynamic optimization”, In CGO, pp 265-275. IEEE Computer Society, 2003.

[10] M. A. Al Faruque, R. Krist, J. Henkel, “ADAM: run-time agent-based distributed application mapping for on-chip communication”, In Proceedings of the 45th annual Design Automation Conference, DAC ’08, 760–765, 2008

[11] O. Lysne, J. Montanana, J. Flich, J. Duato, T. Pinkston, and T. Skeie, ``An efficient and deadlock-free network reconfiguration protocol,'' IEEE Transactions of Computers, vol. 57, no. 6, pp. 762--779, 2008.

[12] W. Dally, L. Dennison, D. Harris, K. Kan, and T. Xanthopoulus, ``The reliable router: A reliable and high-performance communication substrate for parallel computers,'' in Proceedings of the Workshop on Parallel Computer Routing and Communication (PCRCW), May 1994, pp. 241--255.

[13] C. Glass and L. Ni, ``Fault-tolerant wormhole routing in meshes without virtual channels,'' IEEE Transactions Parallel and Distributed Systems}, vol.7, no.~6, 1996.

[14] M. Gómez, J. Duato, J. Flich, P. López, A. Robles, N. Nordbotten, O. Lysne, and T.~Skeie, ``An efficient fault-tolerant routing methodology for meshes and tori,'' Computer Architecture Letters}, vol. 3, no. 1, pp. 3--3, January-December 2004.

[15] C.-T. Ho and L. Stockmeyer, ``A new approach to fault-tolerant wormhole routing for mesh-connected parallel computers,'' IEEE Transactions on Computers, vol. 53, no. 4, pp. 427--439, 2004.

[16] K. M. et al., ``Fibre channel switch fabric-2 (fc-sw-2),'' NCITS 321-200x T11/Project 1305-D/Rev 4. 3 Specification, Tech. Rep., March 2000.

[17] M. Schroeder, A. Birrell, M. Burrows, H. Murray, R. Needham, T. Rodeheffer, E. Satterthwaite, and C. Thacker, ``Autonet: a high-speed, self-configuring local area network using point-to-point links,'' IEEE Journal on Selected Areas in Communicartions, vol. 9, no. 8, pp. 1318--1335, October 1991.

[18] R. Casado, A. Berm\'udez, , J. Duato, F. Quiles, and J. S\'anchez, ``A protocol for deadlock-free dynamic reconfiguration in high-speed local area networks,'' IEEE Transactions on Parallel and Distributed Systems, vol. 12, no. 2, pp. 115--132, February 2001.

[19] O. Lysne and J. Duato, ``Fast dynamic reconfiguration in irregular networks,'' in Proceedings of the 2000 International Conference of Parallel Processing (ICPP).

[20] T. Pinkston, R. Pang, and J. Duato, ``Deadlock-free dynamic reconfiguration schemes for increased network dependability,'' IEEE Transactions on Parallel and Distributed Systems, vol. 14, no. 8, pp. 780--794, 2003.

[21] J. Duato, O. Lysne, R. Pang, and T. Pinkston, ``Part I: A theory for deadlock-free dynamic network reconfiguration,'' IEEE Transactions on Parallel and Distributed Systems, vol. 16, no. 5, pp. 412--427, May 2005.

[22] O. Lysne, T. Pinkston, and J. Duato, ``Part II: A methodology for developing deadlock-free dynamic network reconfiguration processes,'' IEEE Transactions on Parallel and Distributed Systems, vol. 16, no. 5, pp. 428--443, May 2005.

[23] D. Avresky and N. Natchev, ``Dynamic reconfiguration in computer clusters with irregular topologies in the presence of multiple node and link failures,'' IEEE Transactions Computers, vol. 54, no. 5, pp. 603--615, May 2005.

[24] J. Acosta and D. Avresky, ``Intelligent dynamic network reconfiguration,'' in Proceedings of the 21st International Parallel and Distributed Processing Symposium (IPDPS)

[25] D. Fick, A. DeOrio, J.H., V. Bertacco, D. Blaauw, and D. Sylvester, ``Vicis: A reliable network for unreliable silicon,'' in Proceedings of the 46th Design Automation Conference (DAC), July 2009, pp. 812--817.

[26] C. Feng, Z. Lu, A. Jantsch, J. Li, and M. Zhang, ``A reconfigurable fault-tolerant deflection routing algorithm based on reinforcement learning for Network-on-Chip,'' in Proceedings of the International Workshop on Network on Chip Architectures (NocArc), 2010.

[27] Z. Zhang, A. Greiner, and S. Taktak, ``A reconfigurable routing algorithm for a fault-tolerant 2D-mesh Network-on-Chip,'' in Proceedings of the 46th Design Automation Conference (DAC)

[28] V. Puente, J. Gregorio, F. Vallejo, and R. Beivide, ``Immunet: A cheap and robust fault-tolerant packet routing mechanism,'' in Proceedings of the 31th Annual International Symposium on Computer Architecture (ISCA).

[29] J. Flich, A. Mejia, P. Lopez, and J. Duato, ``Region-based routing: An efficient routing mechanism to tackle unreliable hardware in network on chips,'' in Proceedings of the First International Symposium on Networks-on-Chip, ser. NOCS '07.

[30] K. Aisopos, A. DeOrio, L.-S. Peh, and V. Bertacco, ``Ariadne: Agnostic reconfiguration in a disconnected network environment,'' in Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT), 2011.

[31] A. Ghiribaldi, D. Ludivici, F. Trivi\v{n }o, A. Strano, J. Flich, J. Sánchez, F. Alfaro, M. Favalli, and D. Bertozzi, ``A complete self-testing and self-configuring noc infrastructure for cost-effective MPSoCs,'' ACM Transactions on Embedded Computing Systems, 2011.

[32] S. Stergiou et al., ``Xpipes lite: a synthesis oriented design library for networks on chips,'' in DAC, 2005.

[33] S. Rodrigo, J. Flich, A. Roca, S. Medardoni, D. Bertozzi, J. Camacho, F. Silla, and J. Duato, ``Addressing manufacturing challenges with cost-efficient fault tolerant routing,'' in Proceedings of the 2010 Fourth ACM/IEEE International Symposium on Networks-on-Chip, ser. NOCS '10.

[34] A. Mejia, J. Flich, and J. Duato, ``On the potentials of segment-based routing for nocs,'' in Parallel Processing, 2008. ICPP '08. 37th International Conference on, sept. 2008, pp. 594 --603.

[35] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, ``Simics: A Full System Simulation Platform,'' Computer, vol. 35, no. 2, pp. 50--58, 2002.

[36] M. M. K. Martin, et al. , ``Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset,'' SIGARCH Comput. Archit. News, vol. 33, no. 4, pp. 92--99, 2005.

[37] C. Bienia and K. Li, ``Parsec 2.0: A new benchmark suite for chip-multiprocessors,'' in Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation, June 2009.

[38] F. Gilabert, M. E. Gómez, S. Medardoni, and D. Bertozzi, ``Improved utilization of noc channel bandwidth by switch replication for cost-effective multi-processor systems-on-chip,'' in Fourth ACM/IEEE International Symposium on Networks-on-Chip, 2010, pp. 165--172.