A Transition-Signaling Bundled Data NoC Switch Architecture for Cost-Effective GALS Multicore Systems

Alberto Ghiribaldi
ENDIF, University of Ferrara
Ferrara, Italy

Davide Bertozzi
ENDIF, University of Ferrara
Ferrara, Italy

Steven M. Nowick
Dept. of Computer Science, Columbia University
New York, NY, USA

Abstract— **Asynchronous networks-on-chip (NoCs) are an appealing solution to tackle the synchronization challenge in modern multicore systems through the implementation of a GALS paradigm. However, they have found only limited applicability so far due to two main reasons: the lack of proper design tool flows as well as their significant area footprint over their synchronous counterparts. This paper proposes a largely unexplored design point for asynchronous NoCs, relying on transition-signaling bundled data, which contributes to break the above barriers. Compared to an existing lightweight synchronous switch architecture, *xpipesLite*, the post-layout asynchronous switch achieved a 71% reduction in area, up to 85% reduction in overall power consumption, and a 44% average reduction in energy-per-flit, while mastering the more stringent timing assumptions of this solution with a semi-automated synthesis flow.**

I. Introduction

There is today little doubt on the fact that a high-performance and cost-effective network-on-chip (NoC) can only be designed in nanoscale technologies under relaxed synchronization assumptions. On the other hand, such relaxation is even desirable for the upper design layers, since modern systems are typically structured into multiple, highly power-manageable voltage and frequency domains. The Globally Asynchronous Locally Synchronous (GALS) design paradigm can effectively support this architectural trend [1], [2]. Absorbing the heterogeneity of timing assumptions in such systems is ultimately a burden of the system interconnect, which serves as the global integration framework.

Asynchronous NoCs are the best candidates to take on this role, since they rely on clockless handshaking for inter-domain communication. These designs come with a number of potential advantages: average-case instead of worst-case performance, no switching power of a clock tree, easier convergence of hierarchical design flows, robustness to process/voltage/temperature variations, and efficient delivery of differentiated per-link performance. However, asynchronous NoCs are not yet at the stage of a mature interconnect technology for widespread industrial uptake, and their exploitation in real systems turns out to be slower than expected.

There are two fundamental barriers that prevent asynchronous NoCs from becoming a mainstream technology. First, they suffer from poor CAD tool support, in that design methods and tools for synchronous design cannot be directly applied. Many rely on a full-custom approach for the design of such circuits [3]. Some recent work holds the promise of bridging this gap to some extent [4], [5]. However, asynchronous NoC components are still typically delivered as rigid hard macrocells [6]. Second, the vast majority of previous work makes use of four-phase return-to-zero (RZ) protocols, involving two complete round-trip channel communications per transaction, as well as delay-insensitive (DI) data encodings (namely dual-rail, 1-of-4 or m-of-n) [7]. These design choices have typically resulted in an overly large area and energy-per-bit overhead with respect to their synchronous counterparts [8].

More recently, the above considerations have raised the interest in single-rail bundled data asynchronous protocols [7], which in principle provide designs with a lower timing robustness while significantly reducing area, wire-per-link and energy-per-bit overhead. In practice, with a bundled data approach, circuit timing must be carefully specified and controlled to ensure correct operation. At the same time, transition-signaling (i.e., two-phase communication protocol) is gaining momentum as a preferred match for high-performance asynchronous systems [9], especially for signaling across inter-router links [10]. Inside routers, four-phase protocols are still generally preferred since most existing two-phase asynchronous pipeline components are complex, with large latency, area and power overheads.

The objective of this paper is to materialize a new design point for NoC switch architectures, bringing the benefits of asynchronous communication within reach of cost-constrained multicore systems. To the best of our knowledge, this is the first time a full 5-ported asynchronous switch is designed with a transition-signaling bundled data protocol. Our main target is twofold. On the one hand, we aim at a switch architecture that largely outperforms its synchronous counterpart with respect to area footprint, energy-per-bit and power consumption, while maintaining roughly comparable performance. In many previous quasi delay-insensitive (QDI) implementations, with delay-insensitive channels, while lower overall power is delivered, there are significant overheads in area and energy-per-bit. To validate our claim, we compare with one of the most cost-effective synchronous switch architectures for the multicore domain, called *xpipesLite* [11]. On the other hand, we aim to be fully compatible with a standard cell design methodology and with a mainstream CAD tool flow for synchronous design. More specific contributions of this work are the following:

- *we take on the challenge of using two-phase bundled data* not only on inter-switch links but also *inside the switch microarchitecture,* to obtain high performance while not missing the low-complexity target;
- while a promising trade-off between cost and performance with transition-signaling bundled data has been previously been obtained with simple routing and arbitration primitives, this work is practically extended to a more intricate *full 5-ported switch architecture*;
- we aim at design convergence and high performance – *above 900 Mflit/sec* – in low-power standard-Vth 40nm technology;
- we introduce two novel, highly-concurrent and efficient asynchronous components, a transition-signaling *circular FIFO* and a *4-way arbiter,* each of which can be useful in other domains;
- we present a *semi-automated design flow* specific for transition-signaling bundled data design, which exploits commercial synchronous tools, and allows the creation of partially-reconfigurable macros;
- we *compare quality metrics of a post-layout design* of the new asynchronous switch *with a lightweight synchronous switch architecture (xpipesLite [11]),* in order to prove that through the selected asynchronous design style it is possible to provide an even more competitive design point, thus aiming to bring the benefits of asynchronous interconnect technology within reach of resource-constrained multicore systems;
- in the comparison framework with the synchronous switch, we consider *link parasitic effects,* which are of key importance in nanoscale technologies and whose implications on asynchronous switch performance are typically overlooked.

This work was supported in part by NSF awards CCF-1219013 and CCF-0811504, and by NaNoC and vIrtical European Projects (FP7-ICT-248972 and FP7-ICT-288574).

II. Previous Work

There has been a surge of interest in recent years in GALS and asynchronous design [1], [2]. Several GALS NoC solutions have been proposed to enable structured system design. Several of these approaches have been highly effective, especially for low- and moderate-performance distributed multicore systems [6], [12], thus targeting a different point in the design space than the proposed work. Some have low throughput (e.g., 200-250 MHz) [13], while those with moderate throughput (e.g., near 500 MHz [6], [14], [15], [16]) often have significant overheads in router node latency/area/energy-per-bit. Almost all use *four-phase return-to-zero protocols,* involving two complete roundtrip channel communications per transaction (rather than the single roundtrip communication targeted in our work), and delay-insensitive data encoding, resulting in lower coding efficiency than the single-rail bundled encoding used in this paper [14], [15], [12], [13], [17].

Closer to our work is a promising recent approach targeting a two-phase protocol using a commercial computer-aided design (CAD) flow [18]. However, it has overheads due to a delay-insensitive (LEDR) data encoding and flipflop-based registers, and is not currently even suitable as a NoC. The GALS neural network system of [13] also includes two-phase channels between chips, with four-phase channels on chip, but uses delay-insensitive encoding.

The proposed NoC is based on MOUSETRAP pipelines [7], [19], which use a low-overhead single-latch-based architecture. This paper delivers a previously unexplored design point for asynchronous NoC architectures, relying on two-phase bundled data encoding. We propose a more aggressive approach than [10], who limits the two-phase protocol to inter-switch links. The proposed solution builds on the work of [9] and [20], which demonstrate that transition-signaling single-rail bundled data can be efficiently employed in basic routing and arbitration functions. However, [9] and [20] target only simple tree-based switch architectures, while this work addresses the intricacy of a full 5-ported switch design.

III. Switch Architecture

The proposed switch architecture is highly modular and can support the connection of an arbitrary number of input ports (Input Port Modules, IPMs) with output ports (Output Port Modules, OPMs). While the design space is potentially quite large, this paper analyzes and characterizes a specific design point with the following features: 5 input and 5 output ports, 32-bit flit width, wormhole switching and algorithmic dimension-order routing. The ultimate goal is in fact to assess the quality metrics that transition-signaling bundled data can achieve on a specific design point of practical interest.

The switch architecture is inspired by the xpipesLite architecture [11], which represents an ultra-low complexity design point in the space of fully-synchronous NoCs. Given this, coming up with an asynchronous switch consisting of the same building blocks while further cutting down on area and power is the challenge that this paper takes on. xpipesLite will be retained as reference design point to prove the claim of low implementation overhead and competitive design point.

A. Mousetrap Pipelines

The new asynchronous switch introduced in the following sections, is based on an existing asynchronous pipeline called MOUSETRAP [7], [19], which provides high-throughput operation with low hardware overhead. Each MOUSETRAP stage uses a single register based on level-sensitive latches (rather than edge-triggered flipflops) to store data, and simple stage control (replacing the clock) consisting of only a single combinational gate. These designs use single-rail bundled data encoding, where a synchronous-style data channel is augmented with an extra req wire, and a single transition on the req accompanying the data *bundle* indicates the data is valid. The req wire has a simple one-sided timing constraint that its delay is always slightly greater than the data channel. For further details, see [7], [19].

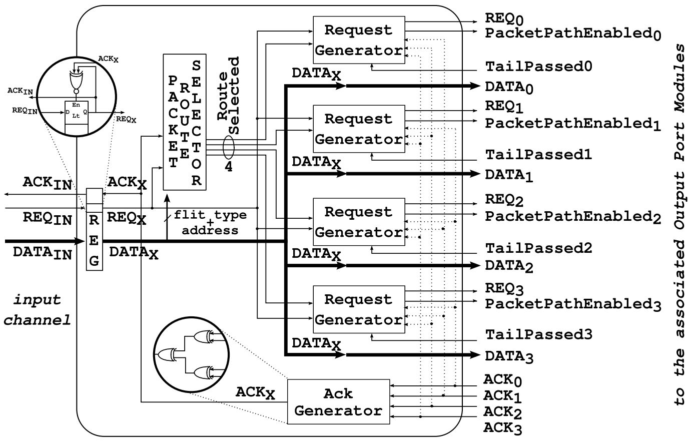

Fig. 1. Input Port Module

B. Input Port Module Architecture

Input Port Modules route the packet to the correct Output Port Module, comparing the internal switch address with the destination address contained in the header flit. The microarchitecture of an IPM is presented in Fig. 1.

The single-latch input register is normally transparent, as in MOUSETRAP pipelines, and the four Request Generator blocks are initially inactive. The basic operation of the module begins with a head flit arriving from the input channel, signalled by REQ_{IN} (soon after $DATA_{IN}$ arrives). The flit passes directly though the the input register, which then makes itself opaque, safely storing the data. It also sends the request (REQ_x) to all four Request Generator blocks, and an acknowledge (ACK_{IN}) on the input channel. The head flit also indicates to the Packet Route Selector to compute the single target output port, and to assert the corresponding one of four *RouteSelected* signals high. This signal sets the corresponding Request Generator to $packet\ processing\ mode$, which asserts its $PacketPathEnabled_i$ output high and sends it to the target Output Port Module. In tandem, the Req_x signal is broadcast to all four Request Generator modules, which result in transitions on *all four* output requests (REQ_0 to REQ_3, one to each of the four Output Port Modules); however, only the one targeted Output Port Module will be activated, and the other requests are ignored (see details below).[1]

The target Output Port Module, after receiving both $PacketPathEnabled_i$ and Req_i, sends acknowledgment ACK_i to the Input Port Module. The Ack Generator then makes a transition on output ACK_x, causing the input register to become transparent again. It is also sent to the Packet Route Selector, which deasserts the *Route Selected* output.

As long as the Request Generator Block is still in $packet\ processing\ mode$, its $PacketPathEnabled_i$ output remains high, and all the flits of the packet are directly transferred to the corresponding Output Port Module. Finally, when a tail flit is received, $TailPassed_i$ is asserted by the Output Port Module, which resets the Request Generator to inactive mode and deasserts the $PacketPathEnabled_i$ signal.

Details of the *Packet Route Selector* are shown in Fig. 2. The XOR2 converts the two-phase signals Req and Ack to a level signal, and a matching delay line enforces hazard-free operation of the combinational routing logic.

The implementation of the *Request Generator* associated with Output Port 0 is shown in Fig. 3. The $Route\ Selected_0$, $TailPassed_0$ and $PacketPathEnabled_0$ signals are all four-phase (i.e. level) and active-high; when the block is inactive, these signals are deasserted low. In contrast, the *Req* and *Ack*

[1]Note that, since a two-phase signaling protocol is used, a REQ_i signal may at times have the *opposite polarity* of the incoming request REQ_x, depending on the number of flits that have been transmitted in previous transfers, and to which of the four Output Port Modules.

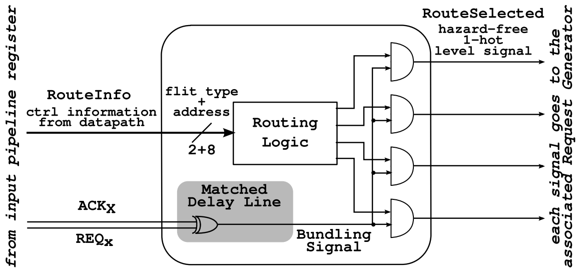
from input pipeline register
RouteInfo
ctrl information from datapath
flit type + address
2+8
Routing Logic
ACKX
REQX
Matched Delay Line
Bundling Signal
RouteSelected
hazard-free 1-hot level signal
each signal goes to the associated Request Generator

Fig. 2. Packet Route Selector

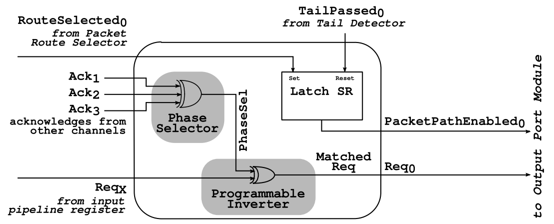
RouteSelected0 from Packet Route Selector
TailPassed0 from Tail Detector
Ack1
Ack2
Ack3
acknowledges from other channels
Phase Selector
PhaseSel
Latch SR
Set
Reset
PacketPathEnabled0
Matched Req
Req0
ReqX from input pipeline register
Programmable Inverter
to Output Port Module

Fig. 3. Request Generator for Output Port Module 0

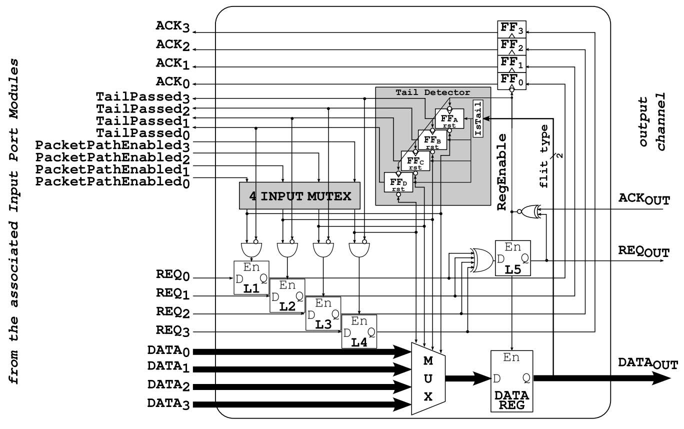

Fig. 4. Output Port Module

signals are two-phase. Whenever a request transition arrives on Req_x, it causes a transition on Req_0.

There are two cases of operation. If this is the Request Generator for the target output port, the unit receives $RouteSelected_0$ asserted high. It then enters *packet processing mode*, and asserts $PacketPathEnabled_0$ high. The XOR2 is used as a programmable inverter, where the correct polarity of the Req_0 output is selected by the XOR3 gate (i.e. phase converter logic). Eventually, when the tail flit arrives, the Output Port Module asserts $TailPassed_0$ high, which resets the Request Generator to an inactive state while deasserting $PacketPathEnabled_0$ low. Finally, the $TailPassed_0$ signal is deasserted low.

Alternatively, if this is not the Request Generator for the target output port, i.e. $RouteSelected_0$ is not asserted, the unit is not activated and $PacketPathEnabled_0$ remains low. As each flit arrives, the Req_x transition causes a Req_0 transition, which is ignored by the corresponding Output Port Module. In fact, in this case, Req_0 makes two transitions for each flit: the XOR3 observes the flit acknowledgment from *every other* Output Port Module (i.e. ACK_1, ACK_2 or ACK_3), thus always returning Req_0 to its original value.

C. Output Port Module Architecture

Output Port Modules arbitrate between multiple incoming requests trying to access the associated output channel. The microarchitecture of an OPM is presented in Fig. 4.

Initially, all $PacketPathEnabled_i$ and $TailPassed_i$ signals are low, and the wires of each transition-signaling REQ_i and ACK_i pair have the same values. Latches L1 to L4 are opaque, blocking new requests until they are arbitrated. Latch L5 and the Data Register are normally transparent,

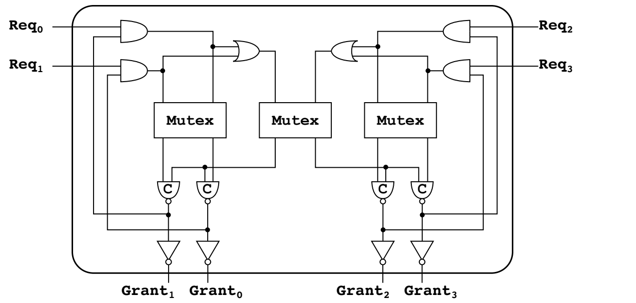

Fig. 5. Microarchitecture of new 4-input arbiter

assuming no congestion, similar to a basic MOUSETRAP pipeline register.

A new transfer begins when a header flit arrives from one of the IPMs, concurrently with the associated $PacketPathEnabled_i$ signal asserted high. The 4-way mutex arbitrates requests from multiple IPM's trying to access to the same output channel, granting access to exactly one of them. Once the mutex is resolved, it performs two concurrent actions: it (i) selects the correct data input of the multiplexer, and (ii) forwards the winning request to the output register by making the corresponding latch (L1 to L4) transparent. The 4-input XOR gate functions as a $merge$ element, joining four mutually-exclusive two-phase signals into a single request. This latch and the multiplexer are programmed once at the start of a packet transmission, and remain unchanged until after the tail flit arrives.

After the output channel request, REQ_{Out}, makes a transition, the data register and latch L5 are made opaque. They become transparent again when the acknowledge, ACK_{OUT}, is received, indicating that the flit has been received downstream. When data and request are safely stored ($ReqEnable$ goes low), the unit sends an acknowledge, ACK_i, to the appropriate IPM, completing the left handshaking communication. As each subsequent body flit of the packet arrives, as long an acknowledgment ACK_{OUT} has been received for the previous flit, its data $DATA_i$ propagates directly through the multiplexer and data register, and its request REQ_i propagates directly through the corresponding latch (L1 to L4), to the output channel.

Packet transmission ends after the tail flit arrives. When the flit is sent on the output channel, the $TailPassed_i$ signal (asserted high) is sent to the source IPM, along with the transition-signaling acknowledge, ACK_i. Once asserted, $TailPassed_i$ will also cause the corresponding request latch (L1 to L4) to become opaque. In turn, the corresponding IPM will deassert $PacketPathEnabled_i$, thereby releasing the mutex, and the Tail Detector then deasserts $TailPassed_i$.

D. 4-Input Mutex Design

The microarchitecture of the new 4-input mutex is presented in Fig. 5. While a previous widely-used 4-input mutex design [16], [21] uses 6 two-input mutex elements and has a serial critical path through 3 mutex elements, the proposed solution uses 3 two-input mutex elements and has a critical path through only 1 mutex element.

In this design, the left mutex element arbitrates between requests 0 and 1, the right mutex element arbitrates between requests 2 and 3, and the center mutex element arbitrates between requests from the right and left pairs. C-elements are used to synchronize the operation of the middle mutex with the side ones, both during the acquire and release phases. Whenever a grant is given, any other request coming from the channel on the same side of the winning one will be killed. The rationale is that, when the winning request will be deasserted, the middle mutex has to be released, so no other requests must be coming from the same side. This behavior provides fair arbitration between incoming requests: in fact, requests from the other side will now have an advantage in acquiring the middle mutex. In other words, the policy implemented is a round robin between left and right side, and round robin between requests within the same side.

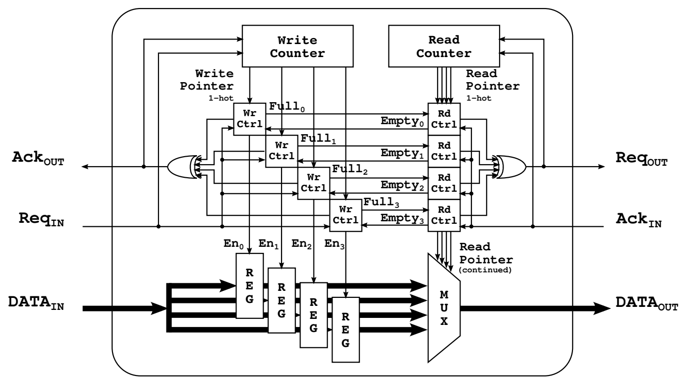
Fig. 6. Circular FIFO: top-level view

E. Transition-Signaling Circular FIFO

FIFOs can be useful to provide additional storage capacity, so to improve system-level performance. Multiple MOUSETRAP registers can be placed one after the other, so to build a serial FIFO, but this introduces severe latency penalty. To overcome this issue, a new circular FIFO is here proposed. Unlike [22], which uses a bus-based interface, the proposed design can provide much lower latency and cycle time.

The microarchitecture of the transition-signaling circular FIFO is shown in Fig. 6. The FIFO uses a two-phase protocol with single-rail bundled data. Write and Read Pointers are 1-hot level signals, selecting the active Write or Read Control Blocks. A new transfer begins when new Data arrives, stabilizes and is then followed by Req_{IN} signal assertion (high in the case under analysis). A new data can be stored in a buffer slot only if: (a) the write pointer selects that slot, and (b) that slot is empty ($Full_i$ is at the same logic level of $Empty_i$). This condition is detected in the Write Control Block by an AND gate merging the two conditions, as can be seen in Fig. 7(a). The 2-input XOR in the Write Control block implements a phase conversion, to provide the correct polarity of the Request signal at the input of the Request Latch, similar to the strategy used in the $PacketRouteSelector$ in the Input Port Module. The new request causes the active Write Control block to assert $Full_0$, close the corresponding register to store the coming Data (by deasserting En_0), and assert the acknowledge to the input. This acknowledge will be merged by the N-input XOR to generate the Ack_{OUT} signal upstream. Finally, when Req_{IN} and Ack_{OUT} are at the same logical value, the Write Counter selects the following buffer position for the next operation. This concludes data storage.

When $Full_0$ and $Empty_0$ are at different logic levels, it means that new data is available in the corresponding buffer position. If this is the case and the read pointer selects that position, the Read Control Block (see Fig. 7(b)) will assert a Request, which will be merged with all the other signals outgoing from the other Read Blocks to generate the output Request Req_{OUT}, together with data from the selected position. After this event (now Req_{OUT} and Ack_{IN} are at different logic values), the $ReadPointer_0$ is immediately deasserted, to safely freeze the current value of the output request. In more detail, request latches inside the Read Control Blocks are now closed, while the active Read Control Block makes its internal acknowledge latch transparent, to route the incoming signal to the correct buffer position. Next, an acknowledge signal comes from the downstream environment, causing $Empty_0$ to make a transition and $ReadPointer_1$ to be asserted high, thereby concluding data transmission.

This FIFO is an important element, not yet exploited in this paper, but necessary when considering complete NoC topologies where larger queues might be required.

F. Timing Constraints

To work properly, the proposed architecture requires some one-sided timing constraints to be enforced.

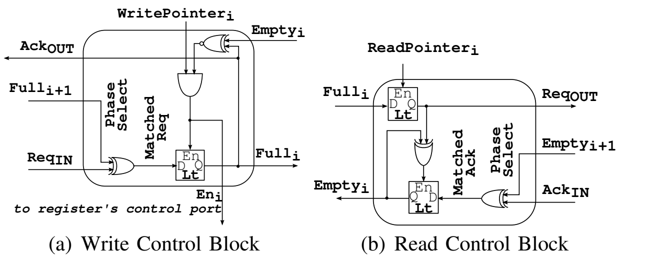
(a) Write Control Block (b) Read Control Block

Fig. 7. Schematic of Write and Read Control Blocks

One is the matching delay inside the $PacketRouteSelector$ block, in order to provide glitch-free operation.

A second constraint is in the Output Port Module: data must be stable long enough before latches are closed inside the output register, i.e. to meet the latches' setup time. This constraint applies to both head flit path setup and body flit propagation, and requires constraining the control path (latches L1-4 and XOR gates) over the multiplexer delay.

Finally, a subtle failure condition can occur during mutex release after a tail flit has passed. The first path starts in the Output Port Module, when acknowledge is generated. This path goes to the Input Port Module, through the acknowledge merge block, through the input register control gate, making the register transparent and allowing a request eventually pushing at its input to enter the module, to propagate through the $RequestControl$ Block, and to reach the Output Port Module. The AND gates above latches L1-4 ensures that once $TailPassed$ signal is asserted, these are closed when the new request comes, reducing the otherwise longer path through *PacketPathEnabled* deassertion and mutex release.

Other relative timing constraints exist inside the 4-input mutex, the Request Control Block and the circular FIFO, but they are typically satisfied in normal operation of the switch.

IV. Semi-Automated Design Flow

All the previously mentioned constraints, plus other delay requirements needed to increase performance, have been enforced across all the steps from logical synthesis to layout design by means of mainstream CAD tools in a semi-automated design methodology. We use a Low-Power Standard-Vt 40nm Industrial Technology library, Normal Process, 1.2V Supply Voltage and 300K Nominal Temperature.

Entry level - The various blocks have been described with low-level RTL models: their functionality has been specified using logical operators, with only few exceptions when implementing specific asynchronous cells. Our technology library does not include C-elements nor mutexes, therefore we use their standard-cell equivalent implementations [23], [24].

Logic Synthesis - The design is synthesized and mapped to the target library using the Synopsys Design Compiler. When using asynchronous design style, not only functionality, but also dynamic behavior is important. In order to ensure glitch-free operation, the tool must be prevented from applying logic optimizations to the design. On the other hand, automatic buffer insertion is a useful optimization option that we would like to exploit. This behavior can be achieved by using the *set_compile_directives* and *set_structure* directives of the tool. While Design Compiler does not understand relative timing constraints, they can be enforced through multiple iterations: in a first run, only max delays are enforced, in order to meet a generic target (max performance, minimum area); then, in a second run, the delay of the paths that have to be matched can be extracted from the netlist (*get_timing_path*, *get_attribute* timing_path *arrival*) and assigned to the required path (*set_min_delay*). The same procedure can be used to check whether the given constraints have been fulfilled, and iterate again if necessary.

Physical Switch Design - For this purpose we used Synopsys IC Compiler. As in the technology mapping procedure, it is possible to enforce, extract and compare the delays between different paths, and automatically verify if constraints have been fulfilled. If not, it is possible to update constraints and iterate place and route again.

Top-Level Implementation - Again, the Synopsys IC Compiler functionality is leveraged. The placed and routed switch netlist is saved as a hard macro and then instantiated in the top level description of the system. This is a typical hierarchical design methodology, which achieves reduced runtimes and faster convergence. Since this hard macro is generated with the standard methodology described above, it still presents some degrees of freedom: data width is parameterizable, as well as its floorplan aspect ratio and the position of input-output ports. For this reason, this is different from a fixed hard macro designed with a full custom approach.

In order to route interconnection wires between any two switches, we choose to give maximum delay constraints over the link, while keeping the same max capacitance and max transition time constraints given inside the switch. After a first routing and buffer insertion, we freeze placement and driving strength of data wires along them (so that their delay will no longer change), and give minimum delay constraints over the request signal to satisfy the bundled data protocol requirement (i.e. a request must arrive always after the corresponding data has stabilized). Then, incremental physical optimization and routing are performed. A different approach has been adopted when implementing links with pipeline stages. A pipeline stage can help reduce cycle time over the link, at the cost of some additional forward latency. As before, switches were implemented as hard macros and placed a few millimeters apart. Architectural repeaters (i.e. MOUSETRAP FIFO stages) are described as soft macros, inferring their placement boundaries in order to have a regular floorplan and guide tool decisions. The tool is then allowed to size pipeline cells and insert buffers to reduce propagation delay.

Even though the previously described methodology could produce a valid outcome, the performance results were not satisfactory. This is because simply setting of max delays on wires does not take into account the internal timing arcs of the latches. This issue prevents the tool to correctly estimate the time spent on link traversal. In order to optimize our procedure, we exploited a similar method to [6], adapting it for the single-rail bundled data case. Through IC Compiler commands, we disabled the path through the feedback loop of register control logic, from input to output pin of latches inside repeaters, and defined the reset signal as a dummy clock. This approach makes latches similar to flip flops, and the clock period gives a constraint on the maximum forward delay. This leaves the acknowledge signal propagating back from each pipeline stage to the previous one unbounded (we disabled the timing path through register control logic), so a maximum delay constraint has been enforced on it in order to speed the path through it. Again, multiple iterations are required to enforce bundling constraint on a request signal. We found that this approach is more suitable when dealing with pipelined links, since it reduces tool run times and the number of iterations required to satisfy matching constraints with respect to the previous methodology.

V. Experimental Results

In order to evaluate our implementation choice, we compared the proposed 5x5 asynchronous switch with the baseline fully synchronous xpipesLite switch.

1) Experimental Setup: While constraints required for correct functionality can be checked and fixed during synthesis and place and route procedures, performance evaluation must be assessed through simulation. In order to assess performance of the asynchronous switch, the following experimental setup has been exploited. The switch under test receives packets from injector modules and forwards them to absorber modules; injector and absorber are simply other instances of the same switch, with fast loops at their boundaries (i.e., maximum injection rate). This environment permits to assess the realistic handshaking mechanism between neighboring switches, thereby avoiding overly-optimistic results.

	Asynchronous	Synchronous
Area	4691 μm^2	16035 μm^2
Latency (Head Flit)	1195 ps	1960 ps
Cycle Time (Avg.)	903 ps	980 ps
Area Efficiency	236 $Mfps/mm^2$	63.63 $Mfps/mm^2$

TABLE I
Asynchronous vs. Synchronous Switch

Latency metric is evaluated as the time interval from a request being asserted at the input port to a request asserted at the output port, assuming the switch is initially empty and there is no congestion. Latency varies depending on the flit position inside the packet. Head flits experience the highest latency since they have to set up the path from input to output port. Latency will be evaluated focusing on head flits, since they are responsible of packet propagation through the network. **Cycle time** is the interval between two successive acknowledgments received at the switch output port.

As far as the synchronous switch is concerned, the implementation allocates one clock cycle to traverse the switch and one clock cycle to traverse the link connecting two switches. For this reason, cycle time is one clock period, while latency amounts to two clock periods.

Area efficiency is a metric that evaluates how much area is necessary to provide a defined performance. It is calculated as:

$$AreaEfficiency = \frac{DeliveredThroughput}{AreaOccupancy}\left[\frac{Mfps}{mm^2}\right]$$

2) Comparative Analysis: Table I presents the measured metrics. The asynchronous switch requires 71% less area, and delivers 39% lower latency and comparable throughput. For this reason, the Area Efficiency metric is 3.7x higher for the asynchronous implementation. It should be observed that part of the area savings come from the different minimum buffering requirements of the two switches. Both of them are latched at inputs and outputs. However, the synchronous switch needs two slot buffers to properly support the stall/go flow control protocol (i.e., not to lose data upon stall assertion). Vice versa, flow control is inherently implemented in the asynchronous clockless handshaking, therefore only a single slot latching stage is sufficient for the asynchronous switch.

3) NoC Link Effect: A typically overlooked issue is the effect of the inter-switch links on performance, which are assumed ideal in Table I. In order to account for this aspect, we implement a complete layout of two switches, placed a few millimeters apart, and route interconnections between them.

Fig. 8 shows how head latency and cycle time degrade for synchronous and asynchronous switches when varying the inter-switch distance. Having an entire clock cycle reserved for link traversal, the synchronous switch maintains a stable performance up to 4 mm link length. From there on, the critical path moves from inside the switch into the link, thus reducing the maximum clock frequency. On the other hand, performance of asynchronous switch gracefully degrades when increasing link length. Latency is always lower when compared with the synchronous counterpart, while cycle time has a steeper degradation, due to the signal roundtrip over the interconnect.

It is well-known that link pipelining is the way to reduce cycle time at the expense of additional latency. This holds for both synchronous and asynchronous design styles, but the implications are completely different. Adding a pipeline stage in synchronous logic always implies one additional clock cycle latency and full retiming and flow control stages, while in asynchronous logic the latency is affected by only a few gates delay since pipeline stages consist of simple latching stages.

Exploiting the procedure described in section IV, effects of link pipelining are analyzed. The number of pipeline stages is selected in order to obtain the same cycle time as the one measured for ideal link. As shown in Fig. 9, the additional pipeline stages negatively affect asynchronous latency, although for link lengths below 2.5 mm performance is always better or equal to that of the synchronous switch. For longer link lengths, the asynchronous solution suffers the most from link parasitics due to the roundtrip delay of its handshaking protocol.

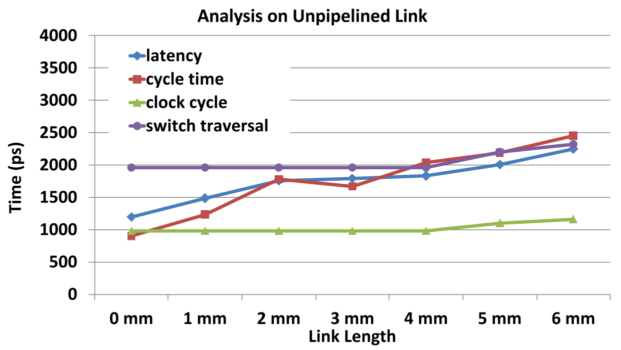

Fig. 8. Performance results with unpipelined links

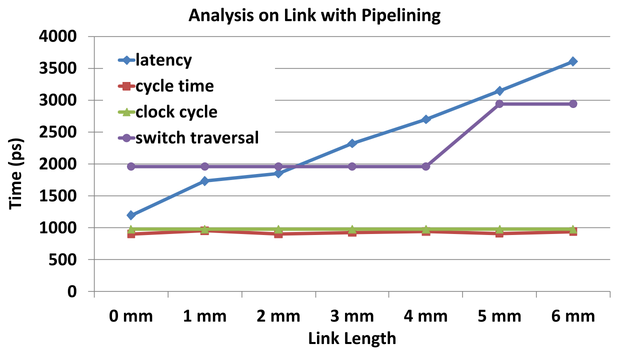

Fig. 9. Performance results with link pipelining

4) Power Analysis: Power consumption of the switches is analyzed under different conditions, using Prime Time on post-layout implementations. For these experiments, the synchronous switch has been implemented in two versions, one without and one with clock gating.

Four different case analysis are presented in Fig. 10. *Leakage power* is smaller in asynchronous design, given the reduced area occupancy. *Idle power* is analyzed considering clock tree power consumption: even if clock gating technique can provide evident benefits with almost no performance penalty with respect to baseline synchronous switch, the asynchronous switch, being completely clockless, requires 90% and 97% less power than clock gated switch and baseline switch, respectively. The remaining two cases use two different traffic benchmarks, with packets composed by 3 and 8 flits. For *hotspot* (a single output channel is accessed by packets coming from every other input channel), the asynchronous switch has an average of 85% less power requirement than synchronous and and 73% less than synchronous with clock gating. For the *parallel* benchmark (all input ports competing for 5 different output ports, no contention), power consumption is reduced on average by 60% with respect to synchronous and 45% with respect to synchronous with clock gating. Overall the small difference between 3 and 8 flits packets is due to the extra power required during arbitration process. Power savings not only come from the clockless architecture of the asynchronous switch but also from its lower complexity and footprint. In fact, observing the energy-per-flit in Fig.11, there is a 44% average reduction with respect to the two synchronous switches.

VI. Conclusions

This paper delivers a largely unexplored design point in asynchronous NoC switch architectures. It relies on transition-signaling bundled data to produce a low overhead design, which at the same time meets the performance of synchronous counterparts. Post-layout results indicate that area is reduced by 71%, idle power by 90%, and energy-per-flit by 45%. Throughput is roughly comparable with the synchronous switch, while latency is actually better up to link lengths of 2.5mm in 40nm technology. Overall area efficiency is superior (3.7x). Finally, the switch is delivered as a partially-reconfigurable hard macro for hierarchical design flows. Timing constraints are tightly controlled through a semi-automatic design flow relying on mainstream synchronous CAD tools.

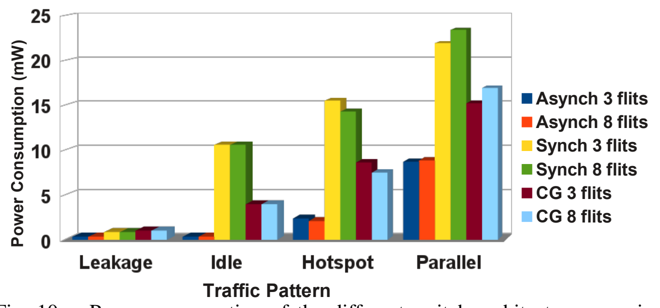

Fig. 10. Power consumption of the different switch architectures, varying the traffic injected

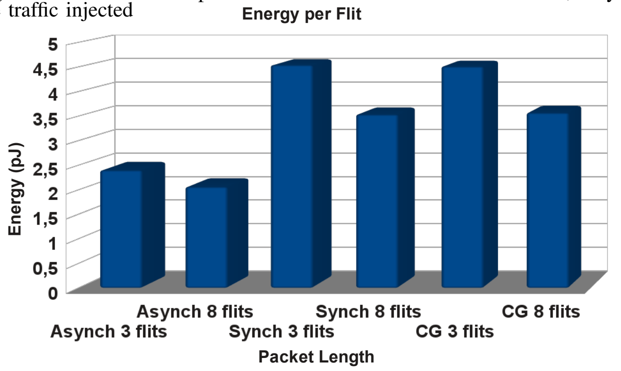

Fig. 11. Average energy required to propagate a Flit form input to output

References

[1] P. Teehan et al., “A survey and taxonomy of GALS design styles”, IEEE Des. Test Comput., vol. 24:5, pp. 418-428, 2007.
[2] D. Chapiro, “Globally-asynchronous locally-synchronous systems”, Ph.D. dissertation, Dept. CS, Stanford University, 1984.
[3] S. Hollis and S. W. Moore, “RasP: An area-efficient, on-chip network”, in IEEE Proc. Int. Conf. Comp. Design, Oct. 2007, pp. 63-69.
[4] K. S. Stevens et al., “Characterization of asynchronous templates for integration into clocked CAD flows”, ASYNC 2009.
[5] Y. Thonnart et al., “A pseudo-synchronous implementation flow for WCHB QDI asynchronous circuits”, ASYNC 2012.
[6] Y.Thonnart, P.Vivet, F.Clermidy, “A fully-asynchronous low-power framework for GALS NOC integration”, DATE 2010.
[7] S.M. Nowick and M. Singh, “High-performance asynchronous pipelines: an overview”, IEEE Des. Test Comput., vol. 28:5, pp. 8-22, 2011.
[8] D. Lattard et al., “A reconfigurable baseband platform based on an asynchronous network-on-chip”, ISSCC, Jan. 2008.
[9] M.N.Horak, S.Nowick, M.Carlberg, U.Vishkin, “A low-overhead asynchronous interconnection network for GALS chip multiprocessors”, IEEE TCAD, vol. 30:4, pp. 494-507, 2011.
[10] D.Gebhardt et al., “Comparing energy and latency of asynchronous and synchronous NoCs for embedded SoCs”, NoCs 2010, pp. 115-122.
[11] S.Stergiou et al., “xpipesLite: a synthesis-oriented design flow for networks on chip”, DATE 2005, pp. 1188-1193.
[12] J. Bainbridge and S. Furber, “CHAIN: A delay-insensitive chip area interconnect”, IEEE Micro, vol. 22:5, pp. 16-23, 2002.
[13] L. Plana et al., “A GALS infrastructure for a massively parallel multiprocessor”, IEEE Des. Test Comput., vol. 24:5, pp. 454-463, 2007.
[14] T. Bjerregaard and J. Sparsoe, “A router architecture for connection-oriented service guarantees in the MANGO clockless network-on-chip”, in Proc. DATE, Mar. 2005, pp. 1226-1231.
[15] A. Sheibanyrad, A. Greiner, and I. Miro-Panades, “Multisynchronous and fully asynchronous NoCs for GALS”, IEEE Design Test Comput., vol. 25:6, pp. 572-580, Nov. 2008.
[16] R. Dobkin, V. Vishnyakov, E. Friedman, and R. Ginosar, “An asynchronous router for multiple service levels networks on chip”, ASYNC 2005, pp. 44-53.
[17] A. Lines, “Asynchronous interconnect for synchronous SoC design”, IEEE Micro Mag., vol. 24:1, pp. 32-41, 2004.
[18] B. Quinton, M. Greenstreet, and S. Wilton, “Practical asynchronous interconnect network design”, IEEE Trans. on VLSI, vol. 16:5, pp. 579-588, 2008.
[19] M. Singh and S. M. Nowick, “MOUSETRAP: high-speed transition signaling asynchronous pipelines”, IEEE Tran. on VLSI, vol. 15:6, pp. 684-698, 2007.
[20] G. Gill, S.S. Attarde, G. Lacourba, S.M. Nowick, “A low-latency adaptive asynchronous interconnection network using bi-modal router nodes”, NOCS 2011, pp. 193-200.
[21] R. Dobkin, R. Ginosar, A. Kolodny “QNoC asynchronous router”, Integration, the VLSI Journal, vol. 42:2, pp. 103-115, 2009.
[22] T. Chelcea and S.M. Nowick, “Low-latency asynchronous FIFO’s using token rings”, IEEE Async Symposium (2000).
[23] van Berkel et al. “Stretching quasi delay insensitivity by means of extended isochronic forks”, WCADM, 1995, pp. 99-106.
[24] L.A.Plana et al., “SpiNNaker: design and implementation of a GALS multicore system-on-chip”, ACM JETC, vol. 7:4, art. 17, 2011.