John von Neumann Institute for Computing

Distributed Congestion Control for Packet Switched Networks on Chip

T. Marescaux, A. Rångevall, V. Nollet, A. Bartic,
H. Corporaal

published in

Parallel Computing:
Current & Future Issues of High-End Computing,
Proceedings of the International Conference ParCo 2005,
G.R. Joubert, W.E. Nagel, F.J. Peters, O. Plata, P. Tirado, E. Zapata
(Editors),
John von Neumann Institute for Computing, Jülich,
NIC Series, Vol. **33**, ISBN 3-00-017352-8, pp. 761-768, 2006.

http://www.fz-juelich.de/nic-series/volume33

Distributed Congestion Control for Packet Switched Networks on Chip

Théodore Marescaux[1], Anders Rångevall[1,2], Vincent Nollet[1], Andrei Bartic[1], Henk Corporaal[3]
[1]IMEC V.Z.W., Kapeldreef 75, 3001 Leuven, Belgium
[2]Department of Information Technology, Lund University, Lund, Sweden
[3]Technical University Eindhoven (TU/e), Eindhoven, The Netherlands
Theodore.Marescaux@imec.be

Packet-switched NoCs are efficient communication architectures for future MP-SoC platforms. However the run-time management of their communication, especially congestion avoidance is a challenging task. This paper discusses a distributed HW/SW congestion control technique. Hardware modules detect early signs of congestion and traffic is re-shaped accordingly to reduce congestion. The shape of the traffic is computed with binomial algorithms running on simple distributed microcontrollers.

The hardware and software extensions we propose to our system provide a congestion control solution that proves to be low-latency (100 clock cycles) and has a low area contribution to the router hardware (about 5% in UMC 0.13 technology).

1. Introduction

In order to meet the ever-increasing design complexity, many future sub-100nm platforms will be on chip multi-processor systems that require the flexibility and scalability of switched communication architectures such as Networks-on-Chip (NoCs). One interesting property of NoCs is their ability to provide a wide range of Quality-of-Service (QoS) levels, ranging from best-effort to soft and hard real-time guarantees.

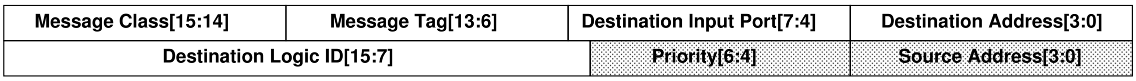
We present the implementation of a packet-switched NoC and its network interfaces (NI) that provide soft real-time guarantees by performing traffic shaping. Traffic is shaped at the sender NI by controlling three parameters of the packets sent: their priority-level, their length and the time they are injected in the NoC. Adapting traffic shapes to the current quality requirements allows to control the congestion in the NoC and provides soft real-time guarantees. This paper focuses on the extensions of our NoC to allow an automatic and distributed mechanism to perform traffic shaping aiming at controlling congestion in the NoC. Our system -emulated on a Virtex-II FPGA coupled to a StrongArm processor- is composed of two NoCs: a 3x3 mesh NoC with virtual cut-through switching for application data traffic and a specialized NoC for control and distributed operating system (OS) services, such as controlling the traffic shaping parameters [1]. These OS services are implemented on small micro-controllers (μC) included in the NIs of the control NoC [2].

Enhancing our system with the ability to dynamically control the traffic shaping in a distributed manner has required a small number of changes, such as modification of the packet header, extension of the interface to the data router for observability of the buffer occupancy and control algorithms implemented in software.

Related work comes in several categories: some authors propose to enforce hard real-time guarantees thus completely avoiding congestion but at the expense of more complex and restrictive scheduling techniques [3]. Other techniques use adaptive routing to distribute traffic and avoid congestion but need to either re-order packets at the expense of large buffers in the network interfaces or need to re-send dropped packets and hence incur a large latency penalty [4,5] . Congestion avoidance by central control of traffic shaping is discussed in [2] and centralized end-to-end flow control in [6].

2. NoC Extensions

Providing distributed congestion control on the data NoC requires hardware and software modifications to our system. The packet header has been extended with the address of the source of the communication, to be able to identify the source of the congestion, and with a priority field (Figure 1). The router has not been modified since it processes the contents of the first header flit and the header extensions only affect the second header flit. The FIFOs inside the data NoC routers have been modified to provide visibility on the number of packets queued and on the *source* and *priority* fields from the header of the last packet buffered. The resulting signals are fed into a *buffer monitor* module that has the role of sensing congestion. A FIFO is considered congested when its occupancy goes above a settable threshold level. The *buffer monitor* keeps a history of *source* and *priority* of the latest packets received. Upon congestion, the history is searched and the *source* of the lowest priority packet is notified to a *congestion monitor* module. The *congestion monitor* multiplexes the congestion notifications of all *buffer monitors* in the router and copies them to the data memory of the μC in the local control NI. The μC local to the router that has sensed congestion can then notify the μC of the *source* router to throttle the traffic coming out of its data NI. As a consequence, congestion is reduced and the path is left free for other communication streams sharing part of the congested path.



<table>
<tr><td>Message Class[15:14]</td><td colspan="2">Message Tag[13:6]</td><td>Destination Input Port[7:4]</td><td>Destination Address[3:0]</td></tr>
<tr><td colspan="3">Destination Logic ID[15:7]</td><td>Priority[6:4]</td><td>Source Address[3:0]</td></tr>
</table>

Figure 1. Packet header extended with source and priority fields.

3. Buffer Monitor

The relatively small size of the buffers in the routers of the data NoC leads to a network that can congest rapidly. In worst-case scenarios, congestion can build-up in as little as $3 * buffer\_depth$ clock cycles, hence the congestion detection mechanism needs to rapidly counteract. The congestion detectors are called *buffer monitors* and there is one *buffer monitor* connected to each output buffer in the routers of the data NoC. On a 2D mesh NoCs a router has between three to five buffers and hence as many *buffer monitors*.

The *buffer monitor* uses the four signals that are exported from each buffer in the Data Router to get information about the packets (Figure 2). Whenever a packet enters the buffer, the router asserts the signal `new_packet`. At the same time the router exports the source and priority of that packet, the `source` and `priority` signals. The fourth signal from the router is the `packets_outqueue` that tells how many packets there are in the buffer.

The primary role of the *buffer monitor* is to sense congestion by measuring how many packets are stored inside the buffer in the router. If there are more packets than the `threshold` level set by the operating system, the buffer is considered to be congested.

Upon congestion, the role of the *buffer monitor* is to select where to send a congestion notification. To make an informed decision, the *buffer monitor* stores information regarding recent packets

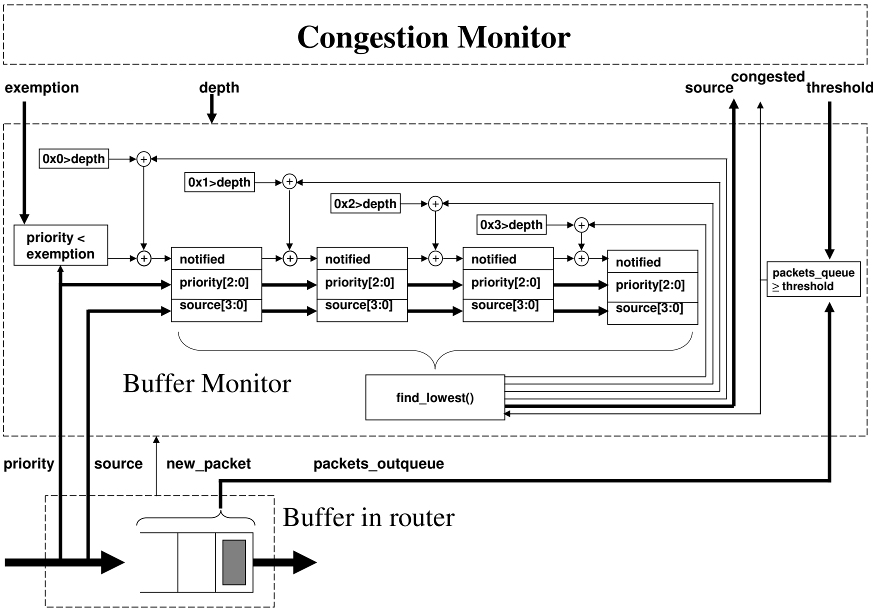


Figure 2. The *buffer monitor*, keeps a history of packets sorted by priority. Upon congestion it reports the lowest priority packet to the *congestion monitor*.

in a separate history FIFO (hFIFO) that contains the source address and the priority of a packet. Whenever a packet header enters the data router buffer, its source address and priority are shifted into the hFIFO.

When a *buffer monitor* senses congestion it uses the `find_lowest` function to search among all the priority fields inside the hFIFO to find the source address of the packet that has the lowest priority. If two packets have the same priority the most recent packet is selected.

A congestion notification containing source address and priority is then reported to the *congestion monitor* (Figure 2). The entry in the history FIFO is then tagged as `notified` to avoid subsequent congestion notifications. The `notified` tag is local to the *buffer monitor* hFIFO, therefore the same packet may trigger more congestion notifications on the next hops along the path.

The maximum size of the hFIFO is set at design-time by the `HISTORY_DEPTH` constant. Nevertheless the effective size of the hFIFO can be controlled at run-time, for the needs of the control algorithm, by setting the `depth` register in the *buffer monitor*. All entries in the hFIFO are searched in parallel with a search tree to allow a very short reaction time, at the expense of increased area. However that area increase is only linear with `HISTORY_DEPTH` (Section 7). The hFIFO implemented in our emulation platform stores information about four packets.

Two `depth` settings are of particular interest: 0x0 and $buffer\_depth$. With a depth of 0x0, information is only stored for one packet. This packet is the most recent one that has entered the buffer. This setting is interesting because it equates to very simple hardware, just store and search one packet. The other setting where `depth` equals $buffer\_depth$ means that the hFIFO stores information about as many packets as the actual buffer holds.

The threshold and depth variables, although they seem similar, are quite different. The threshold variable defines when the buffer is considered to be congested. This is done by comparing the threshold value to the number of packets in the buffer. The depth variable on the other hand only affects for how many packets the source address is remembered.

Another setting of interest is when the `threshold` and `depth` are both set to the same number of packets. For example if the threshold and the depth are both set to two packets, then the source address will be taken from one of the two currently buffered packets. It will not be taken from a packet that has already left the buffer, which can be the case if the threshold is set to two packets and the depth is set to three packets.

To further enhance the flexibility of the congestion control, there is a possibility to completely exempt packets with certain priorities from triggering a state of congestion. This behavior is controlled through the `exemption` register. By setting the `exemption` to 0x3, packets with a higher priority than three – and cause congestion – will never trigger any congestion notifications to be sent. When congestion is caused by another packet, the tile that sends the exempt packet will never receive a congestion notification. In this example the packets with priorities 0x2, 0x1 and 0x0 cannot trigger any congestion notification from being sent. It is important to realize that the exemption mechanism does not prevent congestion from occurring; it merely hides it from the congestion control.

4. Congestion Monitor

Every router on the data NoC contains one *congestion monitor* that fulfills two major roles. On the one hand it acts as a multiplexer/decoder to give the local μC visibility and control over all the *buffer monitors* contained in the data router. On the other hand it is able to autonomously send/receive notification of congestions to/from remote *congestion monitors*. Congestion notifications are sent over a separate control network reserved for low-latency OS traffic [1,2].

Each *congestion monitor* connects to a μC in the local Control Network Interface (Figure 3) and its registers are memory-mapped in the address space of the microcontroller. It is the microcontroller that assigns values to the different variables such as the `depth` and the `threshold` that affect the *buffer monitors*. The *congestion monitor* also contains statistics registers that count the number of congestion notifications received and sent.

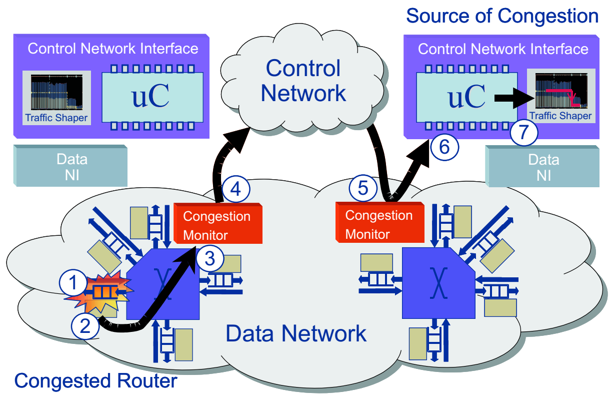
The μC controls the behavior of the *congestion monitor* and of its *buffer monitors*. For instance it can selectively enable *buffer monitors* through the memory-mapped register `cmBufMsk`. Incidentally the *congestion monitor* also contains a hardware timer that is read and set by the μC. It is used to provide the μC with a time-base related to the NoC traffic.

Congestion signaling of the *buffer monitors* is detected by the *congestion monitor* and is relayed as a congestion notification. The destination for the congestion notification is provided by the *buffer monitor* signal `source`. If several *buffer monitors* of the same router indicate congestion at the same clock cycle, only the highest priority congestion notification is sent, the other ones are dropped.

Congestion notifications can either be directly sent from congestion monitor to congestion monitor through the control network, or the autonomous congestion notification can be overridden by the μC by setting the `autonomous_mode` register to zero in the *congestion monitor*. The latter allows the μC to implement in software more complex notification schemes but puts it in charge of sending congestion notifications. By default our system functions in autonomous mode.

5. Congestion Control in Action - an Example

This section shows a simple example of how congestion is detected and how congestion notifications are propagated to the source of congestion. The congestion control mechanism is triggered when a buffer in a router on the data NoC fills up above the threshold level and is thus considered as being congested. Figure 3 shows how the congestion notification is propagated through the system. Steps one to five take place in hardware and take three clock cycles to execute (not including possible, but short, conflicts on the control network). Steps six and seven execute in software on the local μC, typically consuming about 100 clock cycles.



1. The *buffer monitor* detects congestion in the local buffer.
2. The *buffer monitor* searches packet history for congestion source.
3. The source address of the congestion is given to the *congestion monitor*.
4. The *congestion monitor* sends a congestion notification to the congestion source through the *control network*.
5. The congestion notification is processed by the receiving *congestion monitor*.
6. The *uC* polls the *congestion monitor* and gets the congestion notification.
7. The *uC* computes a new window size and configures the *traffic shaper* to reduce throughput accordingly.

Figure 3. Congestion control in action. A buffer in the left router is congested. The arrows show how the congestion notification is propagated to the source of congestion.

6. Traffic Shaping - Control Algorithm

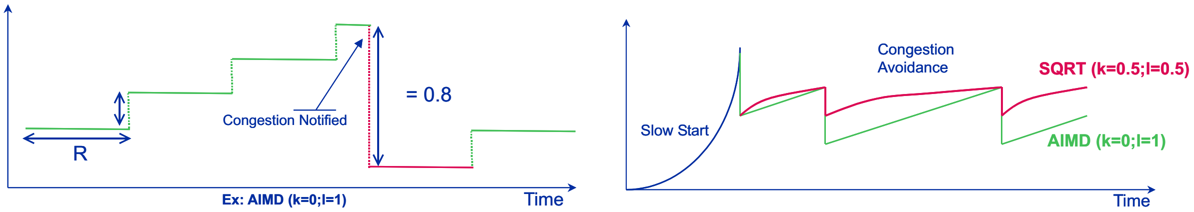
The traffic shaping in our system is based on a sliding window mechanism. Packets are only injected in the network during the time a window ω is opened. The size of the window is based on binomial congestion avoidance [7]:

$$Window\ Increase: \quad \omega(t+R) = \omega(t) + \frac{\alpha}{\omega(t)^k} \quad ;(\alpha > 1) \tag{1}$$

$$Window\ Decrease: \quad \omega(t+\delta t) = \omega(t) - \beta\omega(t)^l \quad ;(0 < \beta < 1) \tag{2}$$

While no congestion is notified, the window size is gradually increased at a rate R (Figure 4(a)), by default 2048 NoC cycles on our system. In the general case, the increase amount is proportional to ω^{-k} (Equation (1)). Shortly after congestion has been notified, the window size is decreased (Equation (2)), usually by a larger amount than it is increased (Figure 4(a,b)). The parameters k and l in Equations (1,2) define the aggressiveness at which the windows are opened and closed and therefore their impact on response to congestion. To ensure a good trade-off between probing aggressiveness and congestion responsiveness, we use the $k + l$ rule defined in [7]: $k + l = 1$ and $l \leq 1$. Figure 4(b) shows the effect of two different sets of (k, l) values that follow the $k + l$ rule. Additive Increase Multiplicative Decrease (AIMD) uses $(k, l) = (0, 1)$ and yields a windowing mechanism that is both efficient and simple to implement. The square root algorithm (SQRT) uses $(k, l) = (0.5, 0.5)$ and thus changes the window size proportionally to $\sqrt{\omega}$ which yields a smoother traffic shaping but is more computationally intensive (Figure 4(b)).

The μCs in the network interfaces are tiny micro-controllers, without floating-point units and have a very limited amount of memory, making it difficult to store constant tables. Therefore, to



(a) Window increase/decrease illustrated on AIMD.

(b) SQRT gives a smoother window variation.

Figure 4. Traffic shaping is based on a sliding-window mechanism.

rapidly compute window sizes, the congestion control algorithm initially implemented on our system is AIMD. In the current implementation the sending rate is divided by half upon notification of congestion and the increase rate is 2048 NoC cycles, or the equivalent of sending 8 packets of maximum length (512 bytes).

7. Results

The distributed congestion-control system is fully implemented and integrated to our emulation platform. An instance with a history depth of 3 packets and 8 levels of priority requires about 3% of the Virtex-II 6000 for all 9 congestion monitors and 33 buffer monitors. The average total response time to congestion is 100 NoC cycles with a variance of 20.

7.1. Synthesis to UMC 0.13 STD Cell Technology

Additionally to the FPGA implementation, we have also synthesized the congestion detection system to standard cell technology using the UMC 0.13 libraries. We believe these numbers to give a good indication of the impact our distributed congestion control system would have in a real chip.

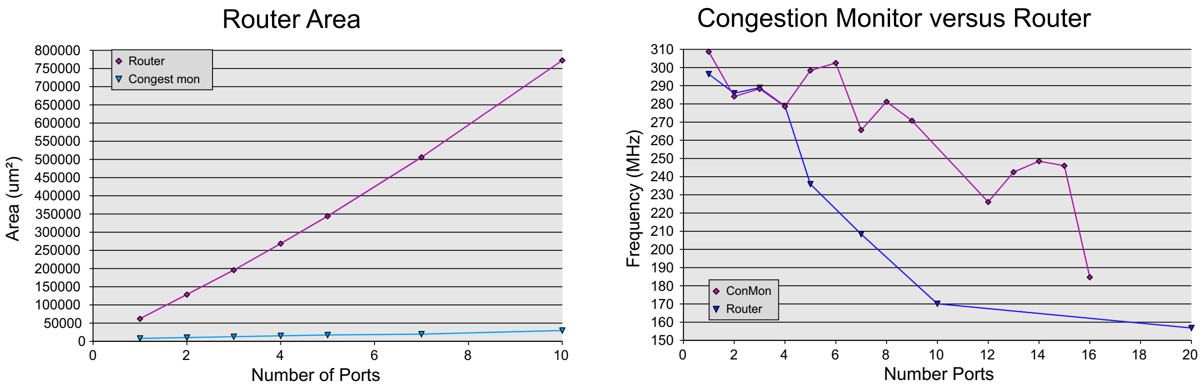
We have compared the area and contribution to the critical path of the whole congestion controller -composed of a congestion monitor and 4 buffer monitors- to the router of the data NoC. It results that the area impact of our congestion control system is negligible with respect to the router area (Figure 5(a)). For instance for a router with 4 ports, the congestion controller occupies about $15000\mu m^2$, which only represent 5% of the router. As Figure 5(b) shows, the critical path of the congestion controller has a negligible effect on that of the router.

Figure 6(a) shows a linear increase to the area contribution of the congestion controller when varying the number of ports, hence the number of *buffer monitors*. Figure 6(b) details the area contribution of a congestion controller with only one *buffer monitors* when varying its `HISTORY_DEPTH`. The dependency is again linear, which shows that `HISTORY_DEPTH` can be increased to slightly higher values to allow more complex control algorithms without having a dramatic impact on area.

7.2. System Latency

A critical requirement of our system is a low latency when responding to a congestion notification. We measure that response latency from the moment a buffer in a data NoC router is congested to the moment the window size is adapted in the traffic shaper to reduce the data throughput at the source of congestion (steps 1 to 7 in Figure 3).

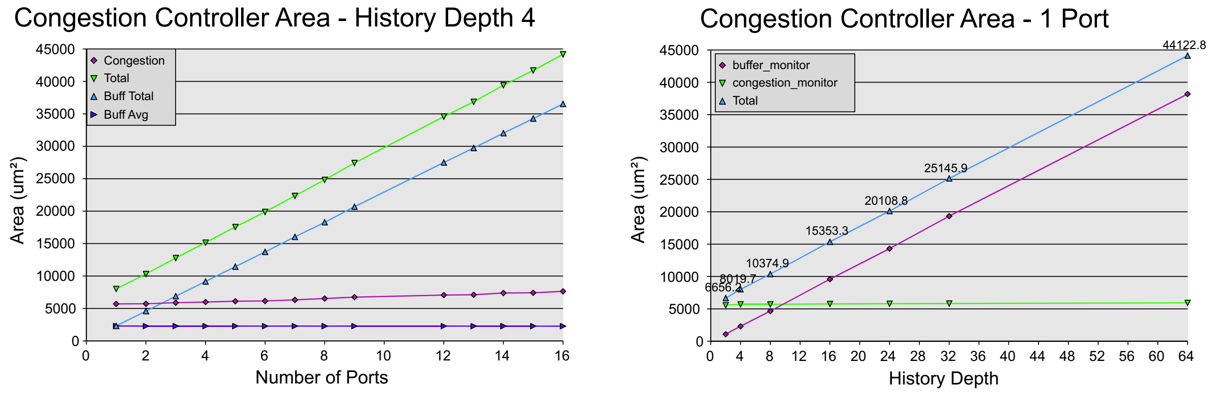
The polling by the microcontrollers determines to a large extent the latency for the congestion control. The whole control loop executes in about 100 clock cycles (Figure 7(a,b)), unless it needs to respond to commands sent from the operating system.



(a) Negligible area of congestion controller.

(b) No impact on router critical path.

Figure 5. The congestion controller proves to be inexpensive (UMC 0.13 STD cell technology).



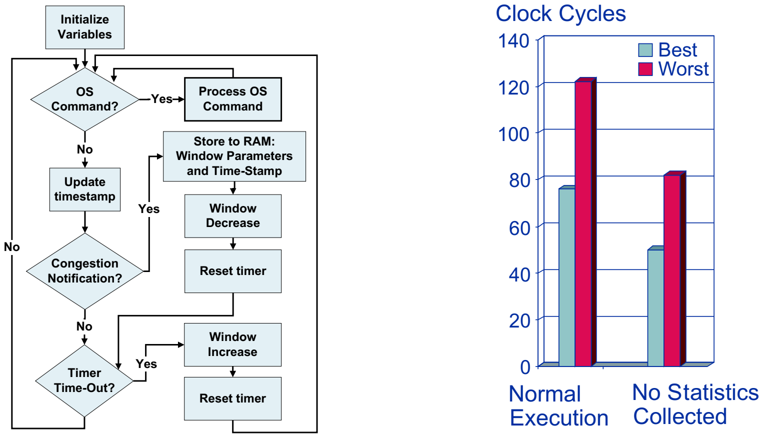
(a) Congestion controller area vs buffer monitors.

(b) Congestion controller area vs history depth.

Figure 6. Congestion controller implementation details in UMC 0.13 STD cell technology.

The hardware composed of *buffer monitors* and *congestion monitors* is optimized, so that the congestion detection and notification only takes 3 clock cycles. All the remaining cycles reported in Figure 7(b) are spent in software on the μC local to the source of congestion to compute window sizes with AIMD, perform I/O accesses or respond to requests from the central operating system. With statistics collection and reporting enabled, the average response latency to congestion is about 100 clock cycles, with a variance of 20.

On our system the packet payload length can vary between 0 and 256 data words (16-bits). The packet travel time is proportional to the number of hops in the system and to the payload length. For packets longer than 100 data words, the congestion control system reacts in a time that is inferior to the packet travel time. For more reasonable packet lengths, the latency of the congestion control is in the order of magnitude of the flight time of 10 packets. The system is therefore fast enough to rapidly counteract when congestion is building-up in the NoC.



(a) Control loop on μC.

(b) Reaction latency.

Figure 7. Control loop on the μC and impact on congestion control latency.

8. Conclusion

This paper discusses distributed congestion control for packet-switched networks-on-chip. Early signs of congestion are detected by distributed hardware modules and traffic re-shaping techniques are used to reduce congestion accordingly. The traffic shaping is based on a sliding-window mechanism for which the window size is computed by software running on distributed microcontrollers that implement simple operating system support on our platform.

The hardware and software extensions we propose to our system provide a low-latency, low-area congestion control solution. The area overhead of the hardware extensions is negligible compared to the area of the packet-switched router (only 5% of the router area in UMC 0.13 technology). The latency of the congestion control system is about 100 clock cycles which is low enough to react to congestion building-up in the system.

Both hardware and software extensions are very flexible and parameterizable to allow the implementation of many different types of control algorithms. It is possible to change the parameters and algorithms at run-time from the central operating-system on the platform. Future work will focus on determining optimal parameters and control algorithms depending on traffic patterns.

References

[1] T. Marescaux, V. Nollet and al., ”Run-time support for heterogeneous multitasking on reconfigurable SoCs”, in Integration VLSI Journal, Elesvier Science Publishers, 2004.

[2] V. Nollet, T. Marescaux, D. Verkest, J-Y. Mignolet, S. Vernalde: ”Operating System controlled Network-on-Chip”, Proc. of the Design Automation Conference, San Diego, June 2004, pp. 256-259.

[3] K. Goossens, J. Dielissen, and A, Rădulescu, ”The Æthereal Network on Chip: Concepts, Architectures, and Implementations”, in IEEE Design and Test of Computers, September 2005.

[4] A. Adriahantenaina et al., ”SPIN: A Scalable, Packet Switched, On-Chip Micro-Network”, DATE Conference, 2003.

[5] M. Millberg, E. Nilsson, R. Thid, S. Kumar, and A. Jantsch, ”The Nostrum backbone - a communication protocol stack for networks on chip”, in Proceedings of the VLSI Design Conference, January 2004.

[6] P. Avasare, V. Nollet, J.-Y. Mignolet, D. Verkest, H. Corporaal, ”Centralized End-to-End Flow Control in a Best-Effort Network-on-Chip”, in Proceedings of EMSOFT, September 2005.

[7] D. Bansal and H. Balakrishna, ”Binomial Congestion Control Algorithms”, INFOCOM 2001.