The Connection-Then-Credit Flow Control Protocol for Heterogeneous Multicore Systems-on-Chip

Nicola Concer, Luciano Bononi, Michael Soulié, Riccardo Locatelli, and Luca P. Carloni, *Senior Member, IEEE*

***Abstract*—Connection-then-credits (CTC) is a novel end-to-end flow control protocol to handle message-dependent deadlocks in best-effort networks-on-chip (NoC) for embedded multicore systems-on-chip (SoCs). CTC is based on the classic end-to-end credit-based flow control protocol but differs from it because it uses a network interface microarchitecture where a single credit counter and a single input data queue are shared among all possible communications. This architectural simplification reduces the area occupation of the network interfaces and increases their design reuse; for instance, the same network interface can be used to connect a core independently of the number of incoming and outgoing communications. CTC, however, requires a handshake preamble to initialize the credit counter in the sender network interface based on the buffering capacity of the receiver network interface. While this necessarily introduces a latency overhead in the transfer of a message, simulation-based experimental results show that the penalty in performance is limited when large messages need to be transferred, thus, making CTC a valid solution for particular classes of applications such as video stream processing.**

***Index Terms*—End-to-end flow control, message-dependent deadlock, multicore systems-on-chip (SoC), network interface design, networks-on-chip (NoC).**

I. Introduction

FUTURE GENERATIONS of systems-on-chip (SoCs) will consist of heterogeneous multicore architectures with a main general-purpose processor, possibly, itself consisting of multiple processing cores, and many task-specific subsystems that are programmable and/or configurable [1], [2]. These subsystems, which are also composed of several cores, will provide more power-efficient support to important classes of embedded applications across multiple use-case scenarios [3]–[5]. Heterogeneity is the combined result of hardware specialization, reuse of intellectual property modules, and the application of derivative design methodologies [6]. Programmability makes it possible to upgrade dedicated software and add the support of new applications and features that were not included at the chip design time.

Some current SoCs already offer task-specific subsystems such as media accelerator subsystems including heterogeneous and specialized cores (e.g., video and audio decoders) that are connected to a shared bus and communicate through the global memory [6]. This approach, however, offers limited programmability and reuse and does not optimize the utilization of the resources (e.g., the buffering queues in the core interfaces are typically sized for the worst-case scenario). Instead, communication among these cores in future SoCs will likely be based on the network-on-chip (NoC) paradigm [8]–[10]. These NoCs will also be heterogeneous, e.g., they may combine regular topologies with the insertion of dedicated long-range links [11] or may combine circuit-switched networks [12], [13] with packet-switched networks [14]. Further, as future SoCs host more specialized subsystems, the NoCs will become hierarchical: e.g., as illustrated in Fig. 1, a top-level network connects the main components of the chip, while other subnetworks support auxiliary subsystems such as the media accelerator.

In some of these task-specific subsystems, the communication will be based on the *message-passing* paradigm, which improves the performance of many important tasks such as the processing of video/audio streams and other multimedia applications [15]. While being of general applicability, the end-to-end flow control protocol that we present in this paper is targeted to the design of the subnetwork interconnecting the cores of a message-passing subsystem. The goal is to optimize the flexibility and reusability of the cores through the definition of a unified network interface (NI) design.

Network interfaces are crucial components for design and reusability in the NoC domain because they decouple the design of the cores from the design of the network. NIs implement the NoC communication protocols and improve performance by providing elasticity between inter-core communication tasks and intra-core computation tasks thanks to their data storage capabilities. As shown in Fig. 2, input and output queues are used to temporarily store the incoming and outgoing messages, respectively.

The message sizes may vary during the execution of one particular application, which typically requires the establishment of multiple connections to support different traffic pat-

Manuscript received September 21, 2009; revised January 8, 2010. Date of current version May 21, 2010. This work was supported in part by the University of Bologna, Bologna, Italy, with the project "Modeling and Analysis of Network Protocols for Spidergon-Based Networks on Chip," by STMicroelectronics, Grenoble, France, the National Science Foundation, under Award No. 0541278, the Gigascale Systems Research Center, which is one of the six centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program. This paper was recommended by Associate Editor A. Jantsch.

N. Concer and L. P. Carloni are with the Department of computer science, Columbia University, New York, NY 10027 USA (e-mail: concer@cs.columbia.edu; luca@cs.columbia.edu).

L. Bononi is with the Department of Computer Science, University of Bologna, Bologna 40127, Italy (e-mail: bononi@cs.unibo.it).

M. Soulié and R. Locatelli are with STMicroelectronics, Grenoble F-38019, France (e-mail: michael.soulie@st.com; riccardo.locatelli@st.com).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2010.2048592

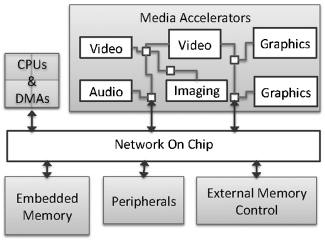


Fig. 1. NoC-based system-on-chip and the media-accelerator subsystem [7].

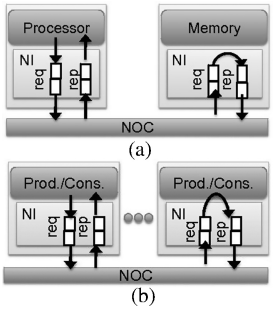


Fig. 2. Network interfaces connecting cores to the NoC and possible message dependencies in (a) shared-memory and (b) message-passing communication paradigms.

terns across the cores of the SoC, and even more across the execution of multiple applications that may be simultaneously running on the same SoC. Messages are the units of transfer at the application level among the SoC cores. In each NI, a message is typically broken down into a sequence of smaller packets, which are the units of transfer for routing purposes (Fig. 3). The size of a packet is either fixed or varies dynamically within a limited range of values. A packet may be further segmented in flow control units (flit) for more efficient allocation of link-level network resources, i.e., the buffering space of the queues that are present in the NIs and the routers, and the bandwidth of the links connecting them [16], [17].[1] The transfer of a long message in a packet-switched NoC normally entails the occupation of its resources for multiple consecutive clock cycles.

The correct operations of a network requires to efficiently handle deadlock situations that may arise due to the circular dependencies on the network resources that are generated by in-flight messages. A variety of methods has been proposed in the literature to either avoid or recover from deadlock [16], [18]. Most of these methods rely on the *consumption assumption*, thereby, the packets of a message traversing the network are always consumed by the destination core once they reach its corresponding network interface [19]. However, as shown in Fig. 2 and discussed in detail in Section II, deadlock may be caused also by dependencies that are *external* to the network, i.e., *internal* to a core. In fact, regardless of the communication paradigm (shared-memory or message-passing) a SoC core typically generates new messages in response to the reception of a previous one. These dependencies between

[1]This is typical for packet-switched NoCs where the size of a flit measured in bits often coincides with the parallelism of a link measured in terms of the number of wires that implement it, i.e., a flit often coincide with a physical-transfer unit (phit) [14].

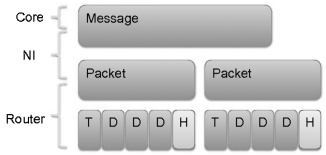


Fig. 3. Messages generated by cores are partitioned into packets by the NIs and then in flits. Each packet has a header that contains the routing information used by the routers to compute the path toward the packet's destination.

messages can generate a different type of deadlock called message-dependent (or protocol)[2] deadlock [15], [19], [20]. Classic approaches to address message-dependent deadlocks rely either on assigning physically-separated [21] (or virtually-separated [6], [22]) networks to different message classes or on using an end-to-end flow control protocol to regulate the access to the NoC [23], [24]. While the first solution is suitable to the shared-memory paradigm, systems that adopt a message-passing paradigm benefit from the implementation of a credit-based (CB) protocol [25], which is discussed in Section III.

Motivated by the trends in the design of heterogeneous multicore SoCs, we build on the CB approach to develop connection-then-credits (CTC), an end-to-end flow control protocol that allows us to handle message-dependent deadlocks while simplifying the design of the network interface. As explained in detail in Section IV, the microarchitecture of a CTC network interface uses a single credit counter together with a single output queue to send all the possible outgoing messages and a single pair of data and request queues that are shared across all possible incoming messages. On the other hand, CTC requires the completion of a handshake procedure between any pair of cores that want to communicate before the actual message transfer starts. In Section V, we analyze the benefits and costs of the proposed solution by presenting a comprehensive set of experimental results and we compare three possible alternative implementations of the NIs implementing the CTC protocol. The current version of the CTC protocol is compatible with best-effort NoC implementations that use simple input-queued wormhole routers [26]. Finally, Section VI discussed related work and outlines future work, including the support of quality-of-service features, such as guaranteed latency or throughput for selected communications, through the design of connection-aware routers.

II. Message-Dependent Deadlock

There are two main communication paradigms for exchanging data among the processing cores of a multicore SoC and they are associated to two corresponding programming models: shared-memory and message-passing.

In a shared-memory paradigm, the processing cores communicate via data variables that are defined in the same logical memory space and are physically stored in one or

[2]Message-dependent deadlock occurs at a level of abstraction that is higher than routing-level deadlock, which can be addressed by deadlock-free routing algorithms such as dimension-order routing [16], [17] We focus on addressing message-dependent deadlock while assuming the use of a deadlock-free routing algorithm. Notice that message-dependent deadlock is different from *application-level deadlock* which is out of the scope of this paper.

more memory cores. As shown in Fig. 2(a), a processor accesses a memory through either a load or a store request by specifying the memory address and the size of the data block to be transferred. In the case of a load request, the addressed memory replies by sending the values of the requested block of data (typically a cache line) to the processor, which saves them in its local cache memory. In the case of a store request, the memory receives new values for a block of addresses, which typically corresponds to a line in the processor's local cache, and it replies by generating a short acknowledgement message to confirm their correct delivery. Shared-memory is the most used paradigm in current multicore SoCs.

In the message-passing paradigm, which is illustrated in Fig. 2(b), the processing cores communicate by sending/receiving data that are pushed directly from a core to another (peer-to-peer communication): the sending and receiving cores are commonly referred as the *producer* and *consumer*, respectively. By having dedicated logical addressing space for each processing core and providing direct communication among their physical local memories, message-passing avoids the issues of shared-memory coherency and consistency [27], thus, potentially reducing the communication latency of each data transfer. This paradigm is particularly suited for multimedia applications, where a system is typically constructed as a pipeline of processing elements which pass streams of data (e.g., video frames, audio frames) from one element to the next [2], [15]. While message passing can be implemented by the operating system on top of a physically centralized memory architecture, the trend in the design of multicore architectures for multimedia applications is to have an increasing number of processing cores exchanging streams of data through a distributed memory organizations, where each core has access to a local memory [28].

The correct implementation of shared-memory and message-passing paradigms in a SoC requires an underlying NoC with communication protocols that guarantee the correct transfer of each message and, particularly, the absence of deadlock. As discussed in Section I, even if the NoC relies on deadlock-free routing algorithms, message-dependent deadlock may arise due to the dependencies among the messages "inside a core," which are shown in Fig. 2: e.g., the dependence between a load request and response in a memory for the shared-memory paradigm and the causal dependency between the consumption and production of data in a core for the message-passing paradigm. For both paradigms, the dependencies between pairs of messages may get combined, thus leading to *message dependency chains* [29]. Indeed, the causal relations among pairs of messages can be modeled as a partial order relation $\prec$ over the set of all possible messages that are transferred in the network. Message dependency chains depend on the chosen communication paradigm and the characteristic of the given application.

Example: Fig. 4 shows an example of a message-dependent deadlock occurring in a packet-switched NoC with wormhole flow control and without virtual channels (VCs). Assuming a routing-level deadlock free interconnect, the system falls in a message-dependent deadlock caused by the dependences between the messages received and sent by the cores of the

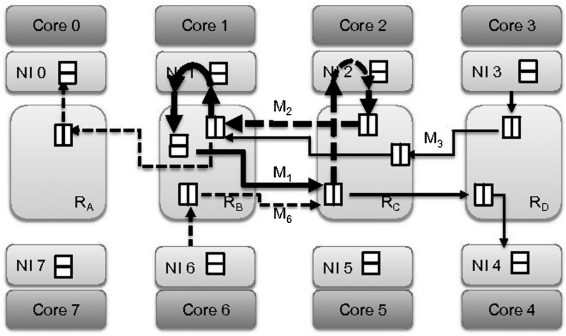


Fig. 4. Example of message-dependent deadlock in a multicore SoC with a message-passing communication scenario.

SoC that communicate through a message-passing paradigm. One stream of data, which is represented with the continuous arrow lines, traverses the network flowing from $core_3$ to $core_1$ and from $core_1$ to $core_4$. Meanwhile, a second stream of data, which is represented with the dashed arrow lines, goes from $core_6$ to $core_2$ and from $core_2$ to $core_0$. Since the input and output queues of all NIs have necessarily limited storage capacity, the transfer of a long stream of data may cause a backpressure effect to propagate backwards into the NoC. For instance, assuming that the rate at which $core_2$ processes the incoming flits of a long message, M_6 sent by $core_6$ is slower than the flits' arrival rate. Eventually, these flits will accumulate first in $core_2$'s network interface and then in the buffering queues of the NoC across the path between the two cores. In particular, they may fully occupy the input queue of the West port of router R_C, thereby, blocking a message M_1 that is attempting to reach $core_4$ from $core_1$. Since message M_1 cannot advance, eventually router R_B will backpressure network interface NI_1 to prevent it from injecting additional M_1's flits. This may stall $core_1$ with the consequence of delaying the completion of the transfer of message M_3 from $core_3$. Hence, some of M_3's flits may be buffered in the input-queue of the East input port of router R_B. But, in turn, this may prevent the progress of message M_2, which needs to traverse router R_B as it goes from $core_2$ to $core_0$. The flits of M_2 would then be blocked in the North input port of router R_C, which would end up exercising backpressure on network interface NI_2. Finally, this may lead to a stalling of $core_2$ with the consequence of interrupting completely the processing of the flits of message M_6: as the chain of dependency turns into a cycle, the SoC falls into a deadlock. □

Similarly, to routing-dependent deadlock, the message-dependent deadlock problem can be addressed with either avoidance or recovery strategies. The relative advantages of the various techniques based on these two approaches depend on how frequently deadlocks occur and how efficiently (in terms of resource cost and utilization) messages can be routed while guarding against deadlocks [19].

The introduction of a virtual network (VN) for each type of message transfer guarantees the solution of the message-dependent deadlock by satisfying the consumption assumption [6], [19]: the input and output queue of each router and each NI in the network is replicated and assigned to a single specific *message class*. For instance, two message

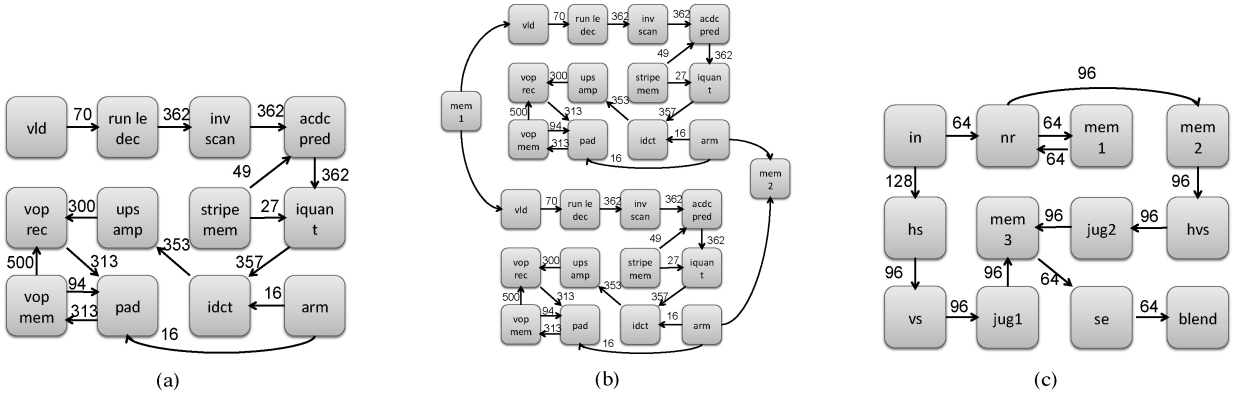


Fig. 5. Task graphs for three multimedia applications: (a) Video object plane decoder (VOPD), (b) Double video object plane decoder (DVOPD), and the (c) Multiwindow display (MWD) task graphs.

classes can be used to distinguish between memory requests and reply messages, while more message classes are necessary to implement cache coherence in a system with distributed shared memory [30].[3]

By introducing dedicated resources for each message class a VN-based implementation allows the removal of the impact of causal dependency relations between messages that belong to different classes on the operations of the network at the cost of a higher buffer requirement and more complex router and network-interface design. This cost, however, may become too high for classes of application generating long message-dependency chains such as stream processing applications. These applications are often implemented with a pipeline of processing cores, where each core produces data for the next consumer core. This leads to potentially long dependency chains of message requests $request_1 \prec \cdots \prec request_n$, where n is the number of cores in the pipeline. For example, Fig. 5 shows the task graphs of three multimedia applications where the nodes represent the computational tasks while the arcs represent the communication between these tasks with the relative bandwidth requirements [31]–[33]. Notice that the task graphs of these applications, which fall in the stream processing class, are similar as they present a fairly-linear pipeline structure with a sequence of multiple stages, each stage corresponding to a task. For these applications, an optimally-concurrent SoC implementation where each task is mapped to a distinct processing core and where each intertask communication is implemented by end-to-end message-passing would lead to as many message classes as the number of arcs. Furthermore, as discussed in Section I, multicore SoCs for embedded products simultaneously support an increasing number of such streaming applications [1], [2], [34]. This translates into the presence of complex communication patterns among the cores, which run multiple threads of computation to implement the multiple tasks of the various applications. In this scenario, an implementation based on virtual networks does not scale well because: 1) the number of distinct message types that travel on the network continues to grow; and 2) the length of the dependency chains is difficult to predict at design time since it depends on the particular subset of applications that are executed at any given time during the SoC operations. The CB flow control protocol, which is discussed in next section, is a possible solution, while the CTC protocol, which we propose in Section IV, offers additional flexibility and reusability with respect to the CB approach.

[3]For instance, a VN-based implementation of the MOESI cache coherence protocol, which defines four different message classes [20], [21], requires four different VNs.

III. CB End-to-End Protocol

An *end-to-end* flow control protocol addresses the message-dependent deadlock by guaranteeing that a sender NI does not ever inject more flits in the network than the corresponding receiver NI can absorb. The CB end-to-end flow control protocol is a simple implementation of this idea that has been used in various NoC designs [23], [24], [35].

With a CB protocol, the sender NI maintains a precise knowledge of the number of queue slots that the receiver NI has still available through the exchange of end-to-end transfer *credits*. A credit is associated with one or more flits depending on the desired level of granularity. What is important is the guarantee that no fragment of a sent message can remain blocked in the NoC due to lack of space in the input queue of the receiver NI, with the potential risk of causing a deadlock situation. Hence, the sender NI can continue to *inject* flits in the network only if it has still enough credits as proofs that the receiver NI will *absorb* these flits. Dually, the receiver NI must send a credit back to the sender NI for each flit that its core has consumed, thereby generating an empty slot in its input queue.

Generally, a single consumer core can be addressed by multiple producers. Also a producer can address multiple consumers and for each of these the producer needs a dedicated credit counter. Differently from CB flow control mechanisms that operate at the link level between a pair of interconnected routers [16], [17], here a pair of communicating cores may be separated by multiple hops in the network. Also, all packets generated by the peer cores arrive at the same NIs input port. Fig. 6 shows the simplest way to address this issue. The NI of each core is provided with multiple and independent input and output queues and credit counters: there is one input queue for each possible sender NI that may send messages to this core; there are also one output queue and one credit counter for each possible receiver NI that may be addressed by this core. In particular, a given receiver network interface NI_r saves the incoming flit that has received from another network

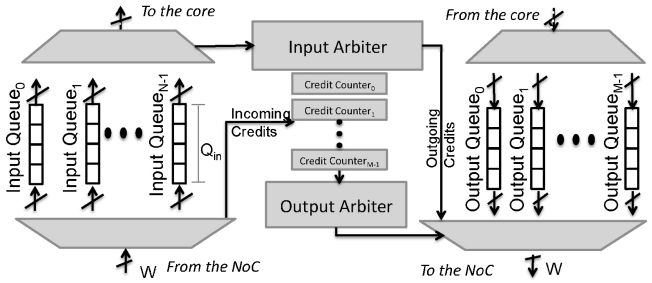


Fig. 6. Block diagram of network interface supporting the CB protocol.

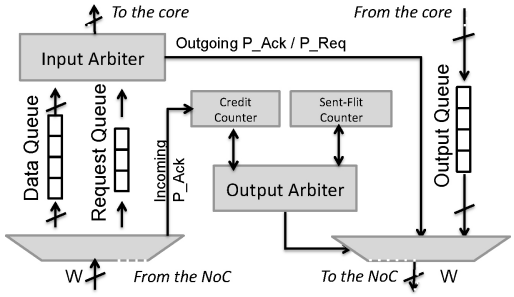


Fig. 7. Block diagram of network interface supporting the CTC protocol.

interface NI_s into the input queue Q_s associated to NI_s. Then, whenever the core of NI_r reads a flit from Q_s, NI_r generates an end-to-end flit-credit that is sent back to NI_s. Upon reception of this credit NI_s updates the credit counter C_r associated to the destination NI_r. The output arbiter of NI_s decides which flit to inject into the NoC based on the current status of each of its output queues and their corresponding credit counters. Multiple credits can be combined into one single end-to-end credit message for better efficiency. As discussed in Section IV-A, the number K of flit-credits associated to a single credit-message and the size of the queues may have a relevant impact on the performance of the NoC.

The CB end-to-end flow control protocol differs from the solution based on VNs for two main reasons. First, a VN-based NoC design requires that all the queues, including those in the routers, must be replicated as many times as the number of VNs. Instead, the CB protocol requires that only the queues of the network interfaces must be replicated. Second, with a VN-based NoC the number of queues per physical link depends on the length of the application message-dependency chain, which may not always be easy to determine at design time. With the CB protocol, instead, this number may vary for each given network interface depending on the number of the other network interfaces that exchange data with it.

On the other hand, adding a dedicated input (output) queue to the network interface for each possible sender (receiver) of messages forces engineers to design a specific network interface for each node of the network, thus, leading to adhoc network-interface designs that are poorly reusable. Further, if all possible communication scenarios are difficult to determine at design time, CB protocols may lead to the over-provisioning of the NI in terms of queues. Particularly, this is the case for multiuse-case SoCs, where the interaction between cores is driven by the applications run by the user [4], [5]. These considerations motivated us to develop the CTC end-to-end flow-control protocol as an alternative to the CB flow-control protocol. As discussed in the next section, CTC rationalizes and simplifies the design of NIs, while guaranteeing the absence of message-dependent deadlock.

IV. CTC Protocol

CTC regulates the exchange of messages between two peer NIs by introducing an handshake-based procedure called connection. Specifically, a new distinct connection is set up for each data message that a sender NI needs to send to a receiver-NI interface before the actual transfer of data occurs.

Fig. 7 shows the block diagram of a network interface supporting the CTC protocols. The CTC NI contains *two* input queues and *one* single output queue *independently* from the number of network interfaces that may require a connection with this NI or the number of cores that this NI may address. The data queue Q_{in} is used to store incoming data flits while the request queue Q_{req} is used to store incoming connection requests. Additionally, the *output arbiter* uses two counters to account for the available storage on the connected peer network interface and the number of flits that remain to be sent before terminating the current connection. The CTC protocol consists of the following main steps.

1) To initiate a connection with a peer-receiver network interface NI_r a sender network interface NI_s sends a special message consisting of a single packet P_REQ (packet-request) to NI_r. The P_REQ packet specifies the source and the total size of the data message to be delivered.
2) Upon the reception of a P_REQ packet, NI_r stores the request in Q_{req} together with the other requests previously received and not yet processed.
3) After the core associated to NI_r has completed processing some of the data previously received and, therefore, queue Q_{in} has enough free space to accept a new message, NI_r selects the next connection request among those pending in Q_{req} and generates an acknowledgement packet called P_ACK.
4) Once delivered to the selected source NI_s, the P_ACK packet initializes the output credit counter to a value $I = Min\{S, M\}$, where S is the available current storage in NI_r's data queue and M is the size of message generated by NI_s measured in flits. Hence, after receiving the first P_ACK, NI_s's credit counter is initialized with as much available storage as it is necessary to the message.
5) Upon the reception of the first P_ACK packet, NI_s generates a data packet that consists of a *header* flit followed by a given number of data flits (Fig. 3). The header flit opens a path through the routers of the NoC toward NI_r based on a wormhole flow-control mechanism [16].[4]
6) After consuming a fixed number K of flits from its input port Q_{in}, a NI_r generates a new P_ACK to update Ni_s credit counter until it has sent enough flit-credits to transfer the whole message.

[4]CTC is orthogonal to the routing algorithm and the network topology. For the experiments presented in Section V, we used a Spidergon NoC with a distributed, minimal deterministic routing algorithm [6].

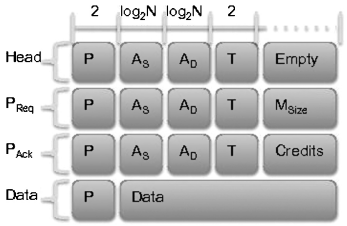


Fig. 8. Formats of header, P_REQ, P_ACK, and data flits.

7) Once the connection is set up, whenever a NI_s receives a P_ACK packet from NI_r it increments its *Credit Counter* by the number K of flit-credits specified by P_ACK. Dually, whenever NI_s injects a data flit into the NoC NI_s it decreases both the Credit Counter and the *Sent-Flit Counter* by one unit. The latter accounts for the total number of flits that remain to be sent to complete the transfer of the message for the current connection.
8) NI_s terminates each data packet by tagging the last data flit as a *tail-flit*. This may happen for one of three possible reasons.
   a) All the flits of the message have been sent and the connection can be terminated. As both sender and receiver NIs know the size of the message to transfer, no further steps are needed.
   b) The number of sent flits reaches a given maximum threshold $P_{\max}$, which is an upper-bound for the size of a data packet. This value, which is a parameter of the given NoC implementation, is used to avoid *starvation* issues: by limiting the size of a packet we obtain a fair sharing of the NoC resources among different connections because the path along the NoC is made available to other possibly-blocked packets. Then, if more flit-credits are available, the NI_s creates a new packet by generating a new header-flit before starting the injection of new data flits.
   c) NI_s runs out of end-to-end flit-credits. In this case, the current data packet is terminated with a tail-flit even if the number of its injected data flits has been less than $P_{\max}$. Then, the NI remains idle until new credits arrive. Notice that the CTC connection remains open until the delivery of all the flits of the current message is completed.

Example: Fig. 9 illustrates the operations of the CTC protocol with three snapshots.

1) Network interfaces NI_0 and NI_2 address NI_1 with two P_REQ packets requesting the transfer of two messages of 100 and 80 flits, respectively.
2) NI_1 selects NI_0 to initiate the connection while it stores the other request in the *request queue*. Assuming that $K = 5$ and that the space currently available in the *data queue* is $S = 10$, NI_1 first generates a P_ACK containing S flit-credits used to initialize NI_0's credit counter with the maximum available storage.
3) Assuming $P_{\max} \geq 10$, a data-packet with 10 flits is injected in the NoC by NI_0 to be later absorbed and stored in the data queue of NI_1. Eventually, NI_1 will generate a total of $\lceil (M-S)/K \rceil + 1 = 19$ P_ACK packets for this connection to enable the transfer of 100 flits from NI_0.

CTC does not impose a specific flit width but requires a particular format for the various flits generated by the NIs with some fixed-width fields (Fig. 8), including the following:

1) position (P)—a two-bit field with sideband signal indicating the position of the flit in the packet, i.e., HEAD, TAIL, BODY or SINGLE (for unit-flit packets);
2) address (A)—a $(\log_2 N)$-bit field indicating the address of the NI;
3) type (T)—a two-bit field indicating the flit type: HEADER, P_REQ, P_ACK or DATA;
4) the remaining fields are dedicated to data or to special functions, such as indicating the size of the message to be sent (M_{Size}) in a P_REQ packet or the number of credit (*Credits*) associated to a P_ACK packet.

Notice that the width of the request-queue Q_{req} is significantly smaller than the width of the data queue. In fact, for each incoming P_REQ an NI needs to store only the source address and the size of the message to transfer. In an NoC with 64 nodes, a core can be identified with $\log_2 64 = 6$ bits. The remaining bits can be used to represent the message size. While flit widths keep increasing [36] (e.g., up to 64/128 bits), a counter of 10 bits allows the transfer of a message with size up to $64 \times 2^{10} = 64$-kbit in a single connection.

Similarly to the flit-credits in a CB protocol, the P_ACK packets are flow-control packets that transport a number of end-to-end flit-credits. The difference between CTC and CB is that in the CTC protocol the first generated P_ACK of a given connection actually *initializes* the output credit counter of NI_s to the number of free slots available on the peer node. In the CB protocol, instead, each counter is initialized at the system start-up time and incoming flit-credits are used only to increment the corresponding counter value.

As we indicated above, an NI_r accepting a new P_REQ first generates a single P_ACK with the total space S available on the data queue Q_{in}. Once the connection is established, NI_r starts receiving flits and its core starts processing them. For every subset of K flits that are processed by the core, thereby freeing the corresponding space in Q_{in}, NI_r generates a new P_ACK packet until the total number of generated credits is sufficient to transfer all the M flits of the message.

Since the value of K is fixed for a given NoC implementation, for any connection request P_REQ made by a sender NI_S the receiver NI_r can generate up to ζ P_ACK packets, where

$$\zeta = \left\lceil \frac{M}{K} \right\rceil. \tag{1}$$

The values of parameters K and $P_{\max}$ are fixed for a given implementation of the CTC protocol and determined at design time. On the other hand, the length M of a message that is transferred during a connection is not known at design time and typically changes across different connections. Some remarks are in order.

1) Since the receiver NI_r sends credits in chunks of size K, it inevitably assigns to the sender NI_s a number of

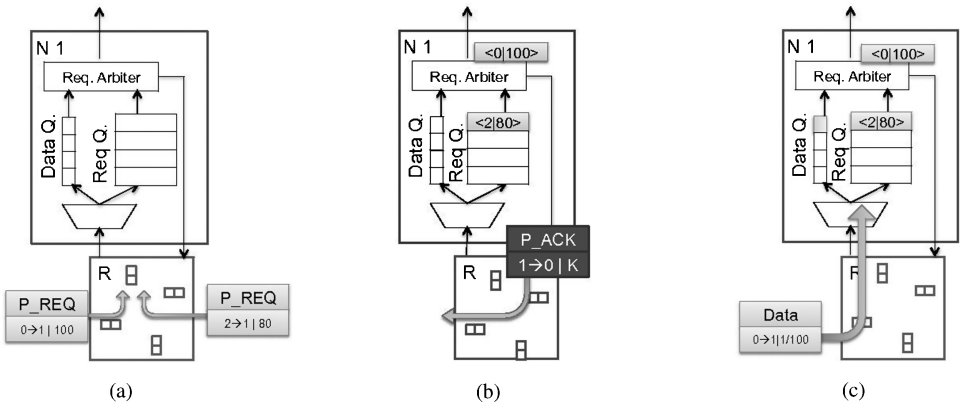


Fig. 9. Example of CTC-based NoC operations: (a) Core interface NI_1 receives two P_REQ requests from NI_0 and NI_2, (b) NI_1 selects the request from NI_0 and generates the corresponding P_ACK while storing the request from NI_2 in the request-queue, (c) NI_1 receives the data-packets from NI_0.

data-queue slots that is larger than necessary whenever a message's length M is not a multiple of K. Notice, however, that at most $K-1$ slots of Q_{in} can be reserved without being actually used. These additional slots are freed when the connection is terminated by the reception of the tail flit.

2) If the sender NI_s runs out of credits before sending all the $P_{\max}$ flits or whenever M is not a multiple of $P_{\max}$, a packet is terminated by tagging the last flit to be sent as a tail-flit; since $P_{\max}$ is just an upper-bound for the size of a packet, it may happen that a given message of length M is transferred through a longer sequence of smaller packets than what would be possible in absence of congestion.

TABLE I
SYMBOLS USED IN THE CTC ANALYTICAL MODEL

Symbol	Description
M	Peer-to-peer message length
K	Number of credits carried by each P_ACK
Q_{in}	Consumer input queue size
$P_{\max}$	Maximum packet size threshold
R	Router pipeline length
$h(NI_s, NI_s)$	Distance (in hops) between two NIs
$\psi(NI_s, NI_s)$	Aggregate number of pipeline stages between two NIs
ζ	Number of P_ACK generated by the consumer to complete connection
ρ	Processing rate (flit/cycle) of a core
ϵ	Delay for processing a P_REQ
π	Minimum number of packets generated per message
δ	End-to-end delay for a single flit-packet

A. Protocol Correctness

An end-to-end flow control addresses the message-dependent deadlock by guaranteeing the consumption assumption of all injected flits of the supported application. This makes the NoC design orthogonal to the supported application, which, therefore, has no constraints on the communication complexity and message dependencies. Nevertheless the dependencies between the data and the control packets (P_REQ and P_ACK) generated by the end-to-end protocol must be properly handled.

The CTC protocol defines the following three message dependencies:

1) P_REQ → P_ACK;
2) P_ACK → *Data*;
3) *Data* → P_ACK.

Since, these are all the possible dependencies that can occur in the system, properly accounting for them guarantees the absence of message-dependent deadlock. The three types of message dependencies are handled by CTC in the following way.

1) P_REQ → P_ACK: To guarantee the consumption assumption of the P_REQ packets, the request queue must be sized accordingly to the maximum number of requests that a NI can receive. In the current version of CTC, a sender network interface NI_s is limited to have *one* single outstanding P_REQ at time. Hence, in a worst-case scenario such as an all-to-all communication pattern, the size of Q_{req} grows linearly with the size of the NoC. By sizing Q_{req} in this way all incoming P_REQ packets are guaranteed to be stored in the receiver NI_r and removed from the NoC.
2) P_ACK → *Data*: P_ACK packets are always consumed by a sender NI_s, which processes them in order as soon as they arrive to increment the credit counter, without the need of storing them.
3) *Data* → P_ACK: the credit mechanism ensures that no more data-flits than those allowed by the credit counter can ever be injected. Hence, all data flits injected into the NoC are guaranteed to be absorbed and stored in the receiver NI_r as soon as they reach their destination.

B. Analytical Model

Table I summarizes the variables that we use to model the performance of the CTC protocol analytically. Two of these variables are actually design parameters that are fixed for any given CTC implementation.

1) K is the number of flit-credits per P_ACK packet: this parameter is related to the size of the input queue in a receiver network interface NI_r and to the number of control packets generated per single message.
2) $P_{\max}$ is the maximum size of a packet measured in flits: this value affects the number of packets that must be generated to complete a connection.

To minimize the length of a connection between cores, a sender NI_s should be able to generate a continuous flow of data flits and the receiver NI_r should be able to absorb and process this flow. Hence, it is important to guarantee that the source NI_s never runs out of end-to-end credits. This performance requirement can be satisfied by sizing properly the input data queues of the receiver NI_r. This requires to account for both the value of K and the round trip time between a receiver NI_r and all its possible peer sender network interfaces. The zero-load latency δ measured in clock cycles that is taken by a flit to traverse the NoC from a given source NI_s to reach a given receiver NI_r is equal to

$$\delta(\mathrm{NI}_s, \mathrm{NI}_r) = h(\mathrm{NI}_s, \mathrm{NI}_r) \cdot R + \psi(\mathrm{NI}_s, \mathrm{NI}_r) + 2$$

where $h(\mathrm{NI}_s, \mathrm{NI}_r)$ is the distance in hops between the two nodes, R is the number of clock cycles used by routers to process and forward a flit, $\psi(\mathrm{NI}_s, \mathrm{NI}_r)$ captures the aggregate number of wire pipeline stages across all the links on the path between NI_s and NI_r, and the value "2" accounts for the two local hops between NIs and routers at the sender and receiver nodes. The latency R of each router depends on the specific architecture of the routers composing the NoC and the clock frequency at which they work. This value generally ranges in an interval between one and five clock cycles [26]. Wire pipelining is often necessary to meet the target clock frequency in the implementation of a synchronous NoC [37], thus making the latency of traversing a link increase linearly with the number of clocked repeaters that have been inserted on the link. The zero-load round-trip time taken by a flit to traverse the NoC from NI_r to NI_s and back is given by

$$T_{\mathrm{rtt}}(\mathrm{NI}_r, \mathrm{NI}_s) = \delta(\mathrm{NI}_r, \mathrm{NI}_s) + \epsilon_s + \delta(\mathrm{NI}_s, \mathrm{NI}_r)$$

where ϵ_s indicates the processing delay of the credit at NI_s.

Putting it all together, the data queue Q_{in} of a given network interface NI_r should be sized according to the equation

$$Q_{\mathrm{in}}(\mathrm{NI}_r) = K + \max_{\mathrm{NI}_s \in \mathcal{N}} \left\{ T_{\mathrm{rtt}}(\mathrm{NI}_s, \mathrm{NI}_r) \right\} \quad (2)$$

where $\mathcal{N}$ is the set of the network interfaces that may require a connection to send data to NI_r. As discussed above, the size of Q_{in} is related to the parameter K because whenever NI_r sends a P_ACK packet with K credits it must have as many available slots for data flits in Q_{in}. The second element accounts for the impact of the longest round-trip time between NI_r and its peer network interfaces. In practice, this value can be set as equal to the maximum round-trip time between any pair of network interfaces in the NoC.

The zero-load latency taken by a connection to complete the transfer of a message of size M flits from a source NI_s to a receiver NI_r is

$$\begin{aligned} L_M(\mathrm{NI}_s, \mathrm{NI}_r) &= \delta(\mathrm{NI}_s, \mathrm{NI}_r) + \delta(\mathrm{NI}_r, \mathrm{NI}_s) \\ &+ \epsilon + \pi \cdot \left(\delta(\mathrm{NI}_s, \mathrm{NI}_r) + P_{\max} \right) \end{aligned}$$

where the term $\delta(\mathrm{NI}_s, \mathrm{NI}_r)+\delta(\mathrm{NI}_r, \mathrm{NI}_s)$ accounts for the round-trip delay of the P_REQ–P_ACK packets and ϵ is the number of cycles taken by the receiver NI_r to select and process the given P_REQ packet. The term $(\delta(\mathrm{NI}_s, \mathrm{NI}_r)+P_{\max})$ accounts for the serialization delay of the header flit followed by at most $P_{\max}$ flits. Finally, variable π captures the minimum number of packets into which a message of size M is decomposed and is equal to

$$\pi = \left\lceil \frac{M}{P_{\max}} \right\rceil. \quad (3)$$

The connection procedure and the header flits used to open a path in the routers of the network affect the maximum throughput that can be achieved by the protocol. On a single message connection, the maximum achievable throughput can be computed as

$$Th_M = \frac{M}{L_M}. \quad (4)$$

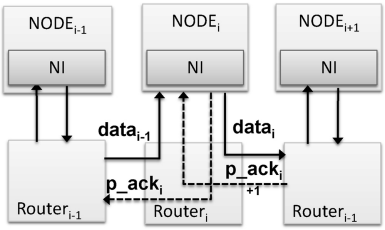


Fig. 10. Contention on node three between P_ACK and data flits.

C. Contention on the Input and Output Ports of a Network Interface

Regardless of whether the CTC protocol is used in combination with a shared-memory or a message-passing communication paradigm, a network interface can simultaneously act both as sender and receiver. This can generate a contention on the input and output ports of the network interface between the data packets (both incoming and outgoing) and the control packets that are used to regulate the communication (both P_REQ and P_ACK). Fig. 10 illustrates a simple example of this situation: network interface NI_{i-1} sends data to NI_i, which in turns sends data to NI_{i+1}. In this scenario, since NI_i acts both as a sender and receiver network interface, it may suffer from access contention on its input and output ports. In particular:

1) the data-flits to be sent to NI_{i+1} compete for access to the same output port with the P_ACK packets to be sent to NI_{i-1};
2) the data-flits coming from NI_{i-1} compete for access to the same input port with the P_ACK packets coming from the NI_{i+1}.

Recall that to minimize the communication latency and avoid idle cycles at the source (bubbles) due to lack of end-to-end credits, a receiver network interface NI_r should be able to provide its peer sender network interface NI_s with a new P_ACK each time the space for K data flits becomes available on its data queue. Let ρ denotes the rate at which data flits are "consumed" by the core associated with NI_r. Notice that $0 \leq \rho \leq 1$ because the NoC can deliver at most one flit per clock cycle to each NI. Hence, a P_ACK packet should be injected into the NoC every $K \cdot \rho$ cycles. This can be obtained by simply making the access of the output port preemptive for P_ACK packets with respect to the data-flits: whenever a new P_ACK packet is ready to be generated the data flit

of the current packet is tagged as a tail-flit so that in the following cycle the P_ACK can be injected. On the other hand, the preemptive procedure may increase the number of header-flits used to open the path through the NoC for the data flits. If it is possible to estimate the value of ρ with sufficient accuracy, then the value of the parameter $P_{\max}$ can be set according to the following equation:

$$P_{\max} = K \cdot \rho. \tag{5}$$

In Section V-A, we present an experimental analysis of the interaction between the values of the main CTC design parameters and the contention at the ports of the network interfaces.

V. Experimental Results

In order to analyze the characteristics of the CTC protocol and compare the performance of a CTC-based NoC versus a CB-based NoC, we developed a *C++* system-level simulator that allows us to model various NoC topologies, routing, and flow-control protocols as well as the traffic scenarios generated by various types of cores. We used the simulator to compare the two end-to-end protocols on the Spidergon NoC [6] using AFirst, a deterministic, minimal, and distributed routing algorithm [38]. Spidergon is a NoC interconnect whose topology is a ring enriched by cross links interconnecting opposite nodes. We considered the traffic scenarios for three distinct SoC applications (VOPD, DVOPD and MWD) from the literature [31]–[33] whose task graphs are reported in Fig. 5. We assigned each task to a different core following the heuristic presented in [39]. We also considered the uniform random traffic (URT) pattern where each node may communicate with any other node in the network [16].

In the CB-based NoC, the NI of each core has a number of input queues equal to the number of incoming streams (Fig. 5). On the input side, the *input arbiter* selects the incoming data to process, while on the output side, the *output arbiter* selects the queue that is used to forward the flit or to send a credit (see Fig. 6). The selection of the input and output queues is made on a packet basis according to a round-robin policy. For both the CTC and the CB-based NoCs, the size of each data input queue is set uniformly based on (2).

Fig. 11 shows the average end-to-end message latency as a function of the offered load for the given traffic patterns. In all cases, K is fixed to 32 flit-credits (results are similar for the other credits values) and the messages are 64 flits. As expected, the CTC protocol gives a higher latency due to the handshake procedure. Nevertheless, for the VOPD application the difference between the two protocols remains under 10% up to the saturation point, which is around 0.4. In the case of MWD, the difference raises close to 20%, while for DVOPD, the two protocols differ by 10%. Finally, in the URT traffic, the two protocols behave in a very similar way.

Fig. 12 shows the average peer-to-peer latency as a function of the number of credits K per P_ACK packet when the offered load is lower than the saturation threshold. Clearly, by increasing the value of K the performance of the system also improves: processing elements can inject more flits per P_ACK, thus reducing the number of control packets (credits and headers). Conversely, increasing K also requires bigger input queues that must support the additional number of flits received per P_ACK sent.

Fig. 13 reports the throughput comparison as a function of the message size M with $K = 4$. In all these scenarios, the performance of the CTC-based NoC increases with the message size for the same offered load because the rate of connections-per-flits that must be set up is reduced. The throughput of the CB-based NoC, instead, decreases as the size of each message increases because this effectively augments the number of times in which a P_ACK packet preempts the data packets. Therefore, CTC represents a valid proposition for message-passing applications such as video stream processing that present large intercore message transfers.

In Fig. 14, we analyze the amount of storage used by the two alternative flow-control protocols assuming a flit width $W = 64$ bits and the request width $R = 14$ bits (4 bits for addresses and 10 bits for message size) and $K = 4$. As discussed in Section III, for both CB-based and CTC-based NoCs the input queues of the network interfaces must be properly sized to avoid throughput degradation. For a CB-based NoC, each input queue must be sized accordingly to (2). For a CTC-based NoC, instead, only the data queue must have this size, while the request queue of a given interface must be as large as the *number* of distinct producer cores that can send message to its core. Notice that in order to provide a single interface design for each possible core, this number can be overestimated without a major loss of area because the request queue has a minor impact on the overall storage relatively to the data queue. For example, in the CTC case of Fig. 14, we assume that all nodes can be reconfigured to communicate with any other node by setting the number of request-queue slots equal to $N - 1$. Neither CTC nor CB instead impose a specific size for the output queue. In this example, we consider the output queue equal to the input ones. By doing so, we have that the request queue occupies 9% of the total storage used by CTC while it reaches 14% with an output queue length of a single slot.[5]

For the cases of VOPD, MWD, and DVOPD, only the interfaces associated to nodes with incident (outgoing) arrows actually require input (output) queues. In case of VOPD, the CTC-based NoC uses a total of 22 data queues, including both input and output queues, while the CB-based NoC needs 30 data queues. Hence, assuming that the length of each data queue is the same in the two NoCs, CTC saves up to 25% of storage space for this particular case study. In the MWD case, since most cores communicate only with one other core the two NoCs have many interfaces with similar structures. Still, even in this case, CTC saves up to 12% of storage as reported in Fig. 14. For DVOPD, which is a SoC implementing two VOPD in parallel [32], we reach 35% of storage savings.

Finally, the URT traffic pattern represents the special case, where each node may communicate with any other node of the NoC. Here, clearly, CB is an expensive solution as it requires $N-1$ queues, where N is the number of nodes in the network.

[5]To obtain the same flexibility in CB we need to replicate all pairs of input and output queues in the NI so that we have one pair for each node of the NoC.

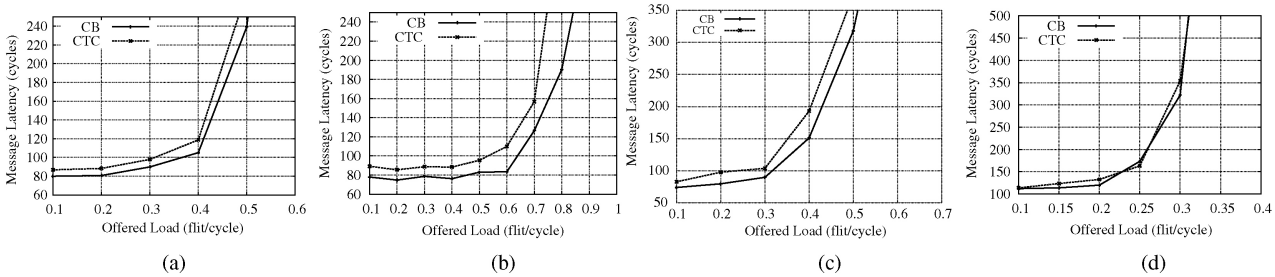


Fig. 11. Message latency as a function of the injection rate for (a) VOPD, (b) MWD, (c) DVOPD, and (d) uniform traffic patterns.

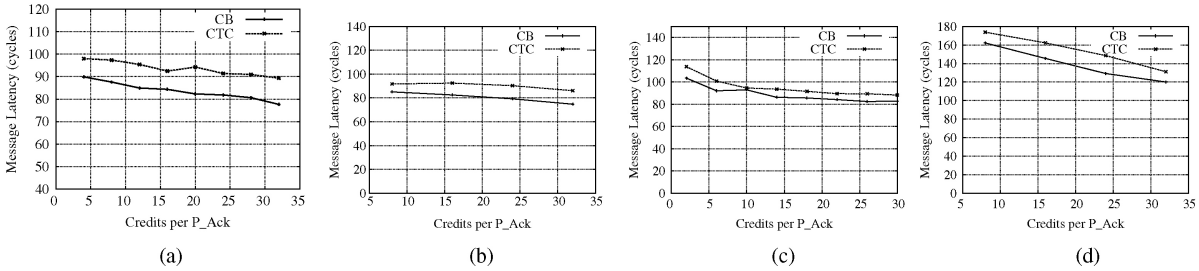


Fig. 12. Message latency as a function of the number of credits associated to a P_ACK for the case of (a) VOPD, (b) MWD, (c) DVOPD, and (d) uniform, traffic patterns.

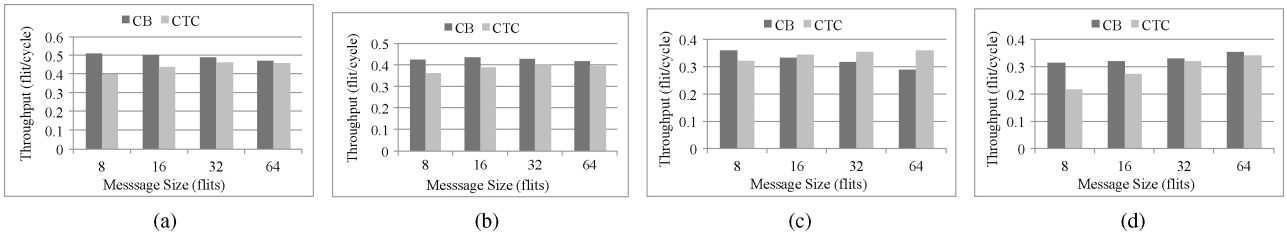


Fig. 13. NoC throughput as a function of the message size when $K = 4$ flit-credits in (a) VOPD, (b) MWD, (c) DVOPD, and (d) URT patterns.

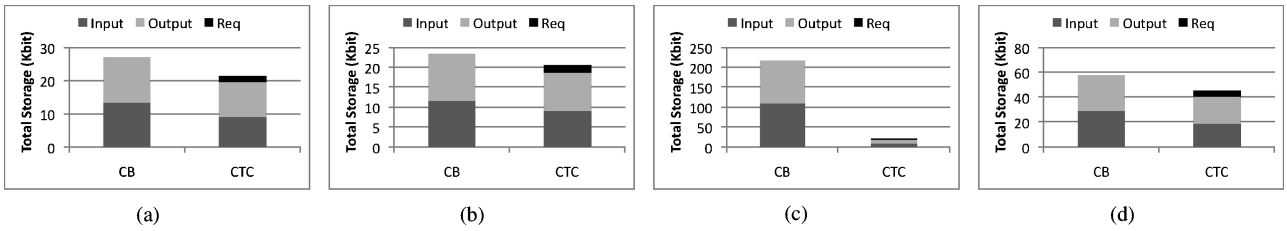


Fig. 14. Breakdown of the aggregate storage measured in bit of input, output and request queues in the NoC NIs for (a) VOPD, (b) MWD, (c) DVOPD, and (d) URT patterns.

In fact, in absence of a negotiation protocol, replicating the queues is the only way to guarantee that flits of different messages are not mixed in a single queue and that all injected flits are also absorbed from the NoC. In comparison, the storage reduction achieved by CTC can be very high because all $N - 1$ input and output queues of each node are replaced by a single input and single output data queues.

In summary, the reduction of the size of the queues that must be installed in each network interface translates directly in a reduction of the area occupation and is expected to lead also to a reduction of the overall NoC power dissipation.

A. Experimental Analysis of NI Port Contention

As discussed in Section IV-C, an NI acting as sender and receiver generates a contention on its input and output ports between the data and control packets. In this section, we study the impact of these contentions on the performance of the NoC. We first use a simple traffic pattern, where a producer NI_{i-1} sends a flow of data toward NI_i, which processes it and forward it to a sink NI_{i+1}. The three NIs and the relative cores are mapped on a three-node array (1D-Mesh) network. In this scenario, P_ACK, P_REQ, and data-flits contend for the single output channel of NI_i so that P_ACK and data packets flow toward NI_{i+1} and credits toward NI_{i-1}.

As discussed in Section IV-B, to minimize the duration of a contention a consumer should minimize the generation time of P_ACK packets. To ensure the quick generation of P_ACK packets one can choose among three main options: 1) preemptive priority for P_ACK packets; 2) VCs; or 3) separated physical networks, called multiplanes (MPs) [40].

In the first case, P_ACK packets have preemptive priority over the data packets: when the producer-consumer NI_i needs

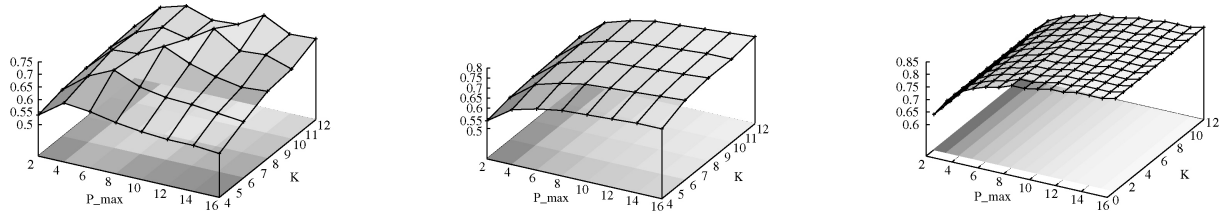


Fig. 15. Throughput as a function of $P_{\max}$ and P_ACK size, when the output flits are dispatched on (a) one link shared between P_REQ, P_ACK and data, (b) one link with two VNs, one for P_REQ and data, the other for P_ACK, and (c) two separated physical links.

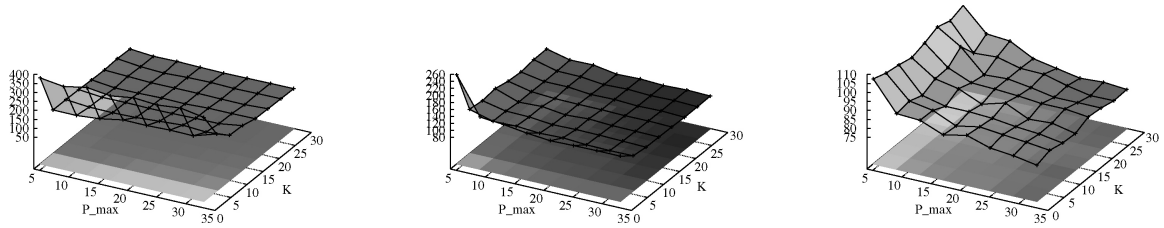


Fig. 16. Effect of $P_{\max}$ and K on the communication latency for the MWD application with (a) preemptive P_ACK, (b) VCs, and (c) separated network planes.

to generate a P_ACK, it terminates the generation of data-flits by tagging the last data flit as *flit tail*. In the following cycles, NI_i's output port can be used to inject the P_ACK packets. Fig. 15(a) shows the average throughput of data flits measured on NI_i and NI_{i+1} in this scenario. The throughput is measured in function of the maximum packet size $P_{\max}$ and the number K of credits associated to each P_ACK. As anticipated by (5), the best throughput is obtained when $K \sim P_{\max}$ so that the protocol forces NI_i to terminate a packet in coincidence with the possible generation of a P_ACK. Also, when $P_{\max}$ is smaller than K a packet is terminated even if there are available credits on the output counter. When $P_{\max}$ is larger then K, instead, delays in the delivery of P_ACK packets due to traffic congestion can force a packet to be terminated before the expected length because the NI has no more end-to-end credits. In both cases, the additional control traffic contributes to reduce the maximum achievable throughput. Indeed, $P_{\max}$ has an important role in the performance of the protocol. A value of $P_{\max}$ that is too small gives a significant loss of performance because the NIs generate many small data packets, each including a header flit that does not carry data but only routing information. In particular, when $P_{\max} = 1$ the throughput reaches the minimum value close to 0.5 flit per cycle. According to (1), to assign a large number of credits to a single P_ACK reduces the number of control credits to be injected in the network, thereby, improving the performance of the system, as confirmed by the experimental results of Fig. 15(a).

The second option is to separate P_ACK and data packets through distinct VCs: P_ACK and data packets travel along different queues but keep sharing the same physical link. Note that VCs in this case are not used to ensure the correctness of the protocol but only to improve its performance. Fig. 15(b) shows the throughput in this scenario. Here, P_ACK packets no longer need to preempt the generation of data flits because they can use separated virtual queues. However, P_ACK and data packets still share the same physical channel. This explains the improvement of the throughput as K increases: again, according to (1), by increasing the credits associated to a P_ACK we reduce the number of credit packets to be generated so that the physical channels can be used mainly for data flits. The value of $P_{\max}$, instead, leads to the same behavior as in the previous case: according to (3), by increasing the maximum packet size we reduce the number of header flits that must be generated and, therefore, we increase the throughput. Notice, however, that the traffic of Fig. 15 is a simple case, where P_ACK packets do not collide (i.e., they travel from NI_i to NI_{i-1}) with other flow of data. Hence, their delivery is never delayed by contentions in the routers. Later on in this section, we propose an analysis with a more realistic traffic pattern.

The third possible approach uses two separated physical channels, one dedicated to P_REQ and P_ACK packets and the other for data packets. Fig. 15(c) shows the throughput of this scenario. In this case, the size of the credit associated to each P_ACK does not influence the performance of the NoC. This is an expected behavior because the simulated traffic is simple and credits of any kind do not find any contention traversing the routers of the network. On the other hand, the maximum size of a packet greatly influences the performance of the system. When $P_{\max} = 1$, each data flit uses a separated header flit so that the throughput reaches 0.5 flit/cycle. The maximum throughput is achieved by increasing the $P_{\max}$ parameter. According to (3), this reduces the number of packets (and hence, headers) per message. Clearly, the throughput cannot reach the ideal unit value because the connection procedure reduces the performance of the system as indicated by (4).

Next, we analyze the effect of the NI Port Contention on the communication latency for the case of the MWD application, whose task graph is reported in Fig. 5(c), when we equip the NoC with VCs or separated MPs. Fig. 16(a) reports peer-to-peer average latency of messages in function of K and $P_{\max}$

when we use preemptive P_ACK packets. Clearly, the preemptive case has the highest latency. Here, whenever a P_ACK is to be injected into the NoC the packet (if any) that is currently being injected is terminated. After the injection of the P_ACK packets, the NI will inject a new header flit and then restart sending the data flits that remain to be sent. By increasing K, we minimize such an event. By increasing $P_{\max}$, we reduce the number of headers that must be injected, thereby, reducing the overall latency of the messages. Fig. 16(b) reports the case when VCs are used so that P_ACK packets travel along separated queues. Still, since P_ACK and data flits have to share the same physical links, to increase the value of K reduces also the average message latency. Finally, Fig. 16(c) reports the average latency, when the control and data traffic are separated on two physically-independent networks. According to (2), the value of K influences only the queue size on the network interfaces. Larger queues in the network interfaces allow the generation of more credits and a higher use of the NoC. On the other hand, the value of $P_{\max}$ greatly affects the message latency because it determines the number of packets into which a message of length M is split.

VI. Related Work

The message-dependent deadlock is a well-known problem that was first addressed by the parallel computing research community. The literature offers many works aimed at addressing this issue. The IBM SP2 supercomputer, for example, uses a bidirectional communication network that avoids message-dependent deadlock by guaranteeing sufficient queue space for each source-destination pair [41]. This solution is hardly portable to NoCs for SoCs due to the small portion of the limited on-chip power budget that can be allocated to buffering queues. The Cray T3E [42] and SGI ORIGIN2000 [43], strictly avoid message-dependent deadlock with the help of a logically-separated network for each message class. Starting from the solutions developed for parallel computing Kim *et al.* [21] propose a shared-memory architecture that avoids message-dependent deadlock by using two physically-separated networks for the request-response message types. These solutions may be difficult to scale to future multicore SoCs, where the increasing number of heterogeneous cores will rely on more complex communication patterns that will generate more elaborate message-dependency chains [15].

The Stanford DASH multiprocessor computer uses a deadlock-recovery mechanism that is based on logically–separated networks for groups of message classes and guarantees that messages belonging to certain class can always sink via deflection (or "backoff") if necessary [44]. Anjan *et al.* [18] propose to add timers into the router's output ports to detect deadlock occurrences and move the blocked packets into specialized queues to guarantee progress. Song *et al.* [19] propose a deadlock-recovery protocol motivated by the observation that in practice message-dependent deadlocks occur very infrequently even when network resources are scarce. However valid, most of these approaches are more suitable for parallel computing systems than for the SoC domain due to the additional resources that they require in terms of storage buffering and circuit logic for deadlock detection/recovery and packet reordering.

The Æthereal [23], [35], [45] and FAUST [24] NoCs use CB end-to-end flow control protocols. In order to limit the number of input and output queues required by the CB end-to-end flow control, Æthereal requires the establishing of a connection between two peers of the NoC. Connections can be defined either at design time or runtime and are managed through a centralized node. The number of input and output queues is related to the maximum number of connections that a NI can support simultaneously.

Wolkotte *et al.* [13] propose a circuit-switched architecture with a dedicated wire on each link of the NoC to carry an acknowledgement signal backward from the destination to the source of any given end-to-end connection. The acknowledgment signal is used in combination with a window counter mechanism to prevent a buffer overflow at the destination of a connection. Every source has a local window counter of size WC indicating how many data-packets the source is allowed to send to a single destination. The destination sends an acknowledgement signal every time it has read X data-packets, where $X \leq WC$. Upon the reception of this signal, the source increases its local window counter WC by X. In this approach, the number of acknowledgement signals to embed in each channel depends on the specific application. Hence, this mechanism does not support new communication patterns not considered at design time.

Another nonreconfigurable NoC is proposed by Murali *et al*. [46], who define a methodology for designing application-specific NoC taking message-dependent deadlock into account. The NoC architecture is tailored for the specific supported application. Hybrid packet/circuit switched NoCs use a request-response handshake procedure to setup virtual circuits between cores [12]. This procedure satisfies the consumption assumption and, therefore, is guaranteed to avoid message-dependent deadlock as long as all the request messages are consumed. Hansson *et al.* present a detailed comparison of *strict ordering* (separated physical networks for each message type) and end-to-end flow control approaches [15]. Their result is that the two approaches have similar costs in terms of power and area consumption but flow control-based networks offer higher flexibility as they can better support more complex traffic applications beyond the simple request-response paradigm.

A natural future extension of the CTC protocol is the implementation of quality-of-service (QoS) guarantees. In fact, the P_REQ–P_ACK handshake procedure can be used to reserve NoC links in combination with connection-aware routers so that throughput and latency guarantees are achieved. The Æthereal NoC uses a similar approach implementing routers with a time-division-multiplexing capability that can support both guaranteed and best-effort traffic packets [47], [48].

VII. Conclusion

Message-dependent deadlock is a destructive event that, even if rare [19], must be properly addressed to guarantee the correct behavior of a network. The CB end-to-end flow control protocol solves this problem by using multiple dedicated input

queues and output registers in the network interfaces. This increases the complexity of the network interface design. Further, since the number of these queues depends on the number of distinct communications that its particular core may have, the same network may present interfaces that have different microarchitectural structures.

We proposed the CTC end-to-end flow control protocol as an area-efficient solution to the message-dependent deadlock problem that is characterized by a simpler and more modular network interface architecture. CTC-supporting network interfaces use one single input data queue and one output credit counter. Hence, the overall number of queues per network interface remains equal to two, the total amount of storage is reduced, and the overall network-interface design becomes independent from the communication requirement of the particular core, thus increasing its reusability. On the other hand, any new communication between a pair of peer nodes requires the preliminary completion of a handshake procedure to initialize the output credit counter on the producer side (after the connection has been established CTC works in a way similar to the original CB flow protocol). This procedure necessarily increases the latency of a message transfer and it also reduces the network throughput for small messages.

In summary, the choice between CB versus CTC may be seen as a case of typical "performance versus area" tradeoff. From this perspective, experimental results show that for a video processing application the latency penalty remains under 10% while the savings in terms of the overall area occupation of the network interfaces reaches 35%. Therefore, CTC is an effective solution to the message-dependent deadlock problem for NoCs that are part of multicore SoCs targeting throughput-driven stream processing applications.

REFERENCES

[1] C. H. van Berkel, "Multicore for mobile phones," in *Proc. Conf. Design, Autom. Test Eur.*, Apr. 2009, pp. 20–24.

[2] P. Kollig, C. Osborne, and T. Henriksson, "Heterogeneous multicore platform for consumer multimedia applications," in *Proc. Conf. Design, Autom. Test Eur. (DATE)*, Apr. 2009, pp. 1254–1259.

[3] C.-L. Chou and R. Marculescu, "User-aware dynamic task allocation in networks-on-chip," in *Proc. Conf. Design, Autom. Test Eur. (DATE)*, Mar. 2008, pp. 1232–1237.

[4] S. Murali, M. Coenen, A. Radulescu, K. Goossens, and G. D. Micheli, "Mapping and configuration methods for multiuse-case networks on chips," in *Proc. Asia South Pacific Design Autom. Conf.*, Oct. 2006, pp. 146–151.

[5] A. Hansson, M. Coenen, and K. Goossens, "Undisrupted quality-of-service during reconfiguration of multiple applications in networks on chip," in *Proc. Design, Autom. Test Eur. (DATE)*, Apr. 2007, pp. 954–959.

[6] M. Coppola, M. D. Grammatikakis, R. Locatelli, G. Maruccia, and L. Pieralisi, *Design of Cost-Efficient Interconnect Processing Units: Spidergon STNoC*. Boca Raton, FL: CRC Press, 2008.

[7] M. Coppola, "Keynote lecture: Spidergon STNoC: The communication infrastructure for multiprocessor architectures," in *Proc. Int. Forum Appl.-Specific Multiprocessor SoC*, Jun. 2008.

[8] L. Benini and G. D. Micheli, "Networks-on-chip: A new SoC paradigm," *IEEE Comput.*, vol. 35, no. 1, pp. 70–78, Jan. 2002.

[9] W. J. Dally and B. Towles, "Route packets, not wires: On-chip interconnection networks," in *Proc. Design Autom. Conf.*, Jun. 2001, pp. 684–689.

[10] A. Hemani, A. Jantsch, S. Kumar, A. Postula, J. Öberg, M. Millberg, and D. Lindquist, "Network on chip: An architecture for billion transistor era," in *Proc. 18th IEEE NorChip Conf.*, Nov. 2000, p. 117.

[11] U. Y. Ogras and R. Marculescu, "It's a small world after all: NoC performance optimization via long-range link insertion," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 14, no. 7, pp. 693–706, Jul. 2006.

[12] M. Anders, H. Kaul, M. Hansson, R. Krishnamurthy, and S. Borkar, "A 2.9 tb/s 8 w 64-core circuit-switched network-on-chip in 45 nm CMOS," in *Proc. Eur. Solid-State Circuits Conf.*, Sep. 2008, pp. 182–185.

[13] P. Wolkotte, G. Smit, G. Rauwerda, and L. Smit, "An energy-efficient reconfigurable circuit-switched network-on-chip," in *Proc. Int. Parallel Distrib. Process. Symp.*, Apr. 2005, pp. 155–161.

[14] G. D. Micheli and L. Benini, *Networks on Chips: Technology and Tools (Systems on Silicon)*. San Francisco, CA: Morgan Kaufmann, 2006.

[15] A. Hansson, K. Goossens, and A. Rădulescu, "Avoiding message-dependent deadlock in network-based systems on chip," in *Proc. Very Large Scale Integr. VLSI Design*, vol. 2007. May 2007, pp. 1–10.

[16] W. J. Dally and B. Towles, *Principles and Practices of Interconnection Networks*. San Mateo, CA: Morgan Kaufmann, 2004.

[17] J. Duato, S. Yalamanchili, and L. Ni, "Message switching layer," in *Interconnection Networks: An Engineering Approach*. San Mateo, CA: Morgan Kaufmann, 2003, ch. 2, pp. 43–80.

[18] K. V. Anjan and T. M. Pinkston, "DISHA: A deadlock recovery scheme for fully adaptive routing," in *Proc. Int. Symp. Parallel Process. (IPPS)*, Apr. 1995, pp. 537–543.

[19] Y. H. Song and T. M. Pinkston, "A progressive approach to handling message-dependent deadlock in parallel computer systems," *IEEE Trans. Parallel Distrib. Sys.*, vol. 14, no. 3, pp. 259–275, Mar. 2003.

[20] H. D. Kubiatowicz, "Integrated shared-memory and message-passing communication in the alewife multiprocessor," Ph.D. dissertation, Dept. Elect. Eng. and Comput. Sci., Masachusetts Inst. Technol., Boston, 1997.

[21] J. Kim, J. Balfour, and W. Dally, "Flattened butterfly topology for on-chip networks," in *Proc. IEEE Micro.*, Dec. 2007, pp. 172–182.

[22] S. Murali and G. D. Micheli, "An application-specific design methodology for STbus crossbar generation," in *Proc. Conf. Design, Autom. Test Eur.*, Apr. 2005, pp. 1176–1181.

[23] J. Dielissen, A. Rădulescu, K. Goossens, and E. Rijpkema, "Concepts and implementation of the Philips network-on-chip," in *Proc. IP-Based SoC Design*, Nov. 2003.

[24] Y. Durand, C. Bernard, and D. Lattard, "FAUST: On-chip distributed architecture for a 4G baseband modem SoC," in *Proc. Design Reuse IP-SoC Conf.*, Nov. 2005, pp. 51–55.

[25] A. S. Tanenbaum, "The network layer," in *Computer Networks*, 2nd ed. Englewood Cliffs, NJ: Prentice-Hall, 1988, ch. 5, pp. 343–423.

[26] L. Peh and W. J. Dally, "A delay model for router microarchitectures," *IEEE Micro.*, vol. 21, no. 1, pp. 26–34, Jan. 2001.

[27] J. L. Hennessy and D. A. Patterson, "Multiprocessors and thread-level parallelism," in *Computer Architecture: A Quantitative Approach*. San Mateo, CA: Morgan Kaufmann, 2006, ch. 4, pp. 196–264.

[28] R. L. Collins and L. P. Carloni, "Flexible filters: Load balancing through backpressure for stream programs," in *Proc. Embedded Software (EMSOFT)*. Oct. 2009, pp. 205–214.

[29] T. M. Pinkston and J. Shin, "Trends toward on-chip networked microsystems," *Int. J. High Performance Comput. Netw.*, vol. 3, no. 1, pp. 3–18, 2001.

[30] M. Chaudhuri and M. Heinrich, "Exploring virtual network selection algorithms in DSM cache coherence protocols," *IEEE Trans. Parallel Distrib. Sys.*, vol. 15, no. 8, pp. 699–712, Aug. 2004.

[31] E. van der Tol and E. Jaspers, "Mapping of MPEG-4 decoding on a flexible architecture platform," in *Proc. Soc. Photo-Optic. Instrum. Eng. Conf.*, Jan. 2002, pp. 1–13.

[32] A. Pullini, F. Angiolini, P. Meloni, D. Atienza, S. Murali, L. Raffo, G. De Micheli, and L. Benini, "NoC design and implementation in 65 nm technology," in *Proc. Int. Symp. Netw.-Chips (NOCS)*, May 2007, pp. 273–282.

[33] D. Bertozzi, A. Jalabert, S. Murali, R. Tamhankar, S. Stergiou, L. Benini, and G. De Micheli, "NoC synthesis flow for customized domain specific multiprocessor systems-on-chip," *IEEE Trans. Parallel Distrib. Syst.*, vol. 16, no. 2, pp. 113–129, Feb. 2005.

[34] M. Bekooij, "Dataflow analysis for real-time embedded multiprocessor system design," in *Dataflow Analysis for Real-Time Embedded Multiprocessor System Design*. San Mateo, CA: Kluwer, 2005, ch. 4, pp. 81–108.

[35] O. P. Gangwal, A. Rădulescu, K. Goossens, S. González Pestana, and E. Rijpkema, "Building predictable systems on chip: An analysis of guaranteed communication in the Æthereal network on chip," in *Dynamic and Robust Streaming in and Between Connected Consumer-Electronics Devices* (Philips Research Book Series), vol. 3, P. van der

Stok, Ed. Berlin, Germany: Springer-Verlag, 2005, pp. 1–36.
[36] E. Carara, N. Calazans, and F. Moraes, "Router architecture for high-performance NoCs," in *Proc. Conf. Integr. Circuits Syst. Design (SBCCI)*, Sep. 2007, pp. 111–116.
[37] N. Concer, M. Petracca, and L. P. Carloni, "Distributed flit-buffer flow control for networks-on-chip," in *Proc. Int. Conf. Hardware/Software Codesign Syst. Synth. (CODES+ISSS)*, Oct. 2008, pp. 215–220.
[38] L. Bononi and N. Concer, "Simulation and analysis of network on chip architectures: Ring, Spidergon, and 2-D mesh," in *Proc. Design, Autom. Test Eur. (DATE)*, 2006, pp. 154–159.
[39] L. Bononi, N. Concer, M. Grammatikakis, M. Coppola, and R. Locatelli, "NoC topologies exploration based on mapping and simulation models," in *Proc. Euromicro Conf. Digital Syst. Design Architect., Methods Tools (DSD)*, Aug. 2007, pp. 543–546.
[40] Y. J. Yoon, N. Concer, M. Petracca, and L. P. Carloni, "Virtual channels versus multiple physical networks: A comparative analysis," in *Proc. Design Autom. Conf.*, Jun. 2010.
[41] C. B. Stunkel, D. G. Shea, B. Aball, M. G. Atkins, C. A. Bender, D. G. Grice, P. Hochschild, D. J. Joseph, B. J. Nathanson, R. A. Swetz, R. F. Stucke, M. Tsao, and P. R. Varker, "The SP2 high-performance switch," *IBM Syst. J.*, vol. 34, no. 2, pp. 185–204, 1995.
[42] S. Scott and G. Thorson., "The Cray T3E network: Adaptive routing in a high performance 3-D torus," in *Proc. Hot Chips*, Nov. 1996, pp. 158–163.
[43] J. Laudon and D. Lenoski, "The SGI origin: A ccNUMA highly scalable server," *SIGARCH Comput. Archit. News*, vol. 25, no. 2, pp. 241–251, 1997.
[44] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy, "The directory-based cache coherence protocol for the DASH multiprocessor," in *Proc. 17th Int. Symp. Comput. Architecture*, Apr. 1997, pp. 148–159.
[45] B. Gebremichael, F. Vaandrager, M. Zhang, K. Goossens, E. Rijpkema, and A. Radulescu, "Deadlock prevention in the Æthereal protocol," in *Proc. Correct Hardware Design Verificat. Methods (CHARME) Conf.*, Oct. 2005, pp. 345–348.
[46] S. Murali, P. Meloni, F. Angiolini, and D. Atienza, "Designing message-dependent deadlock free networks on chips for application-specific systems on chips," in *Proc. Very Large Scale Integr. VLSI-Syst.-Chip (SoC)*, Nov. 2006, pp. 158–163.
[47] E. Rijpkema, K. G. W. Goossens, A. Radulescu, J. Dielissen, J. van Meerbergen, P. Wielage, and E. Waterlander, "Trade-offs in the design of a router with both guaranteed and best-effort services for networks on chip," in *Proc. Conf. Design, Autom. Test Eur. (DATE)*, vol. 150, no. 5. Sep. 2003, pp. 294–302.
[48] A. Jantsch and H. Tenhunen, "Guaranteeing the quality of services in networks on chip," in *Networks-on-Chip*. Hingham, MA: Kluwer, 2003, ch. 4.

Nicola Concer received the Laurea (summa cum laude) and Ph.D. degrees in computer science from the University of Bologna, Bologna, Italy, in 2005 and 2009, respectively.

Since 2009, he has been a Post-Doctoral Researcher with the Department of Computer Science, Columbia University, New York, NY, under the supervision of Dr. L. P. Carloni. His current research interests include the design of power-efficient interconnection systems for heterogeneous multicore system-on-chip architectures.

Luciano Bononi received the Laurea (summa cum laude) and Ph.D. degrees in computer science from the University of Bologna, Bologna, Italy, in 1997 and 2002, respectively.

In 2000, he was a Visiting Researcher with the Department of Electrical Engineering, University of California, Los Angeles. In 2001, he joined University of Bologna as a Post-Doctoral Researcher, and since 2002, he has been an Assistant Professor with the Department of Computer Science at the same university. He is the author of more than 60 conference and journal publications and five book chapters on topics related to mobile and wireless network protocols, standards and architectures.

Dr. Bononi serves as an Associate Editor of six international journals. He has served as the Chair and the Technical Program Committee Member of more than 10 and 100 international conferences and workshops, respectively.

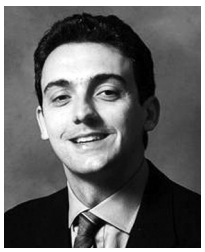
Michael Soulié received the Engineer degree from the Institut Polytechnique de Grenoble, École Nationale Supérieure d'Électronique et de Radioélectricité de Grenoble, Grenoble, France, in 2006.

Since 2006, he has been a Member of the Spidergon ST Network-on-Chip (SSTNoC) Team of Antonio-Marcello Coppola, Department of Advanced Search Technology, STMicroelectronics, Grenoble, France. He has been working on network-on-chip for the past four years under the supervision of Dr. R. Locatelli, focusing particularly in the architecture of the Spidergon STNoC, an STMicrolectronics proprietary efficient on-chip interconnect technology. His current research interests include high-level modeling of the SSTNoC and intellectual property protocol interoperability.

Riccardo Locatelli received the Laurea degree (summa cum laude) in electronic engineering, and the Ph.D. degree in information engineering from the University of Pisa, Pisa, Italy, in 2000 and 2004, respectively.

In 1999, he was a Research Intern with the Microelectronics Section of the European Space Agency, the Netherlands, and a Visiting Researcher with the Advanced Search Technology Grenoble Laboratory of STMicroelectronics, Grenoble, France, in 2003. At Pisa University, he worked on definition and prototyping of video architectures with emphasis on low-power techniques and system communication. He was a Digital Design Engineer with CPR-TEAM, a microelectronic design house in Pisa, where he worked on advanced signal processing schemes for VDSL applications. Since 2004, he has been a Technical Leader and an Architecture and Design Team Leader with STMicroelectronics, Grenoble, France, and the Spidergon ST Network-on-Chip (SSTNoC) Interconnect Processing Unit (IPU), where he introduced novel concepts beyond networks-on-chip (NoC). He has published about 30 papers in international journals and conference proceedings and has filed nine international patents on NoC. He is the co-author of a book on Spidergon STNoC technology (Boca Raton, FL: CRC Press, 2008).

Dr. Locatelli is a member of the technical program committee the NoC Symposium and the Design, Automation and Test in Europe, a Reviewer of the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN journal and of several International Conferences.



Luca P. Carloni (SM'09) received the Laurea degree (summa cum laude) in electrical engineering from the University of Bologna, Bologna, Italy, in 1995, and the M.S. and Ph.D. degrees in electrical engineering and computer sciences from the University of California, Berkeley, in 1997 and 2004, respectively.

He is currently an Associate Professor with the Department of Computer Science, Columbia University, New York, NY. He is the author of over 70 publications and holds one patent. His current research interests include the area of design tools and methodologies for integrated circuits and systems, distributed embedded systems design, and design of high-performance computer systems.

Dr. Carloni received the Faculty Early Career Development (CAREER) Award from the National Science Foundation in 2006, and was selected as an Alfred P. Sloan Research Fellow in 2008. He is the recipient of the 2002 Demetri Angelakos Memorial Achievement Award "in recognition of altruistic attitude toward fellow graduate students." In 2002, one of his papers was selected for "The Best of the International Conference on Computer Aided Design (ICCAD)," a collection of the best IEEE ICCAD papers of the past 20 years. He is a Senior Member of the Association for Computing Machinery.