CTC: an End-To-End Flow Control Protocol for Multi-Core Systems-on-Chip

Nicola Concer, Luciano Bononi
Dipartimento di Scienze dell'Informazione
Università di Bologna
{*concer,bononi*}*@cs.unibo.it*

Michael Soulié, Riccardo Locatelli
ST Microelectronics
Grenoble, France
{*michael.soulie,riccardo.locatelli*}*@st.com*

Luca P. Carloni
Computer Science Department
Columbia University in the City of New York
luca@cs.columbia.edu

Abstract

We propose Connection then Credits *(CTC) as a new end-to-end flow control protocol to handle message-dependent deadlocks in networks-on-chip (NoC) for multi-core systems-on-chip. CTC is based on the classic end-to-end credit-based flow control protocol but differs from it because it uses a network interface micro-architecture 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.*

1 Introduction

Future generations of systems-on-chip (SoCs) will consist of heterogeneous multi-core 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. These sub-systems, which are also composed of several cores, will provide higher efficiency in supporting important classes of applications across multiple use-case scenarios [14, 19].

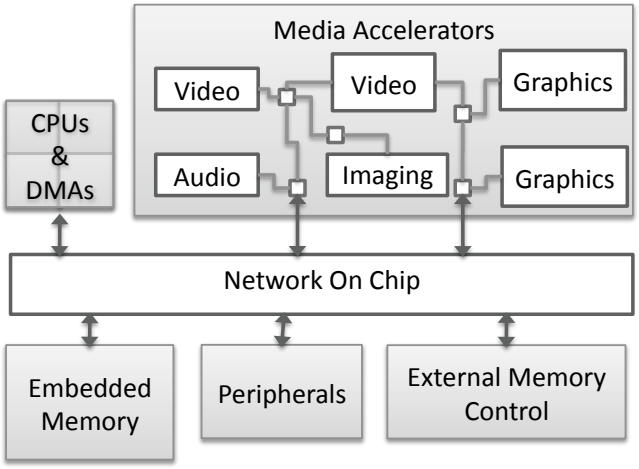

Figure 1. NoC-based System on Chip and the media-accelerator sub-system [6].

Heterogeneity is the combined result of hardware specialization, reuse of Intellectual Property (IP) modules, and the application of derivative design methodologies [7]. 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 [7]. 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-

978-1-4244-4143-3/09/$25.00 ©2009 IEEE

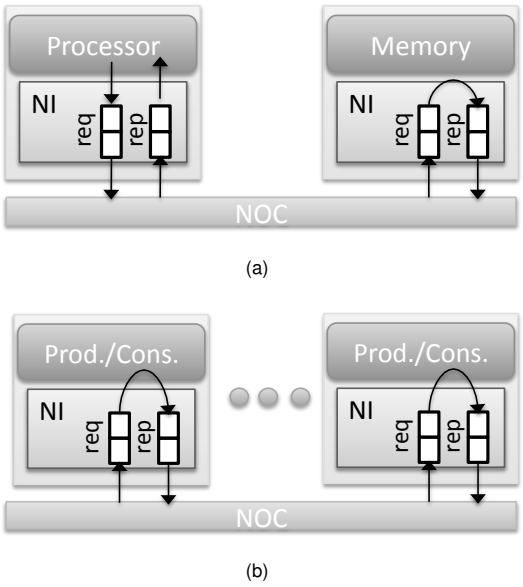

Figure 2. NIs connecting cores to the NoC and the possible message dependencies in the (a) shared memory and (b) message passing communication paradigms.

on-chip (NoC) paradigm [2,8,16]. These NoCs will be also heterogeneous as illustrated in Fig. 1, where a top-level network connects the main components of the chip, while other sub-networks 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 applications such as video stream processing and other multimedia applications [15].

In this paper we focus on the design of the sub-network interconnecting the cores of a message-passing sub-system and we propose an end-to-end flow control that optimizes the flexibility and reusability of the cores through the definition of a unified network interface (NI) design.

Network interfaces are a crucial component 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 storage capabilities. As shown in Fig. 2 input and output queues are used to temporarily store the incoming and outgoing messages. While messages are the units of transfer between the network clients (processors and memories), in the network interface a single message is typically broken down into a sequence of smaller packets for routing purposes; packets may be further segmented in flow control digits (flit) for more efficient allocation of network resources such as link bandwidths and queue capacities [9, 11].

The correct operations of a network requires to efficiently handle deadlock situations which 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 [1, 9]. Most of these protocols assume the *consumption assumption* where the packets of a message traversing the network are always consumed by the destination core once they reach its corresponding network interface [22]. However, as depicted in Fig. 2 and discussed in detail in Section 2, deadlock may be caused also by dependencies that are *external* to the network, i.e. dependencies that are *internal* to a core. In fact, in real SoC systems and multiprocessor systems a core typically generates new messages in response to the reception of a previous message. These dependencies between messages can generate a different type of deadlock that is commonly referred as *message-dependent* (or *protocol*) deadlock [15, 18, 22]. Message-dependent deadlock occurs at a level of abstraction that is higher than the routing-level deadlock, which can be addressed by deadlock-free routing algorithms such as dimension-order routing [9, 11]. [1]

Related Work. Various solutions for message-dependent deadlock have been proposed in the literature. Dielissen *et al.* solve this problem by guaranteeing sufficient storage space for each possible pair of communicating elements [10]. Anjan *et al.*, instead, add timers into the router's output ports to detect deadlock occurrences and move the blocked packets into specialized queues to guarantee progress [1]. Song *et al.* propose a deadlock-recovery protocol motivated by the observation that in practice message-dependent deadlocks occur very infrequently even when network resources are scarce [22]. These three approaches, however, are meant for parallel computing systems and are not expected to scale well to SoC design.

The message-dependent deadlock problem in NoC for shared-memory architectures has been addressed by introducing two physically-separated networks for the two message types (load and store requests) [20] or two logically-separated network (virtual networks) [7]. These solutions may be difficult to scale to future multicore SoCs where the increasing number of heterogeneous cores and message types is likely to grow, thus leading to more complex dependencies among packets.

The ÆTHEREAL [13] and FAUST [12] NoCs use *credit-based (CB) end-to-end flow control protocols*. Similar to the credit-based flow control mechanisms that operate at the

[1] 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.

link level between a pair of interconnected routers [9, 11], a CB end-to-end flow control protocol uses credits to inform a sender NI about the current storage capacity of the queue in the receiving NI. As discussed in more detail in Section 3, the sender NI keeps track of this capacity with a credit counter that is initialized with a value equal to the size of the corresponding queue and is dynamically updated to track the number of available packet slots in the queue. Hence, the sender continuously transmits only a subset of the message packets that is guaranteed to eventually arrive inside the NI, thus avoiding a message-dependent deadlock. Notice that for a given SoC a core that may send messages to N different cores needs N credit counters while if it can receive messages from M different cores it needs M different queues.

Contributions. We build on the CB approach to develop *Connection then Credits* (CTC), an end-to-end flow control protocol that allow us to handle the message-dependent deadlock while simplifying the design of the network interface, which is based on the same micro-architecture regardless of the number of communications that its core may require. This micro-architecture uses a single credit counter together with an output queue for sending all the possible outgoing messages and a single pair of data-request queues that is shared across all possible incoming messages. On the other hand, as explained in Section 4, CTC requires the completion of a handshake procedure between any pair of cores that want to communicate before the actual message transfer starts. This procedure is used to initialize the credit counter in the sender NI based on the current available space in the data queue of the receiver NI. While this necessarily adds a latency overhead to the transfer of the message, the penalty in performance is limited when large messages need to be transferred as it is shown by the simulation results that we report in Section 5.

2 Message-Dependent Deadlock

There are two main communication paradigms for exchanging data among the processing cores of a system-on-chip 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 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 correspond to a line in the processor's local cache, and it replies by generating a short ACK message to confirm their correct delivery. Shared memory is the most used paradigm in current multi-core 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 [17], thus potentially reducing the communication latency of each data transfer. This paradigm is particularly suited for data-flow and stream processing applications that consist of chains of processing cores such as the video processing pipeline [15].

The correct implementation of shared memory and message passing paradigms in a system-on-chip requires an underlying NoC with communication protocols that guarantee the correct transfer of each message and, particularly, the absence of deadlocks. As discussed in the Introduction, even if the NoC relies on deadlock-free routing algorithms, message-dependent deadlock may arise due 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 causality 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* [21]. Indeed, the causality 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 [15].

Example. Fig. 3 shows a simple example of a message-dependent deadlock that may occur due to the dependence between the messages that are received by (sent from) a memory core in a shared memory environment. The network interface NI_0 receives packets for a memory load (or store) request message addressed to $Memory_A$ and in reply sends packets with a response message that includes the requested data (or the acknowledgment of a store operation). Since the input and output queues of NI_0 have necessarily limited storage capacity, a long sequence of requests may cause a back-pressure effect into the NoC. For instance, the packets of a series of load request messages $Load_i$ from $Processor_A$ may not be fully stored within NI_0 and, instead, may have to wait for several clock cycles in the East queue of $Router_0$. Then, let's assume

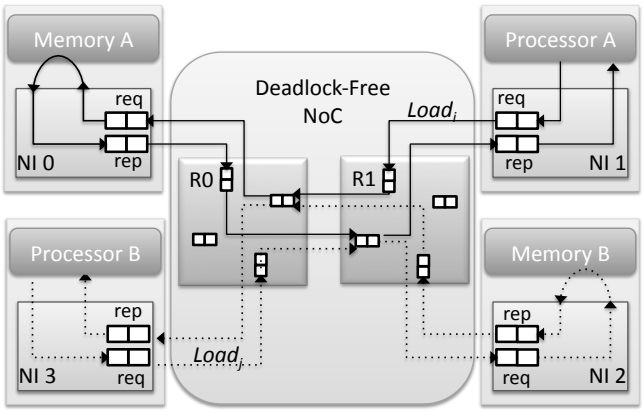

Figure 3. Message-dependent deadlock in a shared-memory request-response paradigm.

that $Processor_B$ sends a series of load request messages $Load_j$ to $Memory_B$. Even if $Memory_B$ can immediately serve a first subset of these requests, the packets of the corresponding response messages will not be able to reach $Processor_B$ because they will be blocked as they attempt to access the East Queue of $Router_0$. On the other hand, when $Memory_A$ will be finally able to serve the request messages $Load_i$, the packets of its response messages will not be able to reach $Processor_A$ because they will be blocked as they attempt to access the West Queue of $Router_1$, which are occupied by some of the packets of the load request messages $Load_j$. In summary, even if the NoC uses a deadlock-free routing algorithm, the dependencies across the messages inside the memory cores cause a circular dependency involving NI_0, $Router_0$, $Router_1$, and NI_1 which leads to 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 [22].

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 [7, 22]: the input and output queue of each router and each NI in the network is replicated and assigned to a single specific message class (e.g. two classes in case of memory request and response messages). This solution "cuts" the causality dependency between messages in the network at the cost of a higher buffer requirement and more complex router and NI design.

Stream processing applications implemented with a pipeline of processing cores, where each core produces data for the next consumer core, lead to a dependency chain of message requests $request_1 \prec \cdots \prec request_n$ where n is the number of cores in the pipeline. For example, Fig. 4 shows the task graphs of the Video Object Plane Decoder (VOPD) and Multi-Window Display (MWD) applications where the nodes represent the tasks while the arcs represent the communications between the tasks and the relative bandwidth requirements [3]. The task graphs of these applications, which are examples of stream processing applications, are similar and present a fairly-linear pipeline structure with a sequence of twelve stages, each stage corresponding to a task. Hence, an optimally-concurrent SoC implementation where each task is mapped to a distinct processing core and where each inter-task communication is implemented by end-to-end message passing would lead to as many message types as the number of arcs. Multi-core SoCs for embedded products simultaneously support an increasing number of such streaming applications. This translates into the presence of complex communication patterns among the cores, which simultaneously run multiple threads of computation to implement the multiple tasks of the various applications. The implementation of the communication requirements among these cores with a NoC requires new solutions for the message-dependent protocol. In fact, a solution based on virtual networks does not scale as the number of distinct message types that travel on the network continues to grow. Furthermore, the length of the dependency chains is difficult to predict because it depends on the given application.

3 Credit Based (CB) Protocol

A different approach to the solution of the message-dependent deadlock is based on the use of an *end-to-end* flow control protocol that guarantees that a sender NI does not ever inject more flits in the network than the corresponding receiver NI can eject. The Credit Based (CB) end-to-end flow control protocol is a simple implementation of this idea that has been used in previous works [12, 13].

With a CB protocol, the sender NI maintains a detailed knowledge of the number of queue slots that the receiver NI has still available through the exchange of end-to-end transmission credits. A credit can be associated to either a packet or to a packet flit depending on the desired level of granularity [2]. What is important is the guarantee that no fragment of a message can remain blocked in the network due to the lack of space in the NI input queue, 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 eject these flits. Dually, the receiver NI must send a credit back to the

[2] In this paper we use the granularity of the flit and we refer to them as *flit-credits*.

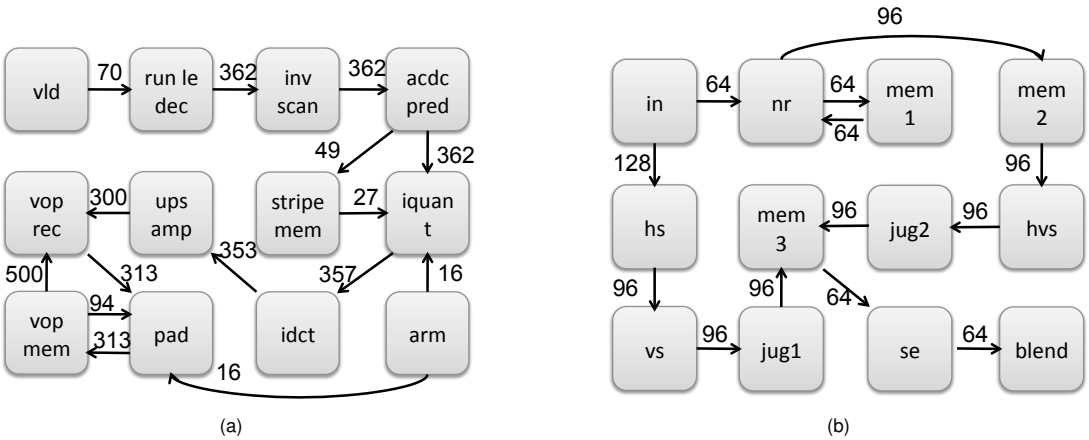

Figure 4. The MP (a) Video Object Plane Decoder (VOPD) and (b) Multi-Window Display (MWD) task graphs.

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 separated credit counter. Differently from credit-based flow control mechanisms that operate at the link level between a pair of interconnected routers [9, 11], 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 NI's input port. Fig. 5(a) shows the simplest way to address this issue. Each NI is provided with multiple and independent input and output queues and credit counters: one input queue for each possible sender NI and one output queue and credit counter for each addressed NI.

A generic NI_d, upon the reception of a flit from NI_s, saves the incoming data into Q_s, the input queue associated to the source NI_s. When NI_d reads a flit from Q_s, it generates an end-to-end flit-credit to send back to NI_s. In turn, NI_s updates the credit counter C_d associated to the destination NI_d. Multiple credits can be combined into one single end-to-end credit message for better efficiency. The amount of flit-credits *K* associated to a single credit-message is a fixed parameter of the system. The size Q of each input queue depends on the value of *K* because the NI must be capable of storing K incoming flits for each flit-credit that it has generated. In particular considering the set of peers $P_c = \{n_i \dots n_j\}$ that can possibly communicate with the consumer n_c, n_c's input queues should be sized as:

$$Q_{n_c} = K + Max\{RTT(n_i, n_c) \mid n_i \in P_c\} \qquad (1)$$

where *RTT* is the zero-load round-trip time function that depends on the given NoC implementation: specifically it depends on the distance in hops between the NIs, the latency of the routers, and, in case of channel pipelining [5], on the number of stages of each pipelined channel.

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. The selection of the input and output queues is made on a packet basis to avoid the delivering/reception of flits of different packets, e.g. according to a round-robin policy.

The CB end-to-end flow control protocol differs from the virtual network approach for two main of reasons. First, in VN all the queues, including those in the routers, must be replicated while in the CB protocol only the queues of the NI must be replicated. Second, in VN the number of queues per channel depends on the length of the application message-dependency chain while in the CB protocol this number varies for each given consumer core depending on the number of producer cores that may address it.

4 Connection Then Credits (CTC) Protocol

Adding a dedicated input and output queue for each possible source/destination of messages, as required by the CB flow control protocol, forces engineers to design a specific network interface for each node of the network. This is the case particularly for multi-use-case SoCs where the interaction between cores is driven by the application run by the user [14,19]. As an alternative to CB, we present the *"Connection Then Credits" (CTC)* flow control protocol. CTC rationalizes and simplifies the design of NIs while guaranteeing the absence of message-dependent deadlock.

CTC regulates the exchange of messages between two peer NIs by introducing a handshake procedure called *Connection*. A CTC-message is a fixed amount of data to be exchanged between the two NIs. As shown in Fig. 5(b) a

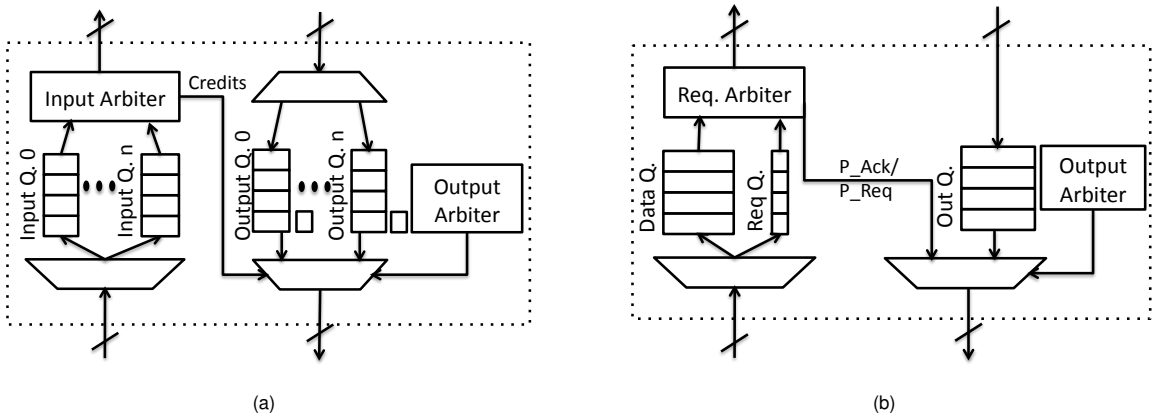

Figure 5. Network Interface implementations: (a) credit based and (b) CTC.

CTC NI is composed of only *two* input queues and *one* single output queue independently from the number cores that can require a connection with this NI. The *data-queue* is used for storing incoming data flits while the *request-queue* instead is used for the incoming transactions requests.

When a producer NI_s needs to initiate a connection towards a consumer peer NI_d, it first sends a request packet[3] called P_REQ (packet-request) to NI_d to signal its request to communicate. The P_REQ packet also indicates the total size of the message to be delivered and some additional information that can be used by the NI (i.e. for priority decisions). Upon the reception of a P_REQ, NI_d stores the request in the request-queue together with the other requests previously received and not yet processed. When the core associated to NI_d is available for processing a new request (i.e., the data queue has enough free space to accept a new message) it generates an acknowledge packet called P_ACK that is forwarded to the source of the given request. The P_ACK packet is similar to the end-to-end credit packet in the CB flow control. The difference is that the first P_ACK sent by NI_d actually *initializes* the output credit counter of NI_s so that it can generate and send a specific amount of data. Upon the reception of credits the producer first generates a *header* flit used to open a path along the routers of the NoC, then it forwards the data flits and decreases the CTC-counter by one unit for each data-flit that has been sent.

Fig. 6 shows an example of the CTC protocol operations: at first NI_0 and NI_2, address NI_1 with two P_REQ messages indicating the size of the transaction they want to initiate (respectively 100 and 80 flits). In Fig. 6(b) NI_1 selects NI_0 to initiate the connections while it stores the other request in the request-queue. Then NI_1 generates a chain of consecutive P_ACK packets so that NI_0's credit counter is initialized with the maximum amount of flit-credits that NI_1 can offer.

As for the CB case, a single P_ACK conveys *K* flit-credits per single message but in the former counters are initialized at start up time whereas in CTC the consumer NI initializes the counter of the selected producer peer. Considering the example in Fig.6 if the size of the data input queue were *Q=10* and *K=3*, NI_1 would generate 3 consecutive P_ACK messages to initialize NI_0's credit counter to the maximum storage it can offer. As *Q* is not a multiple of *K* one slot is not assigned at the connection start-up time but it will be assigned as soon as NI_1 starts receiving flits and forwarding them to the connected core. Finally, Fig. 6 (c) shows the data-packet generated by NI_0 reaching the data-queue of NI_1.

Independently from the consumption rate of the core, in both considered end-to-end flow controls each time a consumer frees *K* slots in the data input-queue, it generates a new P_ACK message until the sent credits are sufficient to transfer all the flits of the requested transition (whose size was specified in the P_REQ message). Considering the example in Fig. 6 with $K = 3$, NI_1 generates a new P_ACK for each *K* flits that are consumed. Hence the 100 flits of message *M* require a total of $\lceil M/K \rceil = 34$ P_ACK messages. As the message's length *M* is not a multiple of *K*, the consumer NI assigns to the connection a bigger amount of memory slots than *M*. Nevertheless at most *K-1* slots of *Q* can be reserved but not actually used. These additional slots are marked as *dirty* and freed when the connection is terminated.

Finally upon the sending/reception of the last flit of *M*, the producer and the consumer terminate the connection. Note that there is no need of specific flags or instructions as both producer and consumer nodes know the exact size of the transfer to process.

[3]Note that P_REQ and P_ACK are actually *messages* composed by one single packet. When referring to these two messages we use the two words without distinction.

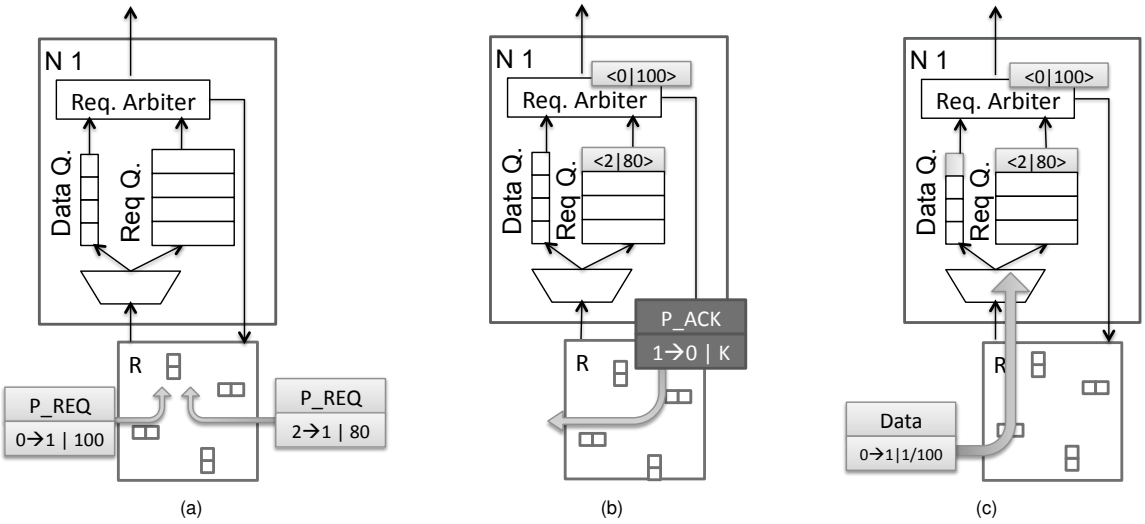

Figure 6. The CTC transaction-definition procedure: (a) the core interface NI_1 receives two P_REQ requests from NI_0 and NI_2; (b) NI_1 selects the request from NI_0 generating the relative P_ACK and stores the one from NI_2 in the request-queue; (c) NI_1 receives the data-packets from NI_0.

To avoid throughput degradation, the data-input queue should be sized accordingly to Equation 1 as function of the number of credits per P_ACK and the maximum round trip time between the consumer NI and the producer addressing it. Using a fixed value of *K* rather than a dynamic one, reduces the number of control packets traveling on the network. Moreover the parameter *K* influences both the number of CTC-control messages traveling along the network and the size of the data input queue on the NIs. Hence it has a clear impact on the design and performance of the network. According to Equation 1 a small values of K allows to reduce the size of the input queue saving area and power consumption. On the other hand, it implies a higher number of P_ACK packets flowing through the network to update the credits on the producer side. To avoid loss of performance and reduce contentions, a producer node that runs out of credits terminates the injection of the packet by tagging the flit using the last credit as *packet tail*. In this way the path along the network is made available to other possibly blocked packets. Note that the CTC connection is not terminated until the delivery of all the flits of the message is completed.

To guarantee the consumption assumption of the P_REQ messages, the request-queue must be sized accordingly to the maximum number of requests that a NI can receive. By limiting each CTC producer to have one single outstanding P_REQ at time the request queue can be sized according to the number of possible sources of data so that all incoming P_REQ can be stored in the NI and removed from the NoC.

CTC defines three message dependencies that are addressed in the following way:

P_REQ→ P_ACK: the request-queue is sized accordingly to the number of possible producer-peers addressing the given NI. CTC limits each node to have at most one outstanding P_REQ at time. Hence the consumption of all injected P_REQ is guaranteed.

P_ACK→data: P_ACK packets are always consumed by a network interface that updates the output credit counter and then deletes them.

data→ P_ACK: the credit mechanism ensures that no more data-flits than those allowed by the output credit counter can ever been injected. Hence all data flits injected in the NoC are eventually consumed by their addressed NI.

Thanks to the CTC protocol design, these three dependencies are independent from the applications that is run by the cores and hence the system can be considered protocol-deadlock free.

5 Analysis and Simulation

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 injected by various types of cores. We used the simulator to compare the two end-to-end protocols on the Spidergon NoC [7] for the case of the VOPD and MWD applications whose task graphs are shown in Fig. 4. We as-

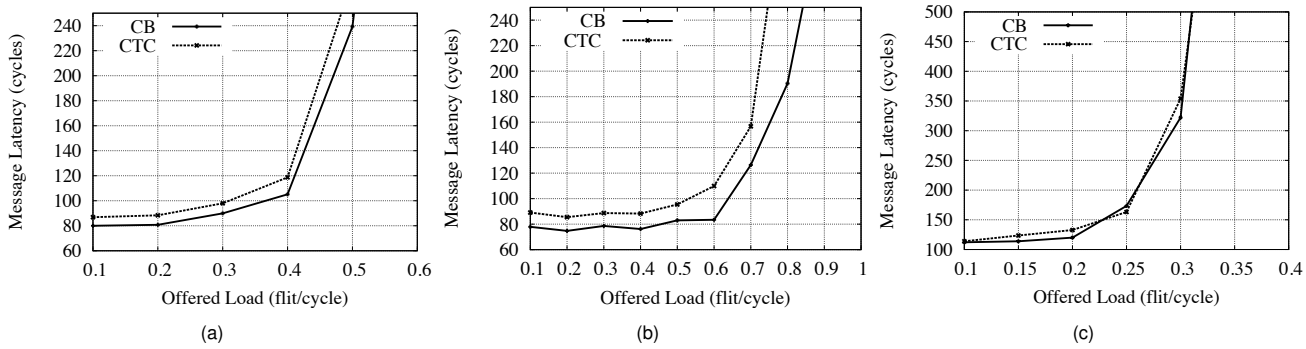

Figure 7. Message latency as function of the injection rate for (a) VOPD, (b) MWD and (c) Uniform traffic patters.

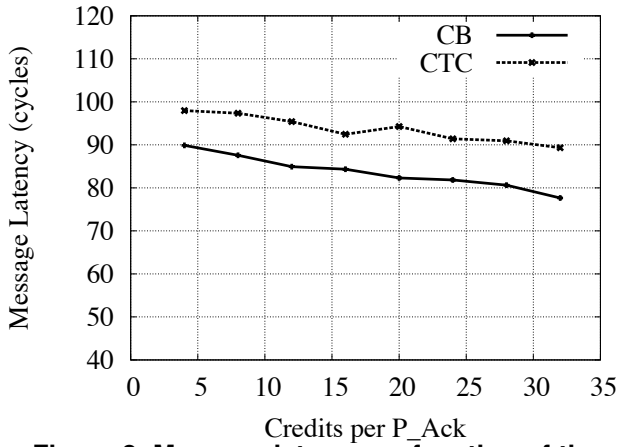

Figure 8. Message latency as function of the number of credits associated to a P_ACK for the case of the VOPD traffic pattern.

signed each task to a different core following the heuristic presented in [4]. We also considered the *random uniform traffic pattern* where each node may communicate with any other node in the network [9]

In the CB-based NoC the NI of each core has a number of input queues equal to the number of incoming streams (see Fig. 4). For both the CTC-based and the CB-based NoC, the size of each data input queue is set uniformly based on Equation 1 and no virtual channels are used.

Fig. 7 shows the average end-to-end message latency as function of the average offered load when 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.

Fig. 8 shows the average peer-to-peer latency as function of the number of credits K per P_ACK packet when the offered load is lower than the saturation threshold for the case of the VOPD application. Clearly, by increasing the value of K the performance of the system also improves: PEs 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. 9 reports the throughput comparison as function of the message size *M*. As expected, 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. Therefore, CTC represents a valid proposition for message-passing applications such as video stream processing that present large inter-core message transfers.

Finally, we analyze the amount of storage used by the two alternative flow control protocols. As discussed in Section 3, 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 Equation 1. 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 over-estimated without a major loss of area because the request-queue has a negligible size compared to the data-queue. Fig. 10 shows the breakdown of the aggregate number of data-queues used in the network interfaces for the two approaches to support the given applications. In VOPD and MWD only the interfaces associated to nodes with incident (outgoing) arrows actually require input (output) queues. Considering VOPD the CTC-based NoC uses a total of 22 data queues, including both input and output, while the CB-based NoC needs 30 data queues. Assuming that the length of each data queue is the same in the two NoCs, CTC allows to save up to 35% of storage space for this particular case study. In the MWD case, since most of

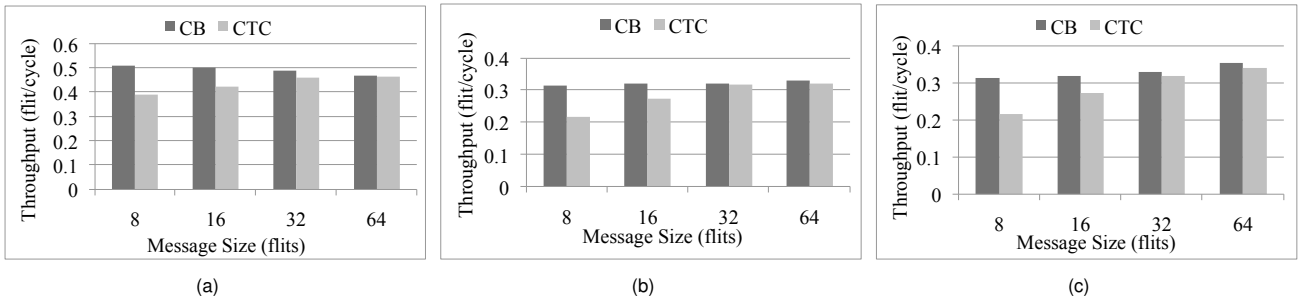

Figure 9. NoC Throughput as function of the message size when $K = 4$ flit-credits.

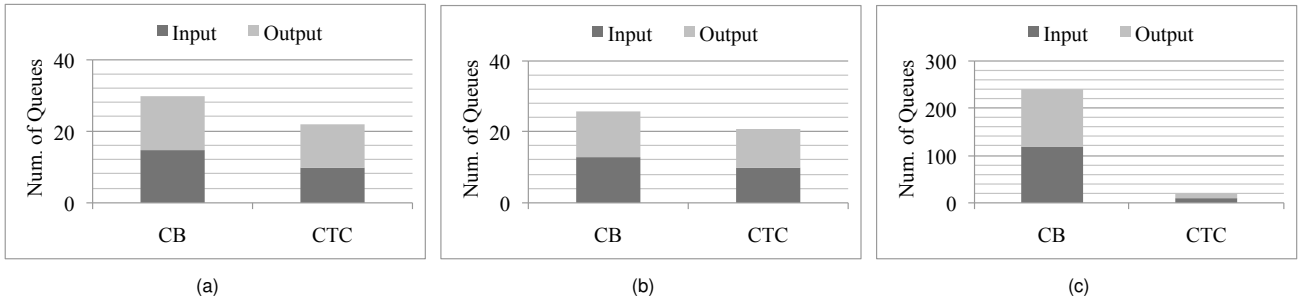

Figure 10. Breakdown of the aggregate number of input and output queues in the NoC NIs for the VOPD application.

the cores communicate only with one other core the two NoCs have many interface with similar structures. Still, even in this case CTC allows to save up to 24% of storage as reported in Fig. 10.

Finally, the Uniform traffic pattern represent the special case where each node may communicate with any other node of the NoC. In this case clearly CB is an expensive solution as it requires $N - 1$ queues where N is the number of nodes in the network. 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 ejected 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 the single input data-queue and the output queue.

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

6 Conclusions

Message-dependent deadlock is a destructive event that, even if rare [22], must be properly addressed to guarantee the correct behavior of a network. The credit based (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 micro-architectural structures.

We proposed the Connection Then Credits (CTC) end-to-end flow control protocol as an area-efficient solution to the message-dependent 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 Credit Based 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 of the message-dependent deadlock problem for throughput-driven stream processing applications.

Acknowledgments

This work is partially supported by the University of Bologna and ST Microelectronics project funds for the project "*Modeling and Analysis of Network Protocols for Spidergon-Based Networks on Chip*", the National Science Foundation (Award #: 0541278) and by the GSRC focus center, one of the five research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program.

References

[1] K. V. Anjan and T. M. Pinkston. DISHA: a deadlock recovery scheme for fully adaptive routing. In *IPPS '95: Proceedings of the 9th International Symposium on Parallel Processing*, pages 537–543, Apr. 1995.

[2] L. Benini and G. D. Micheli. Networks on chip: A new SoC paradigm. *IEEE Computer*, 49(2/3):70–71, Jan. 2002.

[3] D. Bertozzi et al. NoC synthesis flow for customized domain specific multiprocessor systems-on-chip. *IEEE Trans. on Parallel and Distributed Systems*, 16(2):113–129, Feb. 2005.

[4] L. Bononi, N. Concer, M. Grammatikakis, M. Coppola, and R. Locatelli. NoC topologies exploration based on mapping and simulation models. In *DSD '07: Proceedings of the 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools*, pages 543–546, Aug. 2007.

[5] N. Concer, M. Petracca, and L. P. Carloni. Distributed flit-buffer flow control for networks-on-chip. In *The Proceedings of the Sixth International Conference on Hardware/Software Codesign & System Synthesis (CODES+ISSS)*, pages 215–220, Oct. 2008.

[6] M. Coppola. Keynote lecture: Spidergon STNoC: the communication infrastructure for multiprocessor architectures. In *International Forum on Application-Specific Multi-Processor SoC*, Jun. 2008.

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

[8] W. J. Dally and B. Towles. Route packets, not wires: On-chip interconnection networks. In *Proceedings of the Design Automation Conference*, pages 684–689, Jun. 2001.

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

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

[11] J. Duato, S. Yalamanchili, and L. Ni. *Interconnection Networks. An Engineering Approach*. Morgan Kaufmann Publishers, San Mateo, CA, 2003.

[12] Y. Durand, C. Bernard, and D. Lattard. FAUST : On-chip distributed architecture for a 4G baseband modem SoC, in. In *Proceedings of Design and Reuse IP-SOC*, pages 51–55, Nov. 2005.

[13] O. P. Gangwal, A. Rădulescu, K.Goossens, S. G. Pestana, and E. Rijpkema. Building predictable systems on chip: An analysis of guaranteed communication in the Æthereal network on chip. In P. van der Stok, editor, *Dynamic and Robust Streaming In And Between Connected Consumer-Electronics Devices*, volume 3 of *Philips Research Book Series*, chapter 1, pages 1–36. Springer, 2005.

[14] A. Hansson, M. Coenen, and K. Goossens. Undisrupted quality-of-service during reconfiguration of multiple applications in networks on chip. *Proceedings of the conference on Design, automation and test in Europe*, pages 954–959, Apr. 2007.

[15] A. Hansson, K. Goossens, and A. Rădulescu. Avoiding message-dependent deadlock in network-based systems on chip. *VLSI Design*, 2007:Article ID 95859, 10 pages, May 2007.

[16] A. Hemani et al. Network on chip: An architecture for billion transistor era. In *18th IEEE NorChip Conference*, page 117, Nov. 2000.

[17] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2006.

[18] H. D. Kubiatowicz. *Integrated Shared-Memory and Message-Passing Communication in the Alewife Multiprocessor*. PhD thesis, Massachusetts Institute of Technology. Boston, 1997.

[19] S. Murali, M. Coenen, A. Radulescu, K. Goossens, and G. D. Micheli. Mapping and configuration methods for multi-use-case networks on chips. *Proceedings of the 2006 conference on Asia South Pacific design automation*, pages 146–151, 2006.

[20] S. Murali and G. D. Micheli. An application-specific design methodology for STbus crossbar generation. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 1176–1181, Apr. 2005.

[21] T. M. Pinkston and J. Shin. Trends toward on-chip networked microsystems. *Intl. J. High Performance Computing and Networking*, 3(1):3–18, 2001.

[22] Y. H. Song and T. M. Pinkston. A progressive approach to handling message-dependent deadlock in parallel computer systems. *IEEE Trans. on Parallel and Distributed Systems*, 14(3):259–275, 2003.