Document downloaded from:

http://hdl.handle.net/10251/171216

This paper must be cited as:

Picornell-Sanjuan, T.; Flich Cardo, J.; Hernández Luz, C.; Duato Marín, JF. (2021). Enforcing Predictability of Many-cores with DCFNoC. IEEE Transactions on Computers. 70(2):270-283. https://doi.org/10.1109/TC.2020.2987797

The final publication is available at

https://doi.org/10.1109/TC.2020.2987797

Copyright Institute of Electrical and Electronics Engineers

Additional Information

© 2021 IEEE. Personal use of this material is permitted. Permissíon from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertisíng or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

Enforcing Predictability of Many-cores with DCFNoC

Tomás Picornell, *Student Member, IEEE,* José Flich, *Senior Member, IEEE,*
Carles Hernández, *Member, IEEE,* and José Duato, *Senior Member, IEEE*

Abstract—The ever need for higher performance forces industry to include technology based on multi-processors system on chip (MPSoCs) in their safety-critical embedded systems. MPSoCs include a network-on-chip (NoC) to interconnect the cores between them and with memory and the rest of shared resources. Unfortunately, the inclusion of NoCs compromises guaranteeing time predictability as network-level conflicts may occur. To overcome this problem, in this paper we propose DCFNoC, a new time-predictable NoC design paradigm where conflicts within the network are eliminated by design. This new paradigm builds on top of the Channel Dependency Graph (CDG) in order to deterministically avoid network conflicts. The network guarantees predictability to applications and is able to naturally inject messages using a TDM period equal to the optimal theoretical bound without the need of using a computationally demanding offline process. DCFNoC is integrated in a tile-based many-core system and adapted to its memory hierarchy. Our results show that DCFNoC guarantees time predictability avoiding network interference among multiple running applications. DCFNoC always guarantees performance and also improves wormhole performance in a 4×4 setting by a factor of $3.7\times$ when interference traffic is injected. For a 8×8 network differences are even larger. In addition, DCFNoC obtains a total area saving of 10.79% over a standard wormhole implementation.

Index Terms—real-time systems, safety-critical systems, MPSoCs, time division multiplexing (TDM), time predictable network.

1 Introduction

Many-core processors are increasingly considered in safety critical systems to cope with the high computational power demands of new applications (e.g autonomous driving systems). Unfortunately, current commercial off-the-shelf (COTS) many-core processor designs cannot be used in the context of autonomous safety-related applications since safety standards (e.g ISO26262 in the automotive domain) impose strict requirements that cannot be generally met with these platforms. One of the major obstacles relates to the inability for determining good quality (i.e low and tight) worst-case execution time (WCET) estimates for the software functions running on top of COTS many-core platforms. This is in turn a consequence of the way shared resources included in these systems are handled. Due to the high number of cores competing for the shared resources, the network-on-chip (NoC) becomes the resource with highest impact in contention. In particular, wormhole NoCs implemented in COTS many-cores have been shown to introduce a huge negative impact in the quality of WCET estimates [1].

To address this problem, in this work we focus on the design of a new contention-free NoC [2] design (DCFNoC) using time-division multiplexing (TDM) such as the ones proposed in [3] and [4]. DCFNoC relies on the utilization of the channel dependency graph (CDGs) associated to the routing algorithm to understand the existing packet dependencies (contention) and eliminate them by the introduction of delays at strategic router output ports. The CDG helps to identify where conflicts may occur in the NoC and how these conflicts are always the consequence of dependencies between packets reaching their destination with a variable number of hops. With the addition of delays at output ports of specific routers we ensure transmissions are naturally serialised and conflicts are avoided. With this methodology, the network is significantly simplified as it does not need scheduling tables. Also, buffers and associated flow control and arbitration logic within routers can be removed. DCFNoC provides the following benefits to our system:

- **Straightforward scheduling.** Contention is simply avoided when it is enforced that no more than one node is injecting a packet in the same time slot.
- **Scalability.** DCFNoC provides better scheduling periods than competing TDM approaches and is able to find schedules in arbitrarily large NoCs.
- **Constant network message latency.** Once a message is injected into the network a path is guaranteed to reach its destination node with the same delay since all paths are forced to have the same delay.
- **Timing isolation.** Heterogeneous network bandwidth allocation can be assigned to nodes preventing bandwidth starvation or network interference.

DCFNoC is suitable for mixed-criticality systems since it provides the timing isolation requirements imposed by safety critical standards to consolidate several tasks of different criticality levels. Additionally, the good properties of DCFNoC allow easily allocating heterogeneous bandwidth guarantees to the different tasks executed in the MPSoC thus, tailoring the performance bounds of the NoC to the needs of the different applications. Even though, network interferences and bandwidth starvation are avoided. We formalize the DCFNoC properties and prove that when

- *Tomás Picornell, José Flich, Carles Hernández and José Duato are with Department of Computer Architecture, Universitat Politècnica de València, Valencia, 46022, Spain.*

- *Email: tompic@gap.upv.es (Tomás Picornell)*
- *Email: jflich@disca.upv.es (José Flich)*
- *Email: carherlu@upv.es (Carles Hernández)*
- *Email: jduato@disca.upv.es (José Duato)*

This work has been supported by MINECO under Grant BES-2016-076885, by MINECO and funds from the European ERDF *under Grants TIN2015-66972-C05-1-R and RTI2018-098156-B-C51, and by the EC H2020 RECIPE project Grant 801137.*

using this NoC paradigm packet transmission is conflict-free. Besides, we demonstrate that DCFNoC can be obtained for any given topology and any deterministic routing algorithm.

As a second major contribution in this paper, we integrate DCFNoC in a many-core processor, adjusting the design to meet determinism at application level. Indeed, we demonstrate DCFNoC can be smoothly integrated in the many-core system and we show performance guarantees can be achieved with our DCFNoC, contrary to the ones provided by a baseline default wormhole NoC.

The rest of this paper is organized as follows. Section 2 formally describes DCFNoC theory. Section 3 gives a general methodology for designing conflict-free networks by applying this theory. Section 4 shows a detailed router design adapted to this new methodology. Section 5 briefly details a flexible bandwidth allocation mechanism, for DCFNoC. Section 6 describes the DCFNoC integration in a manycore system. Section 7 provides detailed analysis of performance of the proposed DCFNoC manycore integration without losing its time predictability property. Section 8 discusses related work and, finally, conclusions are given in Section 9.

2 Delayed Conflict-Free Network

This section presents and formalizes DCFNoC, a TDM-based NoC design paradigm in which conflicts are avoided by serializing message transmissions. Although DCFNoC can be used for long messages, for the sake of explanation we consider only single-flit messages. We have the following assumptions:

A1 *A node can generate messages targeting any other node at any rate, even broadcast messages.*
A2 *A message arriving at its destination is eventually consumed assuming an end-to-end flow control is used preventing final node contention.*
A3 *Once a message is injected into the network a path is guaranteed to reach its destination node.*
A4 *Messages are forwarded following any deterministic or partially adaptive deadlock-free routing algorithm.*
A5 *All TDM slots composing a period have the same length.*
A6 *Every router and link within the network have the same delay (we assume one cycle).*

The following definitions develop a notation for describing networks, routing functions, conflicts, and dependency graphs. A summary of notation is given in Table 1.

2.1 Definitions

Definition 1. An interconnection network I is a strongly connected multigraph defined as $I = G(N, C)$. The vertices of the multigraph N represent the set of communication nodes. The arcs of the multigraph C represent the set of communication channels. The source node of a given channel c_i is denoted as s_i and the destination node as d_i. Figure 1 shows a 2D mesh topology of an interconnection network.

Definition 2. A routing function $R : C \times N \times N \rightarrow C$ provides the output channel c_y for a message located in the current node n_c at an input channel c_x and with destination node n_d:

$$R(c_x, n_c, n_d) = c_y \quad (1)$$

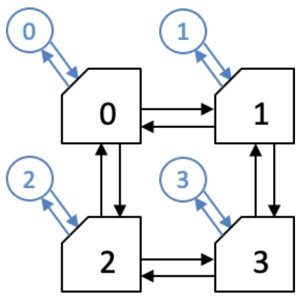


Fig. 1: 2×2 mesh network. End nodes shown as circles.

Routing functions will determine the existence and severity of contention within the network as they set the communication flows. As highlighted in other works [1], [5], [6] NoC contention is a consequence of direct interference between messages, which are in turn a consequence of channel dependencies. Next, we provide a formal definition for channel dependencies.

Definition 3. There is a direct channel dependency from channel c_y to channel c_x, for a given interconnection network I and routing function R if c_y is needed immediately after c_x for a message located at node n_c with destination n_d.

Definition 4. A channel dependency graph CDG for a given interconnection network I and routing function R, is a directed graph, $CDG = G(C, E)$. The vertices of the graph are the channels of I and the arcs of the graph are direct channel dependencies between channels determined by R in the following way:

$$E = (c_x, c_y) \mid R(c_x, n_c, n_d) = c_y \text{ for some } n \in N \quad (2)$$

Figure 2 depicts a channel dependency graph of 2D mesh topology based on Dimension Order Routing (DOR [7]) algorithm. In this plot squares represent the vertices (channels), and the arcs represent channel dependencies. Circles represent injection and ejection channels.

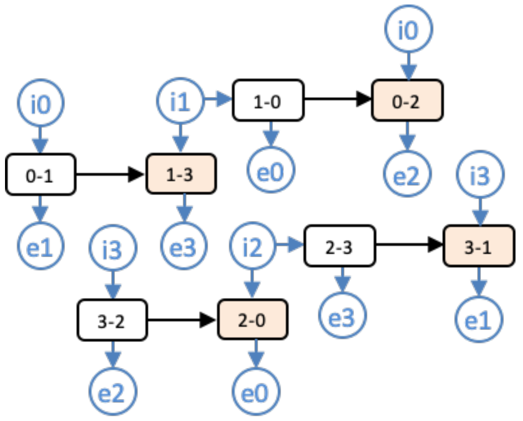


Fig. 2: CDG for the 2×2 mesh topology and the DOR routing algorithm.

Definition 5. A direct conflict $\overline{C(M_a, M_b, t_m)}$ between a pair of messages M_a and M_b for a given interconnection network I, and routing function R may arise at time t_m if

$$R(c_x, n, d_a) = R(c_y, n, d_b) \text{ for some } n \in N, \quad (3)$$

that is, M_a and M_b are in the same node and request the same channel at the same cycle time.

Definition 6. A layered channel dependency graph CDG_l for a given interconnection network I and routing function R, is a layered directed graph, $CDG_l = G_l(C, E)$. The vertices of the graph are the channels of I and the arcs of the graph are the channel dependencies defined by R. In contrast to CDG, all the vertices of CDG_l have an assigned layer id L_h where h represents the position of the layer. Any vertex (channel) is assigned a unique layer id. Therefore, $L_h(v)$ is a bijective function. A channel c_y with a direct dependency with channel c_x will have a higher layer id:

$$L_h(c_y) > L_h(c_x) \text{ if } R(c_x, n_c, n_d) = c_y \tag{4}$$

Note that in order to build a CDG_l the associated CDG must be acyclic. Therefore, the routing algorithm R must be a deterministic one or a partially adaptive one. Figure 3 shows the CDG_l of the 2D mesh with DOR routing. The CDG_l serves us to clearly identify potential conflicts within the network as follows. Let us assume links and routers have a delay of one cycle each and on every cycle no more than one end node injects a message in the network. If we use only dependencies not crossing a layer in the CDG_l (black arrows in the figure) then all messages will take the same amount of time to traverse the path and at every cycle there will be one message at each layer. If, however, we allow dependencies crossing layers (red arcs in the figure) to be used, then conflicts may occur at a given channel. Notice channels are located only in one layer. Indeed, the layer L_h represents the relative cycle time from injection when a specific channel along a path $P(s_i, d_i)$ is used. Assuming assumption A6 we can deduce that every layer has one cycle delay.

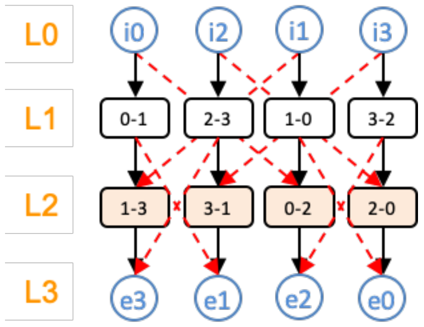


Fig. 3: CDG_l obtained from CDG.

In order to remove all potential conflicts we just need to enforce two messages will not be located on the same layer at the same time (one-message-per-layer rule). To do so, we define a *delayed layered channel dependency graph*.

Definition 7. A delayed layered channel dependency graph CDG_{dl} for a given interconnection network I and routing function R, is a layered direct graph $CDG_{dl} = G_l(C, E)$, in which additional delays are introduced to remove potential conflicts. In this graph, as in CDG_l, all the vertices are assigned to a specific layer L_h. In the CDG_{dl} extra delays are introduced for channel dependencies crossing layers. Every delay is layered also in the CDG_{dl}. Therefore, all paths $P(s_i, d_i)$ have the same delay $\overline{D(P(s_i, d_i))}$.

Figure 4 shows parts of a CDG_{dl}. Notice that all paths have the same latency as injection channels are located at L0 and ejection channels are located at L3. Indeed, let H be the network diameter, c_i the injection channel and c_e the ejection channel. Thereby, and assuming A6, the delay for any path is:

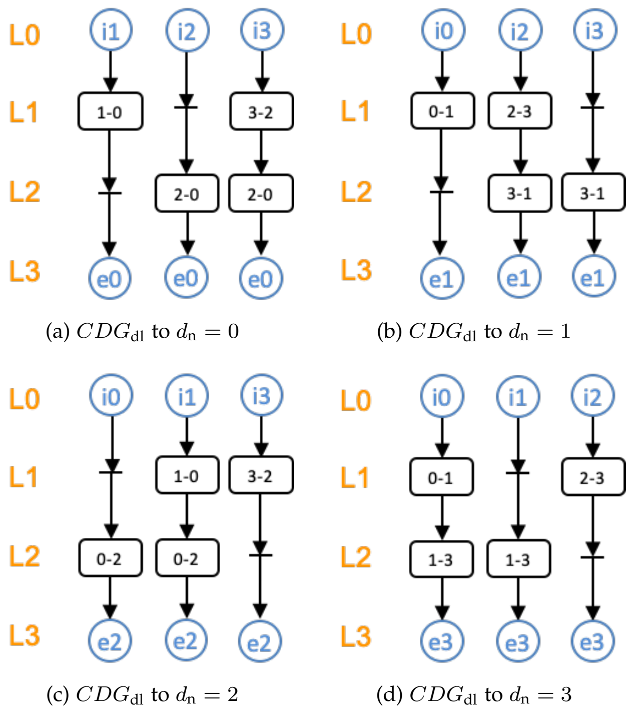
$$\forall P(s_i, d_i) \in CDG_{dl}$$
$$\left\{ \overline{D(c_i, P(s_i, d_i), c_e)} \equiv L\{H\} + 2, \right. \tag{5}$$

In other words, for every possible path in the CDG_{dl} connecting a pair of source and destination nodes (s_i,d_i), the delay of a path is equal to the time required to traverse the set of layers corresponding to the network diameter plus two (injection and ejection layers).

Forcing all messages to traverse the same number of layers allows us to serialize transmissions and avoid conflicts. However, this requires controlling the injection. To do so, we rely on time-division multiplexing. A TDM arbiter operates by periodically repeating a schedule, with a fixed number of time slots t_slot. The scheduler comprises a number of slots, each corresponding to single resource access with bounded execution time in cycles.

Definition 8. A TDM period P_TDM is a set of time slots t_slot, where each slot can be assigned to a single node n_i to control injection slots, $P_\mathrm{TDM} = N \times t_\mathrm{slot}$.

We assume all slots have the same length, according to A5. This ensures every node can only inject messages in a given time slot and that only one node is injecting in a particular cycle.



(a) CDG_dl to $d_\mathrm{n} = 0$ (b) CDG_dl to $d_\mathrm{n} = 1$

(c) CDG_dl to $d_\mathrm{n} = 2$ (d) CDG_dl to $d_\mathrm{n} = 3$

Fig. 4: CDG_dl by destination node.

Theorem 1. *An interconnection network I for which a CDG_{dl} can be derived is conflict-free if injection is controlled in such a way that only a message is injected in a given t_{slot}.*

Proof Sketch. We construct the proof by contradiction. Let us assume there is a conflict $\overline{C(M_a, M_b, t_m)}$ between two messages M_a and M_b. If a conflict exists then a channel c_y will be requested in the same cycle t_m by both messages.

TABLE 1: Summary of Notation

Sign	Description
I	interconnection network,
C	the set of channels,
N	the set of nodes,
c_{c}	a current channel,
s_{i}	source node,
d_{i}	destination node,
n_{i}	a node,
R	a routing function,
C_1	a channel subset,
L_{x}	an assigned layer,
M_{x}	a message,
$P(s_{\mathrm{i}}, d_{\mathrm{i}})$	a path between src to dst nodes,
$\overline{D}$	the delay of a layer or a path,
t_{m}	specific cycle time,
$\overline{C(M_{\mathrm{a}}, M_{\mathrm{b}}, t_{\mathrm{m}})}$	a conflict between two messages at a time t_{m},
CDG	a channel dependency graph,
E	the edges of CDG,
CDG_{l}	a layered CDG,
L_h	a layer of CDG_{l} in position h,
CDG_{dl}	a delayed layered CDG,
H	the network diameter,
$I(c_x)$	a injection channel,
$E(c_y)$	a ejection channel,
t_{slot}	a time slot,
P_{TDM}	a TDM period,

The channel will be mapped in the CDG_{dl} in a given layer $L_h(c_y)$. The distance from the injection channel to c_y is the same for both messages. Therefore, as each layer in the CDG_{dl} implies one cycle delay, the delay between injection and c_y is the same $D(M_a, c_y) = D(M_b, c_y)$. This means both messages have been injected in the same cycle time which contradicts the injection rule where only one end node can inject at a time.

The previous proof is straightforward. Indeed, assuming the existence of a CDG_{dl} it is clear conflicts are not present in the network if messages are serialized at injection time. Therefore, the complexity comes when building the CDG_{dl} for a given network I and routing function R. In the next section we provide a general algorithm for such purpose and demonstrate we can build the CDG_{dl} for deterministic and partially adaptative routing algorithms.

Although an injection of one flit per cycle is a performance limitation the focus of this paper is to provide a clear mechanism to guarantee performance guarantees. On top of DCFNoC we can apply new methods to improve base performance. For example, to inject more than one message we must ensure that messages from two nodes per cycle injected in the same slot do not share any resource along their path. To exploit disjoint paths between messages a previous notification phase is needed. Nodes must know future transmissions and avoid resource sharing in the same slot. Along notification phase all nodes agree on the slot assignment for pending messages to transmit following the priority rules. After the notification phase, the transmission phase starts following slot assignment for each message. In addition, notification and transmission phase can be overlapped by using different DCFNoC networks to potentially improve DCFNoC performance. Improvements on top of DCFNoC are, however, left for future work.

3 Algorithms for the DCFNoC Design Methodology

In this section we propose a methodology for designing TDM-based networks relying on the DCFNoC approach. The proposed methodology consists of two steps. In the first step we start from the CDG and derive the CDG_l. Then, in the second step we construct the CDG_{dl} by inserting delays at certain channels.

3.1 CDG_{l} Algorithm

The algorithm is shown in Algorithm 1. First, in lines 6-8 the CDG is copied to the CDG_l structure and all channels are labelled with an unassigned layer. Then, for every possible path (lines 10-27) the channels are visited and a layer id is assigned to each channel along the paths. The injection channel is assigned always to layer 0 (lines 14-15) whereas router channels are visited in the order set by the path and incremental layers are assigned (lines 19-20). Notice that channels may be already assigned by a previous path. In that case, the channel layer is inspected and if the layer is lower than the one to be assigned by the current path (lines 21-22) then an update_tree call is performed. This function searches the graph and increments the layer of channels with direct and indirect dependencies with the channel by an offset. This guarantees that the channel will have a higher layer than the previous one in the current path. In the case the channel already has a higher layer then nothing is done. After each hop the layer, the input channel and the current node in the path are updated for the next hop (lines 23-25).

```
 1: function build_CDGl(CDG, I, R)
 2:  path p
 3:  channel c
 4:  hop h
 5:  layer l
 6:  CDGl = CDG
 7:  for every channel in CDGl (c)
 8:    CDGl.c.layer = unassigned
 9:  end
10:  for every path (p)
11:    cx = injection_channel(p)
12:    node = p.src
13:    l = 0
14:    if (CDGl.cx.layer == unassigned)
15:      CDGl.cx.layer = l
16:    l = l + 1
17:    for every hop of p (h)
18:      cy = R(cx, node, p.dst)
19:      if (CDGl.cy.layer == unassigned)
20:        CDGl.cy.layer = l
21:      elsif (CDGl.cy.layer<=l)
22:        update_tree(CDGl, cy, l−CDGl.cy)
23:      l = CDGl.cy + 1
24:      cx = cy
25:      node = I.node.next(cy)
26:    endfor
27:  endfor
28: end function
```

Alg. 1: Algorithm for CDG_l

3.2 CDG_{dl} Algorithm

Once we have the CDG_l algorithm we proceed to obtain the CDG_{dl}. Basically we need to add delays to some channel dependencies in order to ensure every path will have the same length in time and that every path will cross all layers. To do this, we add a new field to each channel dependency (arcs in CDG, CDG_l and DCG_{dl}) representing the delay in cycles that need to be enforced as shows Algorithm 2. As a first action, the algorithm copies CDG_l into CDG_{dl} (line 5), then for each pair of channels c_x and c_y of CDG_{dl} (lines 6-7) check if they have a direct dependency (line 8). If so, then the delay of the output channel is set to the difference between layers of both channels (line 9).

```
 1: function build_CDGdl(CDGl, R)
 3:   channel cx
 4:   channel cy
 5:   CDGdl = CDGl
 6:   for every channel in CDGdl (cx)
 7:     for every channel in CDGdl (cy)
 8:       if (R(cx, any, any) == cy)
 9:         CDGdl.arc(cx,cy).delay =
              CDGdl.cy.layer − CDGdl.cx.layer
10:       end
11:     endfor
12: end function
```

Alg. 2: Algorithm for CDG_{dl}

Theorem 2. *Given an interconnection network I, and a deterministic (or partially adaptive) routing function R, the CDG_l and CDG_{dl} can always be obtained and are acyclic.*

Proof Sketch. Given R is deterministic or partially adaptive guarantees the CDG will be acyclic. Therefore, we guarantee that the algorithm used to obtain $CDGl$ and CDG_{dl} prevent cycles from appearing. Both CDG_l and CDG_{dl} have the same structure but only one or two new fields are added (the layer and the delay) to each arc (channel dependency). The set of edges and arcs are the same with the same configuration. Therefore, the same graph shape is inherited. Thus, CDG_l and CDG_{dl} are acyclic as well with added information. As we simply copy the CDG into CDG_l and CDG_l into CDG_{dl} then we guarantee both can always be obtained given CDG is available.

Theorem 3. *Given a path P defined from a routing function R for a network I and an associated CDG_l, the path crosses always channels in increasing layered order.*

Proof Sketch. The way the algorithm is defined guarantees a channel c_y with a direct dependency with channel c_x will have assigned a higher layer. Indeed, $L_h(c_y) = L_h(c_x)+1$. As a path is a list of direct channel dependencies, each hop along P the channel used will have a higher layer assigned.

Theorem 4. *Given a path P for a network I, a routing function R, and an associated CDG_{dl}, the path has a delay of $2 + H$ where H is the diameter of the network.*

Proof Sketch. The path is a set of channels with direct dependencies in the CDG_{dl}. Each channel dependency has an associated delay which is the difference between layers of each channel involved in the dependency. As the depth of the CDG_{dl} is $H + 2$ the delays associated with the channel sum up $H + 2$.

4 Router Design

The structure of the DCFNoC router is shown in Figure 5. This router consists of multiplexers, registers and OR gates.

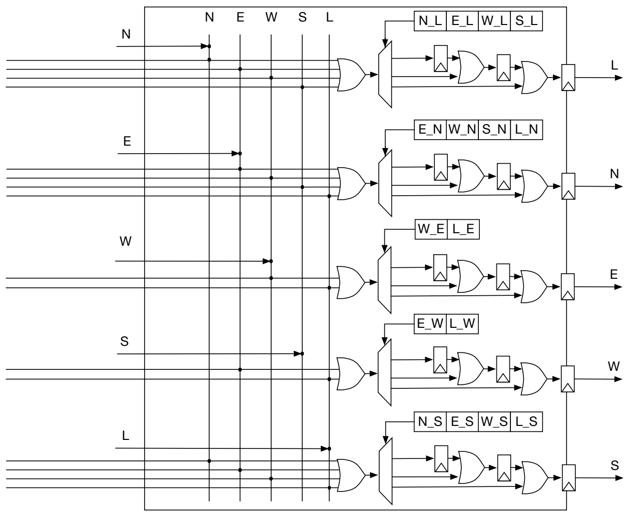


Fig. 5: DCFNoC router input/output ports connections with output delay registers.

This simple design leads to low resource utilization, high frequency, and low power consumption as we will show in Section 7.

The router implements five ports and the DOR routing algorithm. Messages at input ports are routed and latched at the corresponding output port without needing any arbitration logic nor flow control. However, the router supports the case of receiving multiple conflict-free messages through different ports at the same time, and also several of them targeting the same output port but targeting different delay latches. Each output port implements a de-multiplexer with several single-cycle delay latches. A registered configuration vector programs the de-multiplexer on each output port. Input port arrival ID is used to index the configuration register and set appropriately the de-multiplexer. Notice that depending on the routing algorithm the configuration register may have a varying number of slots, from two to four when DOR is used. This depends on the maximum number of output dependencies of a given link accounted in the CDG. Notice that each configuration slot will impose a varying number of delay cycles to the message pipeline transmission. This will enable the proper appliance of the CDG_{dl} methodology.

As an example Figure 6 shows delays introduced by DCFNoC for paths $0 \rightarrow 3$ and $2 \rightarrow 3$ in a 2×2 mesh following the example provided in Figure 4d. As we can see in the plot, the latency for both paths is three cycles since they traverse the same number of latches. Both, injection and ejection end nodes are also shown (represented by circles). Note that the extra cycle delay of path 2 to 3 will be set at output port *local* of router 3. In a $N \times M$ Mesh, the maximum number of extra cycle delays implemented in each output port is $(N - 1) + (M - 1) - 1$.

5 Flexible Bandwidth Allocation

One of the main advantages of DCFNoC over state-of-the-art TDM approaches is that conflict-free transmission can be ensured by simply enforcing no more than one end node injects a message at each time slot. This property emanates from the fact that DCFNoC can be seen as a logical shared bus and thus, conflict-free message transmission can

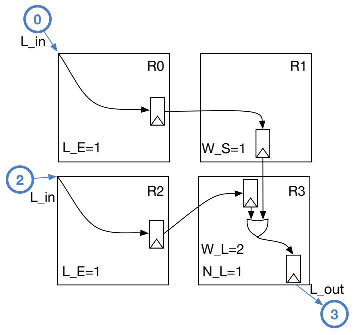


Fig. 6: DCFNoC mesh with output delay registers for paths $0 \rightarrow 3$ and $2 \rightarrow 3$.

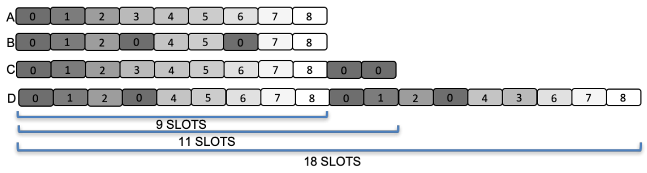


Fig. 7: Different bandwidth allocation options. Each box represents an injection slot and each label indicates the end node the slot is assigned to.

be ensured by simply enforcing the atomic utilization of time slots. This property can be exploited to implement heterogeneous bandwidth allocation schemes across end nodes to accommodate the communication requirements of the different applications running in the system. For instance, heterogeneous bandwidth allocation is a desirable NoC feature in the context of automotive applications. Automotive applications using AUTOSAR are composed of several runnables that can be executed in parallel and have different computing and communication requirements [8].

Heterogeneous bandwidth allocation can be easily implemented in DCFNoC at the edges. Figure 7 shows 4 potential allocation windows (A, B, C, and D) in a 3×3 NoC. The first allocation (A) is the one corresponding to an homogeneous bandwidth allocation strategy in which all nodes get the same bandwidth (1/9). Example B shows the case in which nodes 3 and 6 are inactive and this bandwidth is assigned to end node 0 that gets 3/9 of performance guarantees. Example C shows how increasing the period from 9 to 11 can be used to assign node 0 3/11 of the total bandwidth while the rest get 1/11. Finally, example D shows a period of 18 cycles in which node 0 gets 4/18, and nodes 3 and 5 get 1/18 each. Each of the rest of end nodes get 2/18 of the total bandwidth. In general, DCFNoC allows using fine-grained bandwidth allocation to match different applications needs.

DCFNoC is application agnostic and can be configured to fit the application bandwidth requirements. Indeed, a profile of the application is usually obtained and the network is configured to adapt to the traffic requirements between end nodes. Each end node is then configured with some assigned slots which lets the node to achieve a certain bandwidth of the network.

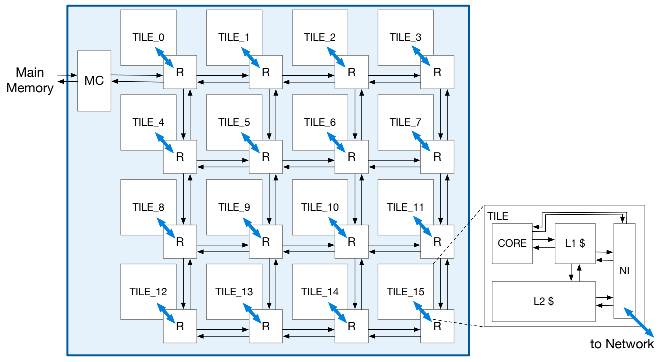


Fig. 8: Baseline many-core architecture.

6 Integrating DCFNoC into a Many-core Design

This section describes the integration of DCFNoC into a many-core design. We start from an existing many-core processor architecture as the one depicted in Figure 8. The design, described in Verilog RTL, is based on several identical tiles interconnected using a standard NoC.

6.1 Tile Architecture

Each tile includes a 32-bit in-order core with L1 private instruction and data caches. A shared L2 cache bank is included also on each tile. All the L2 cache banks from all tiles form the L2 cache of the many-core. To keep data coherent, a coherence protocol is implemented both at L1 and L2 cache levels. The coherence protocol relies on directory structures at L2 level. Both the core and cache memories are interconnected via the Network Interface (NI) module, which provides connectivity between resources within the tile and to resources to/from other tiles. The NI is connected to a router which, in turn, is connected to routers of neighbouring tiles, building a 2D mesh topology.

6.2 Network Interface

The many-core architecture has a wide variety of communication needs. Indeed, memory requests are triggered by the cores as well as coherence requests between memory resources are exchanged. In addition, debug and monitoring information is communicated between the resources. To deal with this communication overhead and complexity, a sophisticated NI is used, depicted in Figure 9.

Seven injector (to net) and ejector (from net) modules are defined. The core uses three modules: L1I (instruction cache), NCA (non-cacheable addresses) and CORE (read-/write to specific control registers). The remaining resources (L1 data cache, the memory controller associated to the tile, the L2 cache bank of the tile, and the control register bank of the tile) have one additional module each.

The injector modules are connected both to the ejector modules (for intra-tile traffic) and to the network inject module (for inter-tile traffic). In the case of inter-tile traffic, serializers are used to adapt the data width on each specific case. Ejector modules are connected also in a similar way to injector modules and to the network eject module. Deserialisers are used to adapt the data width of the network.

The network inject module implements similar logic of a router output port. In the many-core architecture the routers implement virtual networks (VNs) to separate data traffic. A multiplexer separates every input in virtual networks. One input buffer is used for each VN supported. A switch allocator (SA) module is used to assign network resources to messages and to grant access to the eject link. The network eject module is much simpler as it only demultiplexes incoming messages from the network into the corresponding virtual network (VN). A two slot buffer is used on each VN at eject in order to guarantee 100% network throughput.

6.3 Modifications to Include TDM and DCFNoC

In order to integrate DCFNoC we modify the network inject and network eject modules. The remaining NI components are not modified. Moreover, the routers will be replaced by the DCFNoC router presented above. Figure 9 depicts a general view of this integration at NI level. As we can see, VN multiplexing is performed at the entry point of network inject module, thereby messages corresponding to the same VN are multiplexed using a round robin arbiter and allocated at the input port queues (shaded in green). The previous large multiplexer is replaced by one multiplexer per VN.

6.3.1 End-to-end Flow Control

The DCFNoC routers do not implement flow control. However, end-to-end flow control will still be needed since applications may saturate end nodes. To support this functionality, the NI implements an end-to-end Stop&Go flow control protocol based on notification messages and injection filters placed at every NI module (shaded in orange). At network eject module one output buffer is allocated per VN.

When an output buffer reaches the Stop threshold the notification table is updated by using *Update_notification* signal. At network inject module the notification table generates a notification broadcast message to update every node filter allocated at network inject module to stop sending messages with this end node as destination. When a Stop notification broadcast is received at eject module the notification filter is updated by using *Update_filter* signal. Only outgoing messages with this destination node are blocked. Once the saturated destination node reaches the Go threshold the notification table is updated to resume the communication by sending a Go notification broadcast message. At injection time the Stop&Go filter avoid message loss when the destination node experiences saturation.

Notification messages can use preallocated slots for their transmission. Although this would impact performance (bandwidth wastage) the amount of notification messages is negligible. Only when end-point queues fill over a threshold a notification messages would be sent. With a proper design, this rarely occurs. In addition a side DCFNoC can be used for this light weight traffic.

6.3.2 Enforcing Deadlock Freedom

As DCFNoC is a buffer-free network (although it contains latches) flow control between routers is not needed. Indeed, a message that is injected into the network will be forwarded without any stop following the delays and links imposed at design time. This means there is no chance of deadlock within the network. Also, broadcast messages are injected and follow XY paths. Those messages duplicate when needed to reach all destinations. However, every message copy follows the same delay approach using the latches and therefore, can not be blocked within the network. The only deadlock that could occur is at the edges of the network where injection and ejection buffers are used. Those buffers have been sized properly and an end-to-end flow control is used to prevent any message overflow. Deadlock is avoided by using different buffers at the edges of the network to store messages of different types (e.g. requests and responses), thus preventing protocol-induced deadlocks.

6.3.3 TDM scheduler

For TDM management a simple scheduler is implemented at the network inject module. We use a TDM scheduler with a TDM slot wheel where each slot indicates the node ID that can inject in the time slot. It is important to remark that every node has the same information stored at its TDM scheduler.

In our many-core system every node must be able to inject a message in its assigned slot, thus a TDM slot should be sized according to the number of flits for the largest message. However, both single-flit and multi-flit messages (six flits) co-exist and therefore, our scheduler needs to deal with two message sizes effectively. Long messages are data read transactions of 576 bits long divided in 6 flits of 96 bits each. We avoid using more than six flits to achieve a short TDM period. The number of flits in a multi-flit message directly affects the TDM period length.

A trivial approach to deal with multiple message sizes is to define TDM slots of different sizes and send single-flit and multi-flit messages via the same network. In this design the TDM slot wheel needs to combine single-flit slots and multiple-flit slots for every node ID resulting in a longer TDM period for both short and long messages. A TDM period determines the average amount of time every message is waiting to be injected into the network and also proportional to the injection bandwidth. Consequently, a longer TDM period implies lower performance guarantees.

To improve performance guarantees we also explore a second approach in which single-flit and multiple-flit messages are split and use different TDM schedulers and DCFNoC networks. In this setup the short message and long message scheduler implement slots of different duration (one and six in our manycore setup). By doing this, every node is able to inject either a short or a long message, even both at the same time. Thus, performance guarantees and injection bandwidth are significantly better in this latter approach. Assuming one assigned TDM slot per node ID and flow control notification messages using a different network, the maximum time a message gets delayed at TDM scheduler until finding its slot is $(N-1)*(\#flits)$. For a 16-core system a short message delay time is $(16-1)*1=15$ and for long messages $(16-1)*6=90$. However, when all the messages are scheduled using the same TDM scheduler and DCFNoC the maximum amount of time that a messages can be waiting to get injected is $(N-1)+(N-1)*(\#flits)$ being $\#flits$ the number of flits in a long message. In case notification messages use additional TDM slots of the same short messages TDM scheduler, when splitting short and long messages in different schedulers, a short message delay time is $2*(N-1)$ and long messages are not affected. Contrary, when all messages are scheduled in the same TDM scheduler, the maximum waiting time is

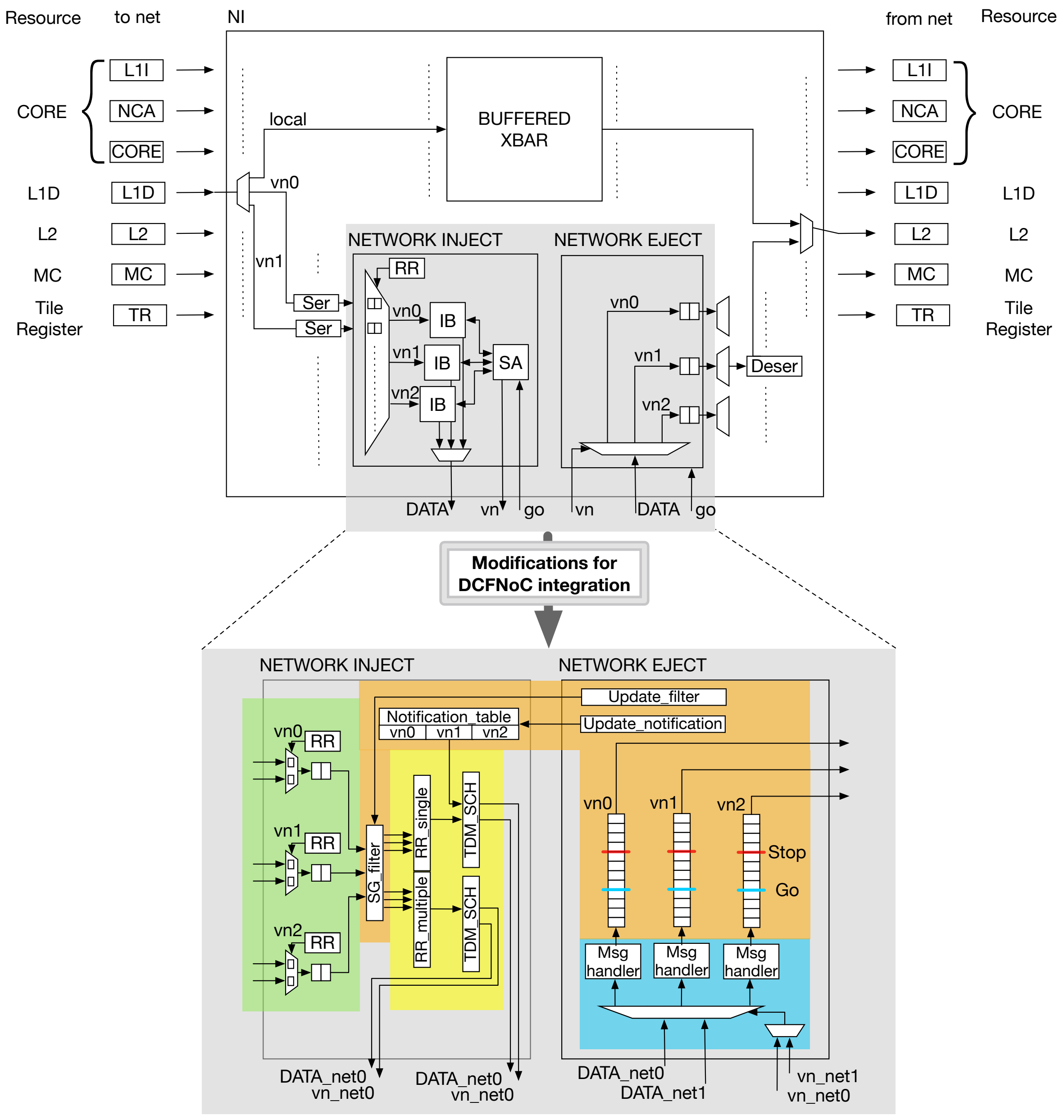


Fig. 9: Network interface controller using Wormhole network and modifications to include TDM and support for two DCFNoC networks.

$2*(N-1)+(N-1)*(\#flits)$ being $\#flits$ the number of flits in a long message.

Figure 9 shows the NI implements two TDM schedulers to separate single-flit and multi-flit messages (shaded in yellow). For the implementation of the two parallel TDM schedulers the network inject module separates messages by length and send them using the corresponding network.

6.3.4 *Network Ejection Module*

At ejection, incoming messages are first multiplexed by VN and later the message handler module is the responsible to establish the order between single-flit and multi-flit corresponding to the same VN (shaded in blue). In case two messages providing from the same virtual network are incoming the message handler serializes the messages in order to preserve the order among the flits they contain and avoid corrupted messages. For example, if a multi-flit message composed of six flits have delivered thee flits and arrives a single-flit message the message handler buffers the single-flit message until the remaining three flits corresponding to the multi-flit message arrives.

Additionally, the NI implements receiving buffers from the end node. Those buffers will be used to store messages generated by the end node until the proper time slot is used to inject the messages. Therefore, the NI decouples generation from the injection.

7 Evaluation Results

In this section we first compare the performance guarantees provided by DCFNoC with the ones provided by similar TDM approaches. We also analyse DCFNoC performance guarantees in a manycore system and compare them with

TABLE 2: A comparison of schedule length and network latency.

Top.	Size	Period				
		TLB [3]	ILP [3]	SYM [4]	PhaseNoC [11]	DCFNoC
Mesh	2×2	3	5	-	4	4
	3×3	8	10	28	9	9
	4×4	16	18	59	16	16
	5×5	32	34	112	25	25
	6×6	-	-	-	36	36
	7×7	-	-	-	49	49
	8×8	-	-	481	64	64

the ones provided by wormhole. Finally, to analyze the feasibility of implementing the proposed NoC we provide area and maximum attainable clock frequency and compare these numbers with the ones obtained with a standard wormhole router [7] [9].

7.1 Experimental Setup

We design DCFNoC and the whole manycore infrastructure using verilog RTL which can be synthetized for FPGAs and ASIC. We simulate the system using the Xilinx Vivado [10] RTL simulator. Thus, results presented in the paper match exactly the number of cycles of a potential manycore implementation.

For the experimental setup we use a 2D-mesh NoC topology to interconnect the different tiles of the previously described manycore system and a memory controller (MC). The MC is modeled using the IP provided by Xilinx to communicate with the off-chip DRAM memory. Alternatively, to force the worst contention scenario the NoC is fed by a message system generator (MS) implemented at each network interface using uniform traffic pattern. DCFNoC configuration used includes 96-bit width links to implement a mesh network topology. At network edges we statically allocate 16 1-cycle time slots to nodes. One time slot is equal to $1/16$ of total bandwidth, resulting to a 6.25%.

7.2 Timing Guarantees

For comparison purposes we model DCFNoC and state of the art TDM algorithms. Table 2 shows the scheduling periods of our approach and compares them with the ones obtained by other state-of-the-art proposals. As shown in the table our NoC design achieves the smallest scheduling periods in all configurations. Our approach is able to improve the period of the ILP-based scheduling [3] that is able to find computationally viable schedules for meshes up to 25 nodes. TLB [3] also formulates a minimum period based on theoretical lower bounds, which is almost equal to our schedule period. The improvement in period achieved by our approach w.r.t [3] for a 5×5 mesh is 26.47%. When compared with SYM [4], which is able to find schedules for larger NoCs, our approach reaches a 77.67% improvement for a 5×5 mesh. PhaseNoC [11] has an period equal to our schedule period but this network implements one domain buffer at each router input port incurring in additional latency at every hop.

Figure 10 shows latency results of DCFNoC compared with the optimal ILP schedule proposed in [3] for a 5×5 NoC for different message injection rates. Latency results are computed for randomly generated messages considering that messages can only be injected in the network in their assigned slot. Latency results shown in this plot represent end-to-end latency values. Additionally, for DCFNoC the latency experienced by the messages once injected in the network is the same for all nodes regardless the target destination since all messages always experience the same latency. On the contrary, in ILP [3] and PhaseNoC [11] the latency experienced once a message is injected depends on the amount of hops each of the communication flow traverses. Thus, in the figure we show values for the min, max, and $average$ latency flows.

As shown in Figure 11 for very small injection rates and the smallest NoC sizes DCFNoC latency is slightly worse than the one achieved in ILP [3] for the shortest paths but better for the longest ones. Note that for a 3×3 mesh PhaseNoC and DCFNoC have nearly the same latency. For higher NoC sizes and/or higher injection rates DCFNoC achieves always better results. The reason for this is the smaller period of DCFNoC that decreases the average time each message is waiting until it is aligned with the assigned slot. The smaller period also enables DCFNoC achieve small latency values for higher injection rates. Although PhaseNoC [11] and DCFNoC have the same schedule period, the latency of PhaseNoC is higher. As we can see in Figure 11 DCFNoC has better scalability than other state of the art TDM proposals for high NoC sizes.

To analyze the performance guarantees of the manycore with DCFNoC we have designed a kernel application in which we can vary the percentage of requests to the NoC with respect to the total number of instructions by injecting random non-memory operations in a kernel containing a specific number of cache accesses that miss in L1 and L2 caches. We deploy this benchmark with the ability to have three different percentages of memory accesses (2%, 7% and 15%) in order understand the impact of the communication of a task in the performance guarantees the same can achieve.

To generate a worst-case contention scenario in the NoC we replace the regular cores in the remaining tiles (all but tile 15) with synthetic message generators with destination router 0 (in which memory controller is placed).

First we take measurements of the benchmark running in core 15 when message generators are disconnected to have the baseline timing measurement (*Alone*) for both DCFNoC and wormhole. Later, we switch message generators on, in all cores but the one running the task under analysis, to measure the impact of NoC interference in our application for both designs. For this experiment, message system generators inject multi-flit (*Long*) messages. Figure 12 shows total execution time of the different benchmark versions when they are executed alone (*Alone*) and when other nodes are injecting long messages (Long) at the maximum speed. The first observation we make is that as expected when the task is executed *Alone* with DCFNoC execution time is higher than when using the wormhole setup. The reason for this is that DCFNoC inherently restricts the injection of messages since they can be only injected in the assigned slot and since we are only using one NoC the TDM scheduler period is high. The slowdown introduced by DCFNoC is $6.21\times$, $7.72\times$, and $8.22\times$, for the 2%, 7% and 15% benchmarks, respectively. However, as we will show later this slowdown

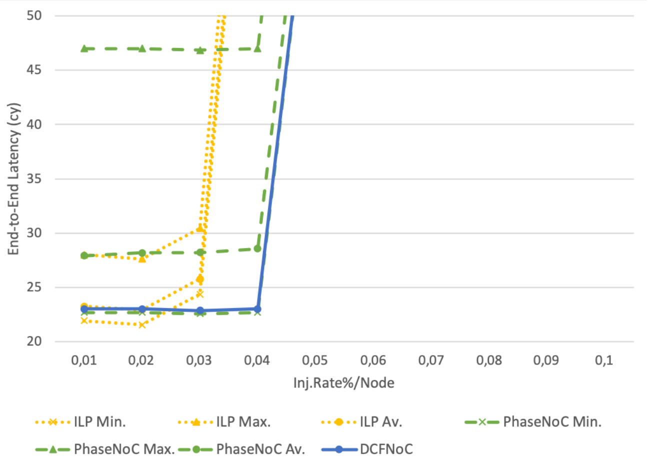


Fig. 10: End-to-End latency of DCFNoC vs ILP [3] and PhaseNoC [11]. Y axes starts at 20 to improve the visibility of the comparison.

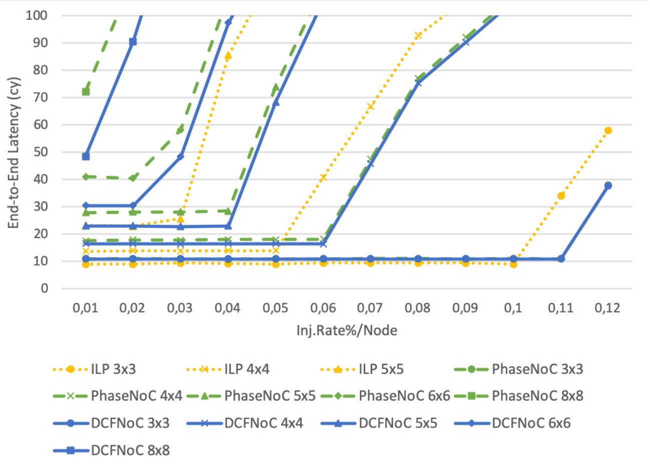


Fig. 11: Scalability of DCFNoC vs ILP [3] and PhaseNoC [11].

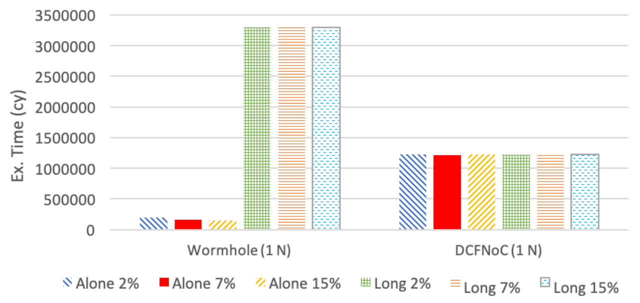


Fig. 12: Execution time for benchmarks with different percentage of memory access instructions.

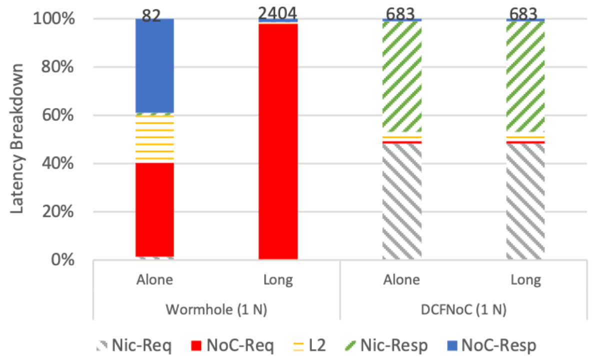


Fig. 13: Average memory transaction latency when using only one network.

is reduced when using two TDM networks. On the contrary, when the task in executed in a high contention scenario the performance of the wormhole setups is degraded significantly (around $19.85\times$ in average) while the performance of DCFNoC is simply unaffected.

Another interesting observation is that the percentage of NoC requests does not have a significant impact in the slowdown of the application in wormhole. Under such heavy message injection the NoC gets saturated quickly. However, requests to the NoC from the task in progress keeps always at the same speed once the NoC is saturated. Note also that since we are focusing in NoC contention we avoid end-node contention by ejecting messages at 1 flit/cycle. Given that the percentage of NoC accesses does not have a significant impact in the worst-contention we use only one of the kernels in the remaining experiments.

7.3 Performance Guarantees of DCFNoC Manycore System

In order to characterize how NoC interference affects to applications running in the manycore system using DCFNoC we launch the benchmark with 2% of memory instructions using only one NoC. In this manycore configuration we can have short (1-flit) and long (6-flit) messages that correspond to NoC requests and responses, respectively. Control messages required by the coherence protocol are 1-flit.

Figure 13 shows average memory transaction latency breakdown. Total average latency is shown at the top of the bars. We break down the total average latency in five components. First two are related to request process, then memory time (L2) and later two for response part. Request and response contains NIC wait time that corresponds to waiting time for inject at the assigned time slot and later the network latency.

Wormhole baseline NoC latency is 32 cycles (4 for injection plus 5 cycles per router and ejection), while DCFNoC takes only 8 cycles (6 for the longest minimal path plus the injection and ejection).

On the contrary, when other tiles inject messages (Long scenario), wormhole NoC experiences message interference along the network and gets congested producing a huge increase of messages waiting time at input port buffers. As a consequence, the application suffers prolonged execution time and long NoC request latencies as shown in Figure 13 (second column). Note that wormhole NoC suffers an average latency of 2404 cycles in this case while DCFNoC average latency keeps constant to 683 cycles. Note that although DCFNoC latency is guaranteed to be equal to the diameter of the NoC the latency of the packets is higher since they are enqueued at the NIC. The small TDM period of DCFNoC decreases the average time each message is waiting until it is aligned with the assigned slot and also enables DCFNoC achieve smaller latency values for higher injection rates.

In order to improve the performance guarantees for both wormhole and DCFNoC we separate short and long messages in two different networks. This allows us to reduce TDM period for short and long messages. Since the TDM period defines the average amount of time every message is waiting to be injected into the network, messages suffer less NIC wait time with this configuration. For wormhole, we also analyze the performance when splitting messages in two virtual networks (VN). The messages share physical links between routers while they use separate input port buffers. Using separate buffers in wormhole avoids head of line blocking problem.

As Figure 14 shows we analyse the application when running alone (*Alone*), also when other tiles inject short messages (*Short*) and when inject long messages (*Long*). These three scenarios are compared in a system using one wormhole network, when using two wormhole networks either virtual or physical (one for short and one for long messages), also when using only one DCFNoC for short and long as well as when using two DCFNoC networks as explained in Section 6. As Figure 14 shows DCFNoC improves execution time of the application by $3.7\times$ with respect to wormhole when short and long messages are divided in two different NoCs due to TDM period reduction. On the contrary, when splitting short and long messages in different NoCs for wormhole does not reduce the contention suffered by our application. This is explained by the fact that wormhole does not restrict the injection of messages as DCFNoC, therefore both wormhole networks get saturated. As the previous case, when the application runs alone in the system the wormhole network takes less execution time than DCFNoC but when using two NoCs the slowdown is only $1.65\times$ as depicted in Figure 14.

Figure 15 shows average memory transaction latency. As shown in the figure, when the application is exposed to maximum contention the wormhole network do not guarantee performance and suffers network inferences increasing NoC request time due to long waiting time of messages at input port buffers. In contrast, DCFNoC preserves bandwidth isolation regardless the amount of network messages and network interferences.

To analyse the scalability of DCFNoC we model a 8×8 manycore system. Figure 16 shows execution time values for a benchmark with 2% of memory instructions when it is executed in the farthest node in the NoC. Unfortunately, we were not able to obtain execution time values for the highest contention scenario for the 8×8 setup with wormhole and thus, we only report execution times of the application executed alone for this NoC. The reason is that when our application is running in core 63 and all other cores are injecting messages at the maximum speed the bandwidth reduction experienced by wormhole is so vast that makes practically impossible to run the application in an detailed RTL simulator as the one we use.

As shown in Figure 16 wormhole NoC running alone has higher execution time since messages need to traverse longer paths. When using DCFNoC the increment of execution time is caused by the network size which in turn affects the TDM period and path length. Execution time impact when using one and two DCFNoC networks is $8.8\times$ and $4\times$, respectively. Note also that although DCFNoC performance decreases, meshes of this size usually have more than one memory controller which reduces the longest paths and allows improving DCFNoC performance.

Figure 17 shows average memory transaction latency. As shown in the figure, request waiting time increases when moving to a 8×8 mesh by $7.17\times$ and $4.93\times$ for one and two DCFNoCs, respectively. However, NoC latency is only affected by hop count being 8 cycles in 4×4 and 16 cycles in 8×8.

DCFNoC provides performance guarantees that are superior to the ones wormhole provides even for smaller NoCs. DCFNoC guaranteed performance for a 8×8 NoC are $2.06\times$ better than those obtained for a 4×4 wormhole NoC. Although wormhole NoC performance is higher when implemented in COTS many-cores, it introduces a significant negative impact in the quality of WCET estimation, preventing any assumption on affordable timing quality of messages. Contrarily, DCFNoC provides perfect timing isolation as well as constant network message latency to fulfill autonomous safety-related applications requirements in many-core systems.

7.4 Performance Evaluation of Real Workloads in a Manycore

In order to evaluate the benefits of using DCFNoC in safety-related applications we have selected (*ndes*) and (*matmult*) benchmarks from mälardalen WCET benchmarks suite [12]. Applications in this benchmark suite have small memory footprint and thus, have very low communication requirements. For comparison purposes we also include a kernel application (*synth*) resembling applications with higher communication needs (5% of total instructions performing NoC requests). Note that large applications cannot be effectively simulated in a detailed RTL manycore model.

The three applications are evaluated in two scenarios: low and high contention. To create this contention scenarios we use synthetic message generators with random destinations injecting long messages at a 6.66% and 100% injection rates. We perform experiments placing the kernels at three different cores $(0, 5, 15)$ in a 4×4 system.

Figure 18 shows a latency comparison for these three benchmarks running in cores $(0, 5, 15)$ in low and high contention scenarios when using wormhole and DCFNoC NoCs. For wormhole we present Highest Observed Values (HOV), represented by blue bars, and average values, represented by a black line. As shown in the plot, the largest HOV is obtained for the kernel with higher communication needs (synth) and in the high contention scenario. This is explained by the fact that high contention conditions are more likely to occur when the NoC is congested. Unfortunately, for hard-real time systems it is not possible to assume HOV represents actual contention bounds. In fact, as shown in [13] the worst possible contention is possible with few NoC requests if they aligned in the worst possible manner. Finally, DCFNoC latency keeps always constant being its value much lower than the average and HOV values of wormhole.

7.5 Flexible Bandwidth Allocation in the Manycore

In order to test the capabilities of the flexible bandwidth allocation of DCFNoC, we perform an experiment on a 4×4 mesh using uniform traffic. Figure 19 depicts application

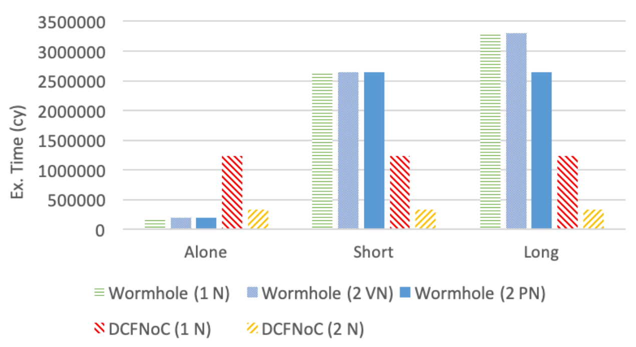


Fig. 14: Execution time when using only one or two networks.

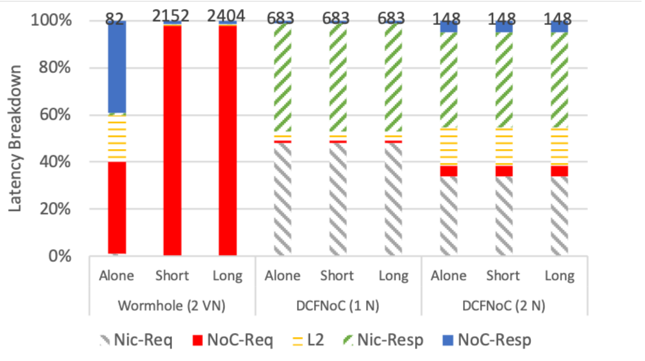


Fig. 15: Average memory transaction latency when using only one or two networks.

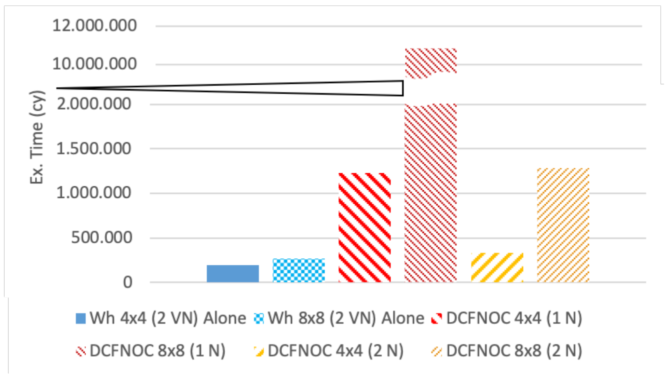


Fig. 16: Scalability of execution time for 4×4 and 8×8.

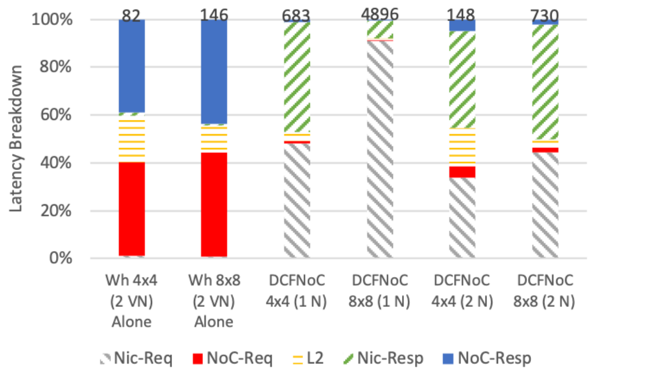


Fig. 17: Scalability of average memory transaction latency for 4×4 and 8×8.

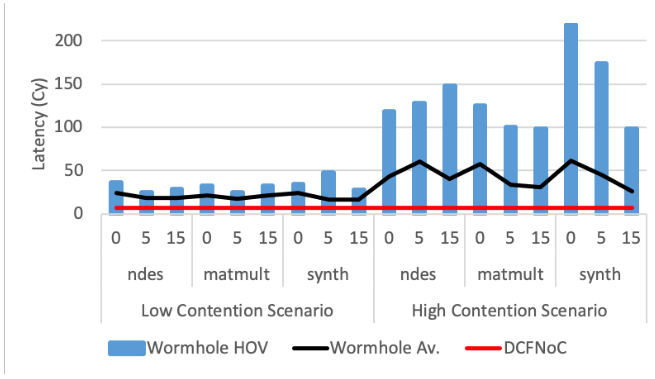


Fig. 18: Latency for different benchmarks execution at different core positions $(0, 5, 15)$ in a 4×4 mesh system using wormhole and DCFNoC NoCs.

execution time when running alone (*Alone*) using wormhole NoC, also when other tiles inject long messages using DCFNoC with $1/16$ of total bandwidth, moreover with a bandwidth of $2/17$.

As figure shows, application execution time using DCFNoC with a bandwidth of $1/16$ increases by 2.14 times compared with *Alone* scenario when using wormhole NoC. As expected, when the application node have 2-cycle assigned time slots of 17, execution takes only 1.6 times. Note also the fact that as Figure 20 shows, transaction latency is improved by 35% due to important NIC average waiting time reduction.

7.6 Area and Frequency of DCFNoC

Maximum operating frequency and area utilization are obtained using Cadence RC Compiler and the 45-nm Nangate library [14]. The wormhole NoC implemented is 64-bit width and uses 8 slot input buffers with Stop&Go flow control. For the implementation results we consider a DCFNoC router for a 8×8 mesh. The wormhole router (WH) used for comparison purposes implements a single virtual channel.

Figure 21 shows area overheads for the two routers when targeting high frequency. The DCFNoC router uses 10.21% less cells than the WH-based one and a total area of $41,586\ mm^2$. As a consequence, we obtain total area savings of 30.42% with a total area of $28,935\ mm^2$. The WH router needs additional logic in order to implement input buffers, flow control logic, routing units, output port arbiters and crossbar interconnect. On the other hand, the DCFNoC router implements very simple routing logic in order to compute the output port, a crossbar interconnect as well as output delay registers.

We have also analyzed the maximum attainable clock frequency by each router. As a first insight, the DCFNoC router is a one-cycle delay router while the wormhole router is a 4-stage pipelined router. Figure 22 shows, as expected, the simpler DCFNoC router design gets a significative boost in clock frequency by improving wormhole router's one by 50%. The critical path of the wormhole router limits clock frequency to 2.22GHz. However, the DCFNoC router exhibits a critical path of 300 ps leading to a clock frequency

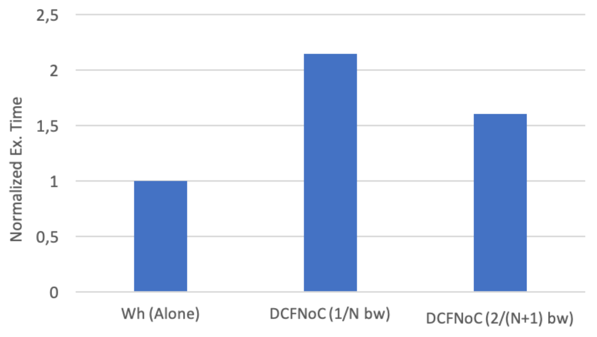


Fig. 19: Execution time using different bandwidth allocations.

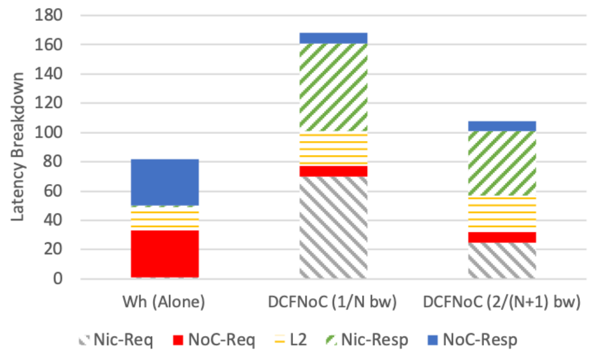


Fig. 20: Average transaction latency using different bandwidth allocations.

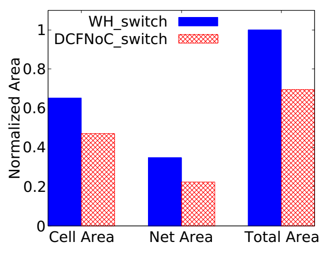


Fig. 21: Switch Area overhead.

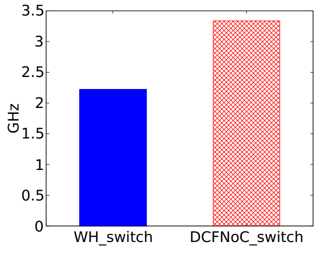


Fig. 22: Switch Maximum attainable clock frequency.

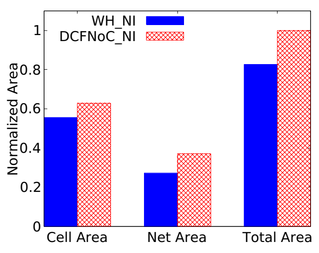


Fig. 23: Network Interface Area overhead.

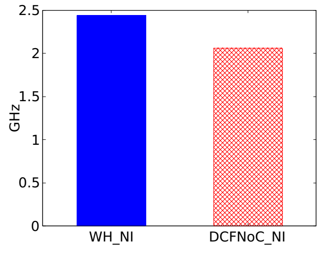


Fig. 24: Network Interface Maximum attainable clock frequency.

of 3.33 GHz.

Figure 23 shows NI area overheads. The DCFNoC NI uses 7.3% more cells than the one used for wormhole and a total area of $25,750\ mm^2$. As a consequence, results in an increment of 17.3% of total area with a total area of $31,137\ mm^2$. Even though wormhole NI multiplexing logic is more complex, it does not require an end to end flow control and therefore consumes less area. The DCFNoC NI uses logic to implement VN multiplexers, TDM arbiters and flow control buffers. End-to-end flow control uses one output buffer per VN.

As a result, total area overhead of DCFNoC router and NI is $60,072\ mm^2$ and $67,336\ mm^2$ for wormhole. As a consequence, we obtain total area savings of 10.79%.

We also analyse the NIC maximum attainable frequency. Figure 24 shows a slowdown of 15.57% in clock frequency of DCFNoC NIC w.r.t wormhole. Although wormhole NI is more complex, it is pipelined in several cycles which allows achieving higher frequencies. On the contrary, TDM arbiters and flow control of DCFNoC NI use only one cycle simplifying flow control notifications. Even with such critical path, DCFNoC NI clock frequency reaches up to 2.06 GHz. Other flow control schemes can be considered in the future to allow reaching higher clock frequencies.

8 Related Work

There is a large set of research and prototype solutions on time-predictable manycore platforms. This section presents related research platforms and discusses the relationship between NoCs and DCFNoC.

We focus on time-predictable NoC design for real-time manycore platforms. Many NoC proposals rely on virtual channels to ensure non-interfering operations across domains [15], [11], and [16]. These solutions implement one domain buffer at each router input port. Thus, no contention arises between different domains but only between VCs within the same domain which improves performance guarantees with respect to conventional wormhole NoC designs. Another existing approach to achieve predictable NoC behaviour is using virtual channel prioritization with flit-level preemption [17]. This approach allows achieving tight latency bounds for the highest priority flows. In general, approaches based on using virtual channels find limitations due to the significant amount of resources required to implement virtual channels. In contrast, our approach removes the need to implement queues at the input buffers.

As in our proposal many previous real-time NoC architectures rely on time-division-multiplexing to achieve predictable message delivery times. However, TDM NoCs have difficulties to find the optimal schedules. TDM schedules can be statically [18], [19], [20], [21], [3], [4], [22] or dynamically computed [23] and may be placed locally at each router [23] for distributed routing or globally in the network interfaces (NIs) for source routing [18], [20].

In the majority of recent proposals ([3] [4] [22]) TDM schedules are allocated and configured off-line to simplify NoC hardware implementation. The theoretical minimum scheduling period for several NoC topologies and sizes are provided in [3] where an ILP formulation is provided to achieve schedules close to the theoretical minimum. However, the computational complexity of the ILP formulation makes unfeasible finding schedules for network sizes beyond 25 nodes. Thus, the approaches in [4] and [22] propose alternative optimization algorithms to find solutions also for larger NoCs. Unfortunately, this comes at the expense of periods that are significantly worse than the theoretical bound. On the contrary, our DCFNoC does not require off-line computations of the schedule since is able to serialize message transmissions in a natural manner leading to a contention-free transmission schedule. The fundamentals of CDGs theory allows DCFNoC to find schedules matching the theoretical minimum period for arbitrarily large NoCs.

9 Conclusion

MPSoCs have been recently introduced in new environments like avionics or automotive. These new domains introduce challenging requirements, such as time predictability and performance isolation, that demand for alternative NoC designs. In this paper we present a many-core system integrating a novel real-time NoC (DCFNoC) to enforce predictability. DCFNoC design is based on the CDGs theory and guarantees by design the avoidance of contention within the NoC providing lower TDM periods and better scalability than previous TDM proposals. Finally, we show that DCFNoC can be smoothly integrated into a manycore design by introducing small modifications at network interface. Our results confirm that the resulting manycore provides performance guarantees that are significantly better that the ones that can be achieved with wormhole NoC designs.

References

[1] M. Panic et al. Modeling High-Performance Wormhole NoCs for Critical Real-Time Embedded Systems. In *RTAS '16*, pp. 267–278.
[2] T. Picornell et al. DCFNoC: A Delayed Conflict-Free Time Division Multiplexing Network on Chip. DAC '19. ACM, 2019.
[3] M. Schoeberl et al. A Statically Scheduled Time-Division-Multiplexed Network-on-Chip for Real-Time Systems. In *NOCS '12*.
[4] F. Brandner and M. Schoeberl. Static Routing in Symmetric Real-time Network-on-chips. RTNS '12, pp. 61–70. ACM.
[5] J. Duato and T. M. Pinkston. A general theory for deadlock-free adaptive routing using a mixed set of resources. *TPDS '01*.
[6] J. Duato. A new theory of deadlock-free adaptive routing in wormhole networks. *TPDS '93*, 4(12):1320–1331.
[7] J. Duato et al. *Interconnection Networks: An Engineering Approach*. IEEE Computer Society Press, 1st edition, 1997.
[8] S. Kehr et al. Parcus: Energy-Aware and Robust Parallelization of AUTOSAR Legacy Applications. In *RTAS '17*, pp. 343–352.
[9] W. J. Dally and H. Aoki. Deadlock-free adaptive routing in multicomputer networks using virtual channels. *TPDS '93*.
[10] Vivado Design Suite 2016.2, 2016.
[11] A. Psarras et al. PhaseNoC: TDM scheduling at the virtual-channel level for efficient network traffic isolation. In *DATE '15*.
[12] J. Gustafsson et al. The Mälardalen WCET Benchmarks - Past, Present and Future. In *WCET 2010*.
[13] M. Slijepcevic et al. Time-randomized wormhole nocs for critical applications. *J. Emerg. Technol. Comput. Syst.*, 15(1), January 2019.
[14] The Nangate Open Cell Library, 45 nm FreePDK, https://projects.si2.org/openeda.si2.org/projects/nangatelib/.
[15] H. M. G. Wassel et al. SurfNoC: A Low Latency and Provably Non-interfering Approach to Secure Networks-on-chip. ISCA '13.
[16] A. Psarras et al. PhaseNoC: Versatile Network Traffic Isolation Through TDM-Scheduled Virtual Channels. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2016.
[17] Z. Shi and A. Burns. Real-time communication analysis for on-chip networks with wormhole switching. In *NOCS '08*.
[18] K. Goossens et al. AEthereal network on chip: concepts, architectures, and implementations. *IEEE Design Test of Computers*, 2005.
[19] A. Hansson et al. Aelite: A flit-synchronous Network on Chip with composable and predictable services. In *DATE '09*, pp. 250–255.
[20] R. A. Stefan et al. dAElite: A TDM NoC Supporting QoS, Multicast, and Fast Connection Set-Up. *IEEE Transactions on Computers*, March 2014.
[21] Z. Lu and A. Jantsch. TDM Virtual-Circuit Configuration for Network-on-Chip. *VLSI*, 16(8):1021–1034, Aug 2008.
[22] R. B. Sørensen et al. A Metaheuristic Scheduler for Time Division Multiplexed Networks-on-Chip. In *ISORC '14*, pp. 309–316.
[23] N. Concer et al. A dynamic and distributed TDM slot-scheduling protocol for QoS-oriented Networks-on-Chip. In *ICCD '11*, Oct.

Tomás Picornell received the MS in Computer and Network Engineering from the Technical University of Valencia (Universitat Politècnica de València), Spain, in 2016. Currently, he is a Ph.D. candidate at the Technical University of Valencia. His research areas include Network-on-Chip architectures with support for time predictability and performance isolation as well as system level solutions to minimize the effects of variability on NoC performance.

José Flich got his PhD in 2001 in Computer Engineering. He is Full Professor at UPV where he leads the research activities related to NoCs. He published over 150 conference and journal papers, and has served in different conference program committees (ISCA, PACT, HPCA, NOCS, ICPP, IPDPS, HiPC, CAC, CASS, ICPADS, ISCC), as program chair (INA-OCMC, CAC) and track co-chair (EUROPAR). José Flich has collaborated with different Institutions (Ferrara, Naples, Catania, Jonkoping, USC) and companies (AMD, Intel, Sun). Current research activities focus routing, coherency protocols and congestion management within NoCs. He has co-invented different routing strategies, reconfiguration and congestion control mechanisms, some of them with high recognition (RECN and LBDR for on-chip networks). He is a member of the Hipeac-2 NoE. He is coeditor of the book "Designing Network-on-Chip Architectures in the Nanoscale Era", and Coordinated the FP7 NaNoC project and leads the H2020 MANGO project.

Carles Hernandez is a senior Researcher at the Universitat Politècnica de València. Previously from 2012 to 2018 he was senior researcher at the CAOS group from Barcelona Supercomputing Center. In 2012 he worked as intern at the IP verification group at Intel Mobile Communications Munich. His area of expertise includes on-chip interconnects, processor design, real-time aware hardware design, and reliability. He is currently co-advising 5 PhD students. Dr. Hernandez participates (has participated) in NaNoC, parMERASA, PROXIMA IP7 and VeTeSS ARTEMIS projects. In 2015 he was granted with a Young Researcher Grant by the Spanish Ministry to conduct research on high-performance and reliable processor design. He was the PI of BSC activities in the FET-HPC RECIPE project on predictable heterogeneous high-performance computing.

José Duato is Professor in the Department of Computer Engineering (DISCA) at the Technical University of Valencia (Universitat Politècnica de València).

His current research interests include interconnection networks, multicore and multiprocessor architectures, and accelerators for deep learning. He published over 500 refereed papers. According to Google Scholar, his publications received more than 15,000 citations. He proposed a theory of deadlock-free adaptive routing that has been used in the design of the routing algorithms for the Cray T3E supercomputer, the on-chip router of the Alpha 21364 microprocessor, and the IBM BlueGene/L supercomputer. He also developed RECN, a scalable congestion management technique, and a very efficient routing algorithm for fat trees that has been incorporated into Sun Microsystem's 3456-port InfiniBand Magnum switch. Prof. Duato led the Advanced Technology Group in the HyperTransport Consortium, and was the main contributor to the High Node Count HyperTransport Specification 1.0. He also led the development of rCUDA, which enables remote virtualized access to GP-GPU accelerators using a CUDA interface.

Prof. Duato is the first author of the book "Interconnection Networks: An Engineering Approach". He also served as a member of the editorial boards of IEEE Transactions on Parallel and Distributed Systems, IEEE Transactions on Computers, and IEEE Computer Architecture Letters.

Prof. Duato was awarded with the National Research Prize in 2009 and the "Rey Jaime I" Prize in 2006. He is a member of the Spanish Royal Academy of Sciences.