Published in IET Computers & Digital Techniques
Received on 6th July 2008
doi: 10.1049/iet-cdt.2008.0093

In Special Issue on Networks on Chip

ISSN 1751-8601

Enabling application-level performance guarantees in network-based systems on chip by applying dataflow analysis

A. Hansson[1] *M. Wiggers*[2] *A. Moonen*[1] *K. Goossens*[3,4] *M. Bekooij*[4]

[1]*Eindhoven University of Technology, Eindhoven, The Netherlands*
[2]*University of Twente, Enschede, The Netherlands*
[3]*Delft University of Technology, Delft, The Netherlands*
[4]*Research, NXP Semiconductors, Eindhoven, The Netherlands*
E-mail: m.a.hansson@tue.nl

Abstract: A growing number of applications, often with real-time requirements, are integrated on the same system on chip (SoC), in the form of hardware and software intellectual property (IP). To facilitate real-time applications, networks on chip (NoC) guarantee bounds on latency and throughput. These bounds, however, only extend to the network interfaces (NI), between the IP and the NoC. To give performance guarantees on the application level, the buffers in the NIs must be sufficiently large for the particular application. At the same time, it is imperative to minimise the size of the NI buffers, as they are major contributors to the area and power consumption of the NoC. Existing buffer-sizing methods use coarse-grained application models, based on linear traffic bounds or periodic producers and consumers, thus severely limiting their applicability. In this work, the authors propose to capture the behaviour of the NoC and the applications using a dataflow model. This enables one to verify the temporal behaviour and to compute buffer sizes using existing dataflow analysis techniques. The authors show what is required from the NoC architecture and demonstrate how to construct an NoC model, with multiple levels of detail. Using the proposed model, buffer sizes are determined for a range of SoC designs with a run time comparable to existing analytical methods, and results comparable to exhaustive simulation. For an application case study, where existing buffer-sizing methods are not applicable, the proposed model enables the verification of end-to-end temporal behaviour.

1 Introduction

Systems on chip (SoC) grow in complexity with an increasing number of independent applications integrated on a single chip [1, 2]. The applications are realised by hardware and software intellectual property (IP), for example the processors and application code, that is reused across platform generations and instances. Additionally, applications are often split into multiple tasks running concurrently, either to improve the power dissipation [3] or to meet real-time requirements that supersede what can be provided by a single processor.

The individual applications have different real-time requirements [4], for example, constraints on periodicity, throughput and latency, that the platform must accommodate. For firm real-time applications, for example, a software-defined radio [5], deadline misses are highly undesirable because of standardisation, for example, upper bounds on the response latency in many wireless standards, or steep quality reduction in the case of misses. Soft real-time applications, for example, an MPEG-2 decoder, can tolerate occasional deadline misses with only modest quality degradation. To guarantee a certain real-time behaviour at the application level, that is, irrespective of other applications, the interconnect must provide guarantees on latency and throughput for the communication between individual IPs [6].

Networks on chip (NoC) offer bounds on latency and throughput by reserving resources on the level of connections [7–10]. However, the bounds only cover the router network and the network interfaces (NI), as shown in Fig. 1*a*. To extend the guarantees to the application level, it is necessary to also include the IPs and the NI decoupling buffers [11–13]. If the buffers are not sufficiently large, the application performance suffers. Thus, for an existing NoC architecture, we must be able to determine the temporal behaviour of the applications mapped to it, given fixed buffer sizes [14]. On the other hand, if we are designing an NoC specifically for a set of applications, then it is desirable to determine minimal sizes for the NI buffers, as they are major contributors to NoC power and silicon area [15].

Existing approaches to compute the size of NI buffers [11, 15] model the application behaviour by means of a traffic characterisation. Buffer sizes are computed such that for every traffic source that adheres to the characterisation, there is always sufficient space in the buffer to allow data production. The work presented in [11] uses linear bounds to characterise traffic, whereas Coenen *et al.* [15] assume strictly periodic producers and consumers. Neither of the approaches allows the availability of buffer space to influence the production time of data. Hence, it is not possible to derive the temporal behaviour given fixed buffer sizes, that is to map a new application to an already existing NoC. Moreover, in [11, 15], it is not possible to capture dependencies between different connections, for example, the dependency between requests and responses in a memory. All these restrictions on the traffic characterisation severely limit the applicability of the methods.

In this paper, we model the NoC and the application behaviour using a dataflow graph [14]. This leverages existing dataflow analysis techniques [16–19] that can: (1) compute the buffer sizes given constraints on the temporal behaviour of the applications, for example, to dimension an NoC architecture for a given set of applications, and (2) determine bounds on the temporal behaviour, that is, a worst-case schedule, for given buffer sizes, for example, to analyse if a new application fits on an existing NoC. This is done, also taking the effect of (non)availability of space into account [17], by showing that a schedule exists with sufficient throughput. The analysis guarantees that buffer capacities are still sufficient to satisfy the application constraints in case data are injected faster.

As the main contribution of this paper, we show how to construct a dataflow graph that conservatively models a connection of any NoC that offers guaranteed latency and throughput. We exemplify the model by constructing several instances based on the Æthereal [9] NoC architecture. The applicability of the model is illustrated by using it together with state-of-the-art dataflow analysis techniques [17] to derive conservative bounds on buffer sizes in the NIs. The computed buffer sizes are compared with existing approaches, for a range of SoC designs. Coupled with fast approximation techniques, buffer sizes are determined with a run time comparable to existing analytical methods [11], and results comparable to exhaustive simulation [15]. For larger SoC designs, where the simulation-based approach is infeasible, our approach finishes in seconds. Moreover, we demonstrate how the dataflow models enable us to capture the behaviour of both the application and the NoC in one model, thus greatly improving the applicability compared to [11, 15]. An application case study, where previous approaches are not applicable, exemplifies how the proposed model is used to determine the temporal behaviour when mapping a new application to an existing NoC instance.

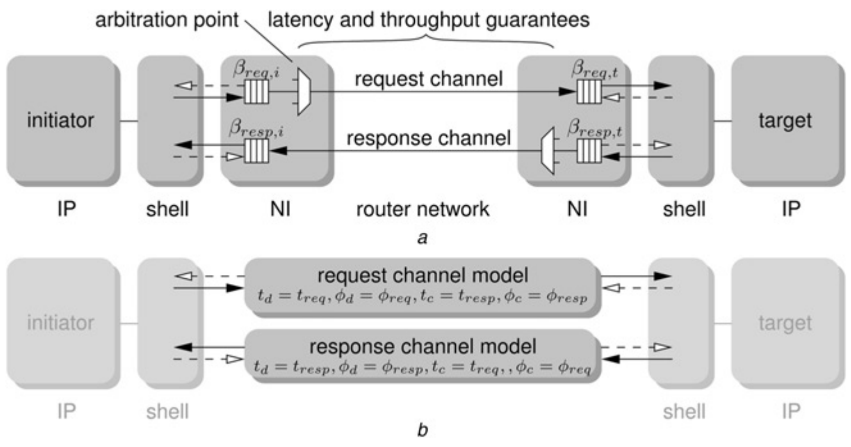

Figure 1 *The hardware blocks and corresponding model of a connection*

a Hardware blocks
b Corresponding model

The remainder of the paper is structured as follows. We start by introducing related work in Section 2. Next, the problem domain is described in Section 3, including an introduction to a general NoC architecture with guaranteed services. Section 4 presents the terminology and concepts of dataflow graphs together with their applications. As the major contribution of this paper, a detailed description of the temporal behaviour of our example NoC architecture is given in Section 5, after which our proposed model of an NoC communication channel is derived in Section 6. Experimental results, using different analysis techniques together with the proposed model, are presented in Section 7, followed by conclusions in Section 8.

2 Related work

Several NoCs offer guarantees on latency and throughput per connection [7–10, 20, 21] and can, thus, be modelled using the techniques proposed in this work. Mango [10] is based on routers with a rate-controlled static-priority scheduler that isolates individual communication connections based on local scheduling decisions. Similar approaches, using different scheduling mechanisms, are used in [20, 21]. Nostrum [8], aSOC [7] and Æthereal [9] implement the guarantees by globally scheduling the communication channels through time-division multiplexing (TDM). The NoC hardware, however, only offers the mechanisms to offer guaranteed performance. It remains a problem to dimension the NoC and allocate resources for the different connections.

Many tools have been presented for designing NoC architectures, with complete design flows for network hardware and software generation [22]. Mapping of IPs to NIs (spatial) and connections to paths (spatial and temporal) [23, 24] are important steps in such a design flow, especially in the presence of real-time requirements. These steps are, however, focused on the NoC internals, and guarantee throughput and latency inside the NoC (the router network in Fig. 1*a*). To guarantee temporal behaviour on the application level, it is necessary to include the application and the buffers between the application and the NoC in the performance analysis [14, 24, 25].

Simulation is a common approach to incorporate the application in the performance analysis [15, 26]. Trace-based buffer sizing [26] provides an optimal bound on the buffer size for the given input traces. However, it does not guarantee that the derived size is sufficient for other traces, that is, for other traces, the applications might even deadlock. Analysis based on statistical models of the application have the same limitations [27] and are, thus, not applicable. The algorithm in [15] uses exhaustive simulation of given periodic traces. While the method provides tight bounds, it does so at the price of a high run time, requiring hours or even days for larger SoC designs. Moreover, the applications, as well as the NoC, are assumed to be completely periodic, thus severely limiting the scope.

More generally applicable than exhaustive simulation is the use of conservative linear bounds on the production and consumption of data [11]. Assuming that it is possible to find such a traffic characterisation, the buffers are sized such that they never overflow. The coarse application model results in a low run time of the analysis, at the cost of large buffers. More importantly, the restrictive application model severely limits the scope of the applications, and it remains a problem to derive a conservative traffic characterisation for a given application. In addition to the limited applicability, both the aforementioned approaches [11, 15] are unable to determine the temporal behaviour for given buffer sizes.

In [14, 24, 25], the application, as well as the NoC, is modelled using dataflow graphs. Thus, in contrast to [11, 15], it is possible to either compute the buffer sizes given the application requirements, or derive bounds on the temporal behaviour (latency and throughput) for given buffer sizes [5, 17, 28]. The latter is demonstrated in [24, 25] where applications are mapped to an existing NoC platform. Although [24, 25] present dataflow models of an NoC, they do so for a specific system architecture, and do not discuss which NoC properties are necessary to derive such a model or how it can be applied to other NoCs. Additionally, neither of the works compares the dataflow analysis with existing approaches for buffer sizing.

Extending on [14], this work gives a detailed exposition on how to construct a cyclo-static dataflow (CSDF) [29] graph that conservatively models an NoC communication channel, and the relation between the architecture and the model. We demonstrate how the model is used to determine buffer sizes for given requirements, and to derive guarantees on the temporal behaviour of an actual application.

3 Problem description

In this paper, we address the problem of constructing an NoC model that enables the verification of the performance requirements of the applications (realised by the IPs), and sizing of the NI buffers. To do this, we must characterise the behaviour and the requirements of the applications, as well as the behaviour of the IPs that run the applications, and the NoC.

The application is implemented by hardware IPs (and potentially their associated software). The IPs, or rather their ports, act as either initiators or targets [30, 31], as illustrated in Fig. 1*a*. Initiators initiate transactions by issuing requests, for example a read or write. One or more targets receive and execute each transaction. Optionally, a transaction also includes a response, returning data or an acknowledgment from the target to the initiator. This transaction model allows for both a distributed shared memory and message-passing communication paradigm.

We assume a general NoC architecture where a protocol shell adapts from the IP protocol, for example, AXI or OCP [32], to a word-based first-in first-out (FIFO) streaming protocol. The shell, thus, performs all necessary handshakes with the IP and (de)serialises the, potentially wider, interface of the IP to a sequence of words that are communicated over the NoC. Both the interface between IP and shell, as well as between and shell NI, use blocking and hence non-lossy flow control [30], [31]. In Fig. 1 (and throughout this work), this is illustrated by a solid arrow for data (valid), and an open-headed dashed arrow for flow control (accept).

The NI is responsible for buffering, packetisation and for implementing the end-to-end services [12, 13]. As shown in Fig. 1*a*, a connection is a bidirectional peer-to-peer interconnection between an initiator and a target. It comprises a request channel, from the initiator to the target, and a response channel in the opposite direction. Every connection is associated with four logical buffers, one on the initiator side and one on the target side, for request and response channels, respectively. Hence, requests are buffered in $\beta_{req,i}$ and $\beta_{req,t}$. Similarly, responses are buffered in $\beta_{resp,t}$ and $\beta_{resp,i}$.

Given an NoC architecture that offers guaranteed latency and throughput for individual channels, it is our problem to construct an NoC channel model, as shown in Fig. 1*b*, that allows us to verify the application requirements and size the NI buffers. We assume that the application models are given as a variable-rate dataflow graphs [33]. Moreover, we assume that all hardware and software tasks, in the application as well as the architecture, only execute when they have input data available and sufficient space in all output buffers; that is, we assume blocking flow control.

4 Analysis model

A dataflow graph can be used to compute buffer capacities and to guarantee the satisfaction of latency and throughput constraints [34]. Fig. 2 shows an example producer–consumer

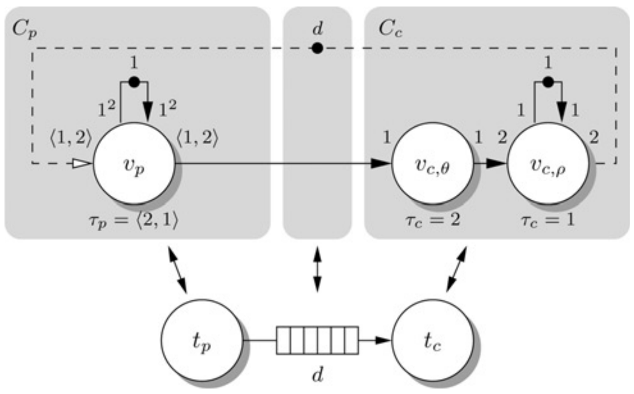

Figure 2 *Example buffer capacity problem with a producer and consumer task*

task graph, with the corresponding dataflow model on top. A task t_p, for example, a software task running on a processor, communicates via a buffer with a task t_c, for example, the scheduler in an NI. If the buffer is full, t_p is stalled, and if the buffer is empty t_c is stalled.

The dataflow graph is divided in components that necessitate a partitioning of the graph. This is exemplified in Fig. 2 by component C_p that models task t_p, and component C_c that models task t_c. Note that the dataflow model also includes an edge from C_c to C_p. This is because a task in the implementation only starts when there are sufficient free buffer locations, that is space, in all its output buffers. To make it easier for the reader, this type of edge is represented with open-headed and dashed arrows, similar to what we have already seen in Fig. 1.

Next, Section 4.1 provides a brief introduction to CSDF graphs and the terminology used in the analysis. For further details and examples, see [17, 19, 34]. Thereafter, in Section 4.2, a sufficient buffer capacity d is computed given a requirement on the minimum throughput of the complete dataflow graph.

4.1 CSDF graphs

A CSDF graph [29] is a directed graph $G = (V, E, \delta, \tau, \pi, \gamma, \kappa)$ that consists of a finite set of actors V, and a set of directed edges, $E = \{(v_i, v_j) | v_i, v_j \in V\}$. Actors synchronise by communicating tokens over edges. This is exemplified in Fig. 2, where tokens, modelling data, flow from actor v_p to the actors $v_{c,\theta}$ and $v_{c,\rho}$, and tokens, modelling space, flow from $v_{c,\rho}$ to v_p. The graph G has an initial token placement $\delta : E \to \mathbb{N}$, as exemplified by the single token on the self edges of v_p and $v_{c,\rho}$, and the d tokens on the edge between them.

An actor v_i has $\kappa(v_i)$ distinct phases of execution, with $\kappa : V \to \mathbb{N}$, and transitions from phase to phase in a cyclic fashion. An actor is enabled to fire when the number of tokens that will be consumed is available on all its input edges. The number of tokens consumed in a firing k by actor v_i is determined by the edge $e = (v_j, v_i)$ and the current phase of the token consuming actor, $\gamma : E \times \mathbb{N} \to \mathbb{N}$, and therefore equals $\gamma(e, ((k-1) \bmod \kappa(v_i)) + 1)$ tokens. The specified number of tokens is consumed atomically from all input edges when the actor is started. By introducing a self edge (with tokens), the number of simultaneous firings of an actor is restricted. This is used in the example graph in Fig. 2, where actor $v_{c,\theta}$ models latency, that is, it does not have a self edge, and actor $v_{c,\rho}$ models throughput, that is, it can only consume and produce tokens at a certain rate.

The response time $\tau(v_i, f)$, $\tau : V \times \mathbb{N} \to \mathbb{R}$, is the difference between the finish and the start time of phase f of actor v_i. The response time of actor v_i in firing k is therefore $\tau(v_i, ((k-1) \bmod \kappa(v_i)) + 1$. When actor v_i

finishes, it atomically produces the specified number of tokens on each output edge $e = (v_i, v_j)$. The number of tokens produced in a phase are denoted by $\pi : E \times \mathbb{N} \rightarrow \mathbb{N}$. In the example, v_p has the response time sequence $\tau_p = \langle 2, 1\rangle$ and $v_{c,\theta}$ has the response time sequence $\tau_{c,\theta} = \langle 2\rangle$. In this graph, v_p consumes and produces two tokens in the first phase from the edges to and from $v_{c,\theta}$ and $v_{c,\rho}$, respectively. For brevity, the notation x^y denotes a vector of length y in which each element has a value x. In the example, this is seen on the self edge of v_p where $1^2 = \langle 1, 1\rangle$.

For edge $e = (v_i, v_j)$, we define $\Pi(e) = \sum_{f=1}^{\kappa(v_i)} (e, f)$ as the number of tokens produced in one cyclo-static period, and $\Gamma(e) = \sum_{f=1}^{\kappa(v_j)} \gamma(e, f)$ as the number of tokens consumed in one cyclo-static period. We further define the actor topology Ψ as an $|E| \times |V|$ matrix, where

$$\Psi_{mi} = \begin{cases} \Pi(e_m) & \text{if } e_m = (v_i, v_j) \text{ and } v_i \neq v_j \\ -\Gamma(e_m) & \text{if } e_m = (v_j, v_i) \text{ and } v_i \neq v_j \\ \Pi(e_m) - \Gamma(e_m) & \text{if } e_m = (v_i, v_i) \\ 0 & \text{otherwise} \end{cases}$$

If the rank of Ψ is $|V| - 1$, then a connected CSDF graph is said to be consistent [29]. For a consistent CSDF graph, there exists a finite (non-empty) schedule that returns the graph to its original token placement. Thus, the implementation it models requires buffers of finite capacity.

We define the vector $\boldsymbol{s}$ of length $|V|$, for which holds $\Psi \boldsymbol{s} = 0$, and which determines the relative firing frequencies of the cyclo-static periods. The repetition vector $\boldsymbol{q}$ of the CSDF graph determines the relative firing frequencies of the actors and is given by

$$\boldsymbol{q} = \Lambda \boldsymbol{s} \quad \text{with } \Lambda_{ik} = \begin{cases} \kappa(v_i) & \text{if } i = k \\ 0 & \text{otherwise} \end{cases}$$

The repetition rate q_i of actor v_i is therefore the number of phases of v_i within one cyclo-static period times the relative firing frequency of the cyclo-static period. For the example in Fig. 2, the vector $\boldsymbol{s}$ is found to be $[2 \quad 6 \quad 3]^{\mathrm{T}}$, and the repetition vector is $\boldsymbol{q} = [4 \quad 6 \quad 3]^{\mathrm{T}}$.

For a strongly connected and consistent CSDF graph, we specify the required minimum throughput as the requirement that every actor v_i needs to fire q_i times in a period μ [19]. In [35], it is shown how also latency requirements can be taken into account. Given such a requirement on the period μ, sufficient buffer capacities are computed, as discussed in Section 4.2.

A CSDF graph is said to execute in a self-timed manner when actors start execution as soon as they are enabled. An important property is that self-timed execution of a strongly connected CSDF graph is monotonic in time [36]. This means that no decrease in response time or start time of any firing k of any actor v_i can lead to a later enabling of firing l of actor v_j. We return to discuss the importance of this property in the following section.

4.2 Buffer capacity computation

Two conditions must hold to compute the buffer capacity using dataflow analysis [17]. First, there must be a one-to-one relation between components in the dataflow graph, and tasks in the implementation. Secondly, the model of each component must be conservative. That is, if component C models task t, then it should hold that if data arrives not later at the input of task t than tokens arrive at the input of component C, then data should be produced not later by task t than tokens are produced by component C. The mentioned relation between token arrival and production times is required to hold for tokens that represent data as well as space. As previously mentioned, the self-timed execution of a strongly connected CSDF graph is monotonic, that is there are no scheduling anomalies in the model. Together with conservative component models, this means that bounds on buffer sizes, throughput and latency are valid even if a component produces or consumes faster in the actual implementation.

In addition to a model that fulfils the aforementioned conditions, which is one of the major contributions of this work, we also need an algorithm to perform the analysis. In this work, we use a low-complexity (polynomial) approximation technique [17] to compute sufficient buffer capacities for CSDF graphs given a throughput constraint and given constraints on the maximum buffer capacities. In this algorithm, a schedule is constructed for each actor individually, which satisfies the throughput constraint. Subsequently, token production and consumption times resulting from these schedules are linearly bounded. Using these linear bounds, sufficient differences in start times of the individual schedules are derived such that tokens are always produced before they are consumed. These minimal differences in start times form the constraints in a network flow problem that computes minimal start times that satisfy these constraints, thereby minimising the required buffer sizes. Buffer sizes are in the end determined using the computed start times together with the linear bounds on token production and consumption times. We refer the reader to [16, 17] for more details.

Together with an appropriate model, the algorithms in, for example [16]–[19], are used to find sufficient buffer capacities. For example, consider the task graph and corresponding dataflow graph in Fig. 2. Assuming that C_p and C_c conservatively model t_p and t_c, a sufficient buffer capacity is computed by finding a token placement d such that the throughput constraint is satisfied, for example, $\mu = 6$. For the example graph, we have that the graph deadlocks with $d = 2$, whereas with $d = 3$ the graph has a period of 16. The throughput constraint is satisfied with $d = 7$.

Next, we proceed with the major contributions of this work, where we leverage the existing algorithms for buffer capacity computation and: (1) show what is required from an NoC for these techniques to be applicable (Section 5), (2) exemplify with a model of a specific NoC (Section 6) and, (3) evaluate the applicability of the proposed model (Section 7).

5 Network architecture

In this section, we discuss what must be considered in the NoC architecture to construct a channel model as shown in Fig. 1*b*. First, the channel flow control must be blocking. In other words, data are never lost if a buffer is full, and reading from an empty buffer causes the reading party to stall until data are available. This is the most common way to implement non-lossy flow control in NoCs, and is used in for example, [7–10, 20] (and probably many others).

Secondly, we require that all the arbitration points of a channel can be modelled as latency-rate servers [37], independent of other channels. Any number of arbitration points is allowed, the resources (such as buffers and links) do not have to be statically partitioned to individual channels. Moreover, the arbitration scheme does not have to be TDM based. In contrast, latency-rate characterisation is possible for any starvation-free arbiter. Examples of NoCs that fulfil these requirements are the TDM-based NoCs in [7–9], all of which have a single arbitration point per channel. Other examples, with multiple arbitration points are [20, 10] that use round-robin and rate-controlled static-priority arbitration, respectively.

For an NoC architecture that fulfils the requirements, the latency and throughput must be conservatively modelled. Although this initially might sound like an easy task, the actual NoC implementation has a wide range of mechanisms (pipelining, arbitration, header insertion, packetisation and so on) that affect the latency and throughput and have to be taken into account. We exemplify this using the Æthereal NoC.

5.1 Æthereal architecture

To model the architecture, we must understand its internals, and what affects the latency and throughput. The Æthereal router network is wormhole switched, and works on the granularity of flow control units (flits). Flits are transported between NIs using contention-free routing [9]. That is, the injection of flits into the network is governed by TDM slot tables in the NI [12], such that flits do not contend, similar to [7, 8]. The NI is, thus, the only arbitration point, in contrast to for example, [10, 20] where every router is an arbitration point. The allocation of contention-free paths and time slots is outside the scope of this work, and we refer the reader to [6, 23] for more details.

Because of the use of source routing, the path is included in every packet in the form of a packet header. The contention-free routing removes the need for buffers in the routers as no packet ever has to wait. It does, however, require that data be always immediately accepted by the NIs, otherwise, the data would have to be dropped by the NI and the performance guarantees would be violated. Æthereal therefore uses credit-based end-to-end flow control per channel [12].

The latency and throughput of a channel is determined by the resources allocated to a connection, and the direction of the channel, that is, request or response. In the case of Æthereal, the resource reservation is captured by the slot table and the path, t_{req}, ϕ_{req}, t_{resp} and ϕ_{resp} for the request and response channels, respectively. As seen in Fig. 1*b*, when analysing the request channel, we must also take the response channel into account and vice versa. This is because of the end-to-end flow control that travels back on the channel in the opposite direction. Note that credits travelling on the request channel do not affect data on the request channel, and similarly for the response channel.

As will be seen in the following sections, the behaviour depends on the number of slots reserved and the distance between them (Section 5.1.1), the number of slots that are used to send headers rather than data (Section 5.1.2), the length of the path through the network (Section 5.1.3), and the availability of flow-control credits (Section 5.1.4). Additionally, a number of architectural constants, summarised in Table 1, affect the temporal behaviour. In the following sections, we present the different contributions in the order they appear when a data word traverses a channel and credits are returned.

5.1.1 Slot table injection:

Data injection is regulated by the forward slot table, t_d, which is a sequence of slot reservations. All slot tables in the NoC have the same number of slots and, thus, have the same period, namely p_n, and the flit size is fixed at s_f throughout the NoC. The number of slots reserved affects the throughput of the channel, whereas the distances between reserved slots

Table 1 Symbols used for the router and NI

Symbol	Description	Unit
s_f	flit size	words
s_h	packet header size	words
p_n	TDM slot table period	cycles
s_p	maximum packet size	flits
s_c	maximum credits per header	words
$\theta_{p,NI}$	NI (de)packetisation latency	cycles
$\theta_{d,NI}$	NI data pipelining latency	cycles
$\theta_{c,NI}$	NI header pipelining latency	cycles

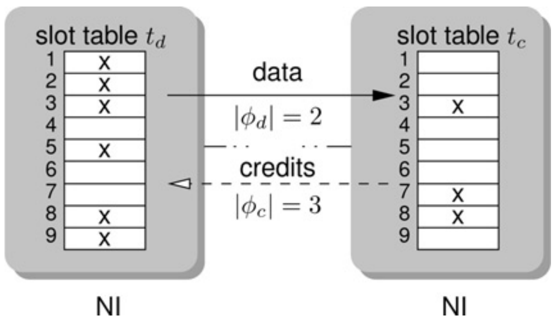

Figure 3 *Example channel with slot tables and paths*

contributes to the latency. In the channel model, we use $|t_d|$ to denote the number of slots reserved, and $d_d(t_d)$ to denote the maximum latency from the arrival of a word until the average data rate can be sustained. Exhaustive search over the possible arrival times is used to determine $d_d(t_d)$. For most practical cases, however, the latter is simply the largest distance between two allocated slots. This is exemplified in Fig. 3, by $t_d = \langle 1, 2, 3, 5, 8, 9 \rangle$ with $|t_d| = 6$ and $d_d(t_d) = 3s_f$, which happens if a word appears in the first cycle of slot 5.

5.1.2 Header insertion: Besides data, the forward channel also carries headers. This affects the throughput of the channel, and it is necessary to bound how much of the capacity be used to send headers rather than data. The header insertion is governed by the NI, based on the reservations in the slot table, and the maximum packet size, s_p. The header has a fixed size of s_h words, with $s_h < s_f$. For a group of consecutive slots, the first one always has to include the path, and hence a packet header. For example, consider t_d in Fig. 3 where slots 5 and 8 include a packet header (slot 1 follows after slot 9). In addition, with $s_p = 4$, also slot 3 includes a packet header (as it comes after 8,9,1 and 2). We must provide a lower bound on the rate of data sent. Therefore we introduce the function $\hat{h}$ that provides an upper bound on the number of headers inserted during a period of p_n. In the example, $\hat{h}(t_d) = 3$, which occurs, for example, if we start in slot 4.

5.1.3 Path latency: As all rate regulation is done in the NI, the traversal of the router network only adds to the latency of the channel, and does not affect the throughput. The latency for a path ϕ depends on the number of hops, denoted $|\phi|$, the pipelining depth of the routers (which is equal to the flit size s_f), and the (de)packetisation latency of the NI, $\theta_{p,\mathrm{NI}}$. The path latency $\theta_p(\phi) = \theta_{p,\mathrm{NI}} + |\phi|s_f$; that is, the time required to (de)packetise the flit plus the time it takes for the complete flit to traverse the NoC. In Fig. 3, $\theta_p(\phi_d) = \theta_{p,\mathrm{NI}} + 2s_f$.

5.1.4 Return of credits: Until now, we have only looked at the data travelling on the forward channel, but there are also credits going in the reverse direction, affecting the latency and throughput of the channel. Whenever a word is scheduled for injection in the router network, a counter in the sending NI is decremented. If the counter reaches zero, no more data are sent for the channel in question. When words are consumed from the receiving NI, credits are accumulated and sent back.

The credits are returned in the packet headers of the reverse channel, with the number of headers depending on the distribution of slots in t_c. To conservatively model the return of credits, we need to determine a lower bound on the rate at which they are sent back, and an upper bound on the latency before they are sent. This is to be compared with the bounds determined for the injection of data in Section 5.1.1.

The function $\check{h}$ provides a lower bound on the number of headers inserted during any interval p_n. For each header, a maximum of s_c credits are sent. With 5 bits reserved for credits, for example, a maximum of 31 credits can be sent in each header. In addition to bounds on the rate of credits, $d_c(t_c)$ denotes the maximum latency between the arrival of credits (i.e. a word is consumed from the receiving NI) until the average credit rate can be sustained. The slot table allocation, as well as the maximum packet size, is taken into account just as for the data. Looking at the example, $d_c(t_c) = 4s_f$ and $\check{h}(t_c) = 2$, which happens when starting in slot 9.

6 Channel model

In this section, we show how to construct a dataflow graph that conservatively models a network channel using the expressions that we derived in Section 5.1. We go from a coarse model in Section 6.1 and successively refine it until we arrive at our final model in Section 6.4. Additionally, Section 6.5 complements the channel model by capturing the behaviour of the protocol shells.

In Section 7, we use the proposed channel models together with models of the application and apply dataflow analysis [17, 19] to the constructed CSDF graph to determine the conservative bounds on the required buffer capacities and to verify application-level performance guarantees.

6.1 Fixed latency (Fig. 4a)

Our first model, depicted in Fig. 4*a*, has only one actor v_{cd}, with a single token on the self edge. This prohibits an execution to start before the previous execution has finished. As seen in the figure, the actor only fires when buffer space is available in the consumer buffer β_c, and then frees up space in the producer buffer β_p. The response time of the actor, appearing below the graph, captures the worst-case latency a data word can ever experience. This happens when a word arrives and there are no credits available in the producer NI. Next, we present the four terms that together constitute the response time.

The first term, $\theta_c(t_c) = \theta_{c,\mathrm{NI}} + d_c(t_c)$, captures the worst-case latency for the injection of credits. The latency is a sum of: (1) the maximum cycles spent updating the credit

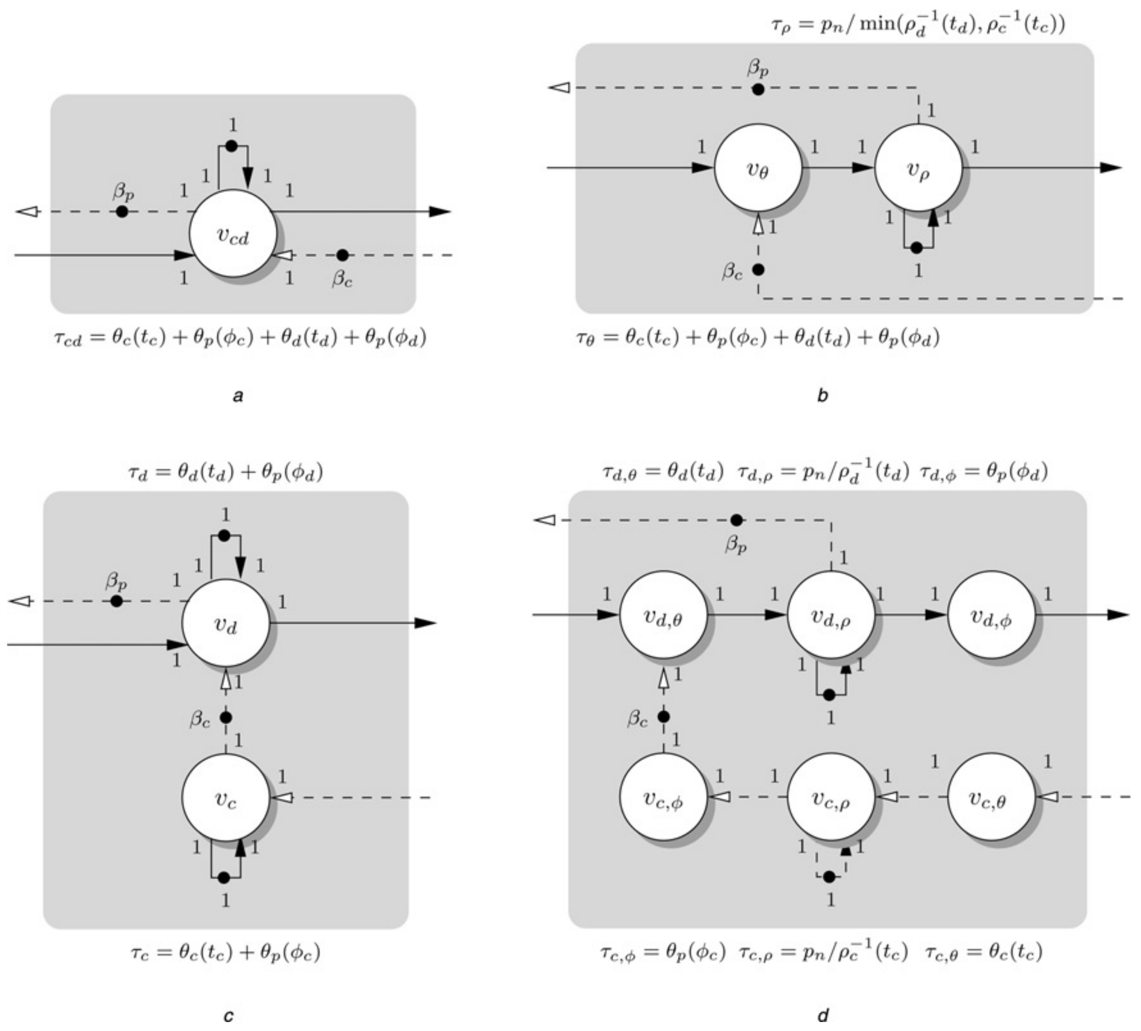
$\tau_{cd} = \theta_c(t_c) + \theta_p(\phi_c) + \theta_d(t_d) + \theta_p(\phi_d)$
v_{cd}
β_p
β_c
a
$\tau_\rho = p_n / \min(\rho_d^{-1}(t_d), \rho_c^{-1}(t_c))$
v_θ
v_ρ
$\tau_\theta = \theta_c(t_c) + \theta_p(\phi_c) + \theta_d(t_d) + \theta_p(\phi_d)$
b
$\tau_d = \theta_d(t_d) + \theta_p(\phi_d)$
v_d
v_c
$\tau_c = \theta_c(t_c) + \theta_p(\phi_c)$
c
$\tau_{d,\theta} = \theta_d(t_d)$ $\tau_{d,\rho} = p_n/\rho_d^{-1}(t_d)$ $\tau_{d,\phi} = \theta_p(\phi_d)$
$v_{d,\theta}$
$v_{d,\rho}$
$v_{d,\phi}$
$v_{c,\phi}$
$v_{c,\rho}$
$v_{c,\theta}$
$\tau_{c,\phi} = \theta_p(\phi_c)$ $\tau_{c,\rho} = p_n/\rho_c^{-1}(t_c)$ $\tau_{c,\theta} = \theta_c(t_c)$
d

Figure 4 *Different channel models*

a Data and credits joined
b Data and credits joined, latency and rate split
c Data and credits split
d Data and credits split, latency and rate split

counter on data consumption and the maximum latency until the credits are seen by the NI scheduler, and (2) the maximum number of cycles without any slots reserved. The second term, $\theta_p(\phi_c)$, corresponds to the time required in the router network to return the credits to the producer NI. With data and credits available, it only remains to bound the time until the data are available in the consumer buffer.

Similar to the injection of credits, $\theta_d(t_d) = \theta_{d,\mathrm{NI}} + d_d(t_d)$, bounds the latency experienced by a data word in the sending NI. The latency consist of: (1) the number of cycles before a word that is accepted by the NI is seen by the scheduler, and (2) the worst-case latency for data. The fourth and last term is attributable to the router network in the forward direction, which adds a latency of $\theta_p(\phi_d)$.

The model in Fig. 4*a* is sufficient to model the NoC channels and perform buffer sizing and application-level performance analysis. It is, however, overly conservative as it does not distinguish between credits and data, and assumes a worst-case arbiter state for every data and credit item that is sent. Note in particular that only latencies appear in the model. The number of slots reserved, for data as well as credits, are not taken into account.

Next, we show how it is possible to refine the model along two different axes. First, by looking over a larger interval we can create less conservative models. If data/credits arrive fast enough, we only have to assume the worst-case state for the first item. For subsequent items, we have more knowledge about the state [37]. This leads to a model where latency and rate is split. Secondly, by distinguishing between the forwarding of data and return of credits, we capture the fact that the two happen in parallel. This leads to a model where data and credits are split. Finally, we present a model that combines both these refinements.

6.2 Split latency and rate (Fig. 4b)

We split our first model into multiple actors according Fig. 4*b*. The difference with Fig. 4*a*, is that the latency, now modelled by v_θ, can be experienced by more than one word at a time, that is, the actor has no self edge. Instead, the actor v_ρ bounds the rate at which data and credits are sent. With one token on the self edge, the response time of the actor is a conservative bound on the length of time it takes to serve one word after an initial latency τ_θ. As seen in the figure, the response time is the period of the TDM wheel, p_n, divided by the minimum of the maximum amount of data and maximum amount of credits. The amount of data is upper bounded by $\rho_d^{-1}(t_d) = s_f|t_d| - \hat{h}(t_d)s_h$, that is, the amount of words reserved during p_n, minus the maximum space required for headers. Similarly, the credits are upper bounded by $\rho_h^{-1}(t_c) = \check{h}(t_c)s_c$. In this channel model, we see that the latency experienced is the sum of the credit and data latency, and the rate is determined by the minimum, that is the most limiting, of the two.

6.3 Split data and credits (Fig. 4c)

Our third model, shown in Fig. 4*c*, splits the data and credits into two different actors, v_d and v_c. Actor v_d, which models the arbitration on the forward channel, fires when it has input data and credits available, as seen by the edge from v_c. The firing of v_d also frees up one buffer space, as seen by the edge going back to the producer. The return of credits is modelled by v_c that fires when a word is consumed from β_c. The response times of the actors, appearing above and below the graph, capture the time it takes for a word/credit to be scheduled by the NI and traverse the path through the network. Compared with our first model, we see that the latency for data and credits now appear as the response times of v_d and v_c, respectively. We also see the asymmetry between the producer buffer β_p, which is local and the consumer buffer β_c, which is located in the receiving NI.

6.4 Final model (Fig. 4d)

By splitting the model into a data and credit part, as well as a latency and rate part, we arrive at our final model, shown in Fig. 4*d*. In the forward direction, data experience scheduling latency and rate regulation in the NI, modelled by $v_{d,\theta}$ and $v_{d,\rho}$. The router network also adds latency, $v_{d,\phi}$. For the return of credits, the situation is similar. The NI is modelled by $v_{c,\theta}$ and $v_{c,\rho}$, and the router network by $v_{c,\phi}$. In our final model, we use independent actors to model latency/throughput and data/credits.

Note that the channel model is independent of the application using it. If the application behaviour changes, or the model of an application is refined, nothing has to be modified in the channel model. Similarly, the application model is not affected if the channel model is replaced or refined, that is, by choosing one of the aforementioned levels of detail.

6.5 Shell model

Thus far, we have only modelled the NI and router network, and to complete the model we must also include any potential protocol shells. The shell is closely coupled to the IP and the protocol it uses. If the IP uses a word-based FIFO protocol, for example, an audio A/D converter, no shell is needed. For a bus protocol like AXI, however, the shell serialises and deserialises the hundreds of parallel command, address and data signals into a sequence of words that are communicated over the NoC.

In Table 2, we show an example of how the read and write bursts of an IP are transformed into a number of words. As seen in the table, the size of a burst, from the perspective of the NI, depends on the burst size of the IP, the transaction type, the channel direction and the size of the message headers. Table 2 shows the case when the IP and the NoC use the same data width. As seen in the table, a read operation requires the read command (plus potential flags) and the address to be sent as a request. The response, in turn, carries the actual data as well as status information. For a write operation, the command, flags and address are sent together with the data (and potentially a write mask). When executing a write operation, the target may also return status information and error codes.

Note that a shell model only captures command handshakes and data transfer for requests and responses. Higher-level protocol issues, for example, coupling between requests and responses or dependencies between different requests from the same initiator are captured in the model of the application, as exemplified in Section 7.2.

Table 2 Burst sizes for different channel types

Transaction	Direction	IP (words)	Shell (words)	Total (words)
read	request	0	$h_{req,r}$	$h_{req,r}$
read	response	b	$h_{resp,r}$	$b + h_{resp,r}$
write	request	b	$h_{req,w}$	$b + h_{req,w}$
write	response	0	$h_{resp,w}$	$h_{resp,w}$

7 Experimental results

In this section, we demonstrate the applicability of the model; first, by comparing the run time and buffer sizes derived using our approach with those of [11, 15] in Section 7.1. and secondly, by showing how to apply it to verify the end-to-end temporal behaviour of an audio post-processing application in Section 7.2.

Throughout our experiments, we assume a word width of 32 bits used by both the IPs and the NoC. The NoC operates at a frequency of 500 MHz and has a flit size $s_f = 3$, a header size $s_h = 1$, a maximum packet size $s_p = 4$, an upper limit on the credits per header $s_c = 31$, pipelining latencies $\theta_{d,\mathrm{NI}} = 2$ and $\theta_{c,\mathrm{NI}} = 2$, and a packetisation latency $\theta_{p,\mathrm{NI}} = 1$ (as defined in Table 1). The shells are assumed to have $h_{req,r} = 2$, $h_{resp,r} = 0$, $h_{req,w} = 2$ and $h_{resp,w} = 0$. Note that the period of the NoC, p_n is determined per NoC instance when allocating resources for the various use-cases [23].

7.1 Buffer sizing

We compare the run time and buffer sizes derived using our approach with those of [11, 15]. Using the terminology of [15], we hereafter refer to the two approaches as analytical and simulated, respectively. Moreover, using our proposed channel model, we also show the differences using the dataflow model with fast approximation algorithms [17] and exhaustive back-tracking [19]. Thus, we use the same model as input, but two different tools in the analysis. The run time is measured by using the Linux command time, looking at the user time, including potential child processes. The reason we use this command is that the dataflow analysis tools are separate binaries, that are called for every connection that is analysed in the NoC design flow [22]. For the dataflow analysis, the time includes the forking of the parent process, the writing of XML files describing the CSDF graphs to disk, and then reading and parsing of those files.

The first step to comparing with [11, 15] is to adopt an equivalent application model (Section 7.1.1). Thereafter, we apply the methodologies to a range of synthetic benchmarks (Section 7.1.2), a mobile phone SoC design (Section 7.1.3), and a set-top box SoC (Section 7.1.4).

7.1.1 Modelling the application: To facilitate a comparison with existing work, we choose to employ a model that subsumes [11, 15], where the input specification is done per connection, and contains a transaction type $a \in \{R, W\}$, determining if a transaction is read or write, a burst size $b \in \mathbb{N}$ in words, and a period $p \in \mathbb{N}$ in network cycles (if the IP and NoC are using different clock frequencies).

Similar to [15], the model is based on the notion of a producer and consumer. To determine the sizes of the four buffers in Fig. 1*a*, where the initiator and the target act as both producers and consumers, we first look at the data going from initiator to target to determine $\beta_{req,i}$ and $\beta_{req,t}$. Thereafter, we swap the roles and let the target be the producer and the initiator the consumer to determine $\beta_{resp,t}$ and $\beta_{resp,i}$.

Dividing the buffer calculation for request and response channels into two separate steps, implicitly assumes that the production and consumption inside the IPs is completely decoupled, that is, there is no relation between consumption and production times of the IP. Again, this is to keep the model comparable to [11, 15], and is to be contrasted with Section 7.2.

Having separate connections for reads and writes assumes that they can make independent progress. This corresponds to the separate read and write data paths of, for example, AXI. Protocols with a more strict ordering, for example, AHB, are therefore not accurately modelled without further additions. The reason we use a more limited model than allowed by dataflow analysis is again the comparison with [11, 15].

Fig. 5*a* shows the models we adopt for periodic producers. Since we model an IP that produces words on a bus interface, we know that a burst of more than one word cannot be produced in one cycle. This is reflected in the model, where only one token is produced per actor phase. Space is acquired in the first b phases, each taking one cycle. Data are released in the last b phases, also taking one cycle each. Both actors have a cumulative response time of p. Note that the model of the consumer is completely symmetrical, with the only difference being which edges represent data and credits. Hence, the consumer acquires data in the first b phases, and releases the space that the data occupied in the last b phases.

The two models capture periodic producers and consumers with half a period of jitter on the burst. If the producer actually acquires its space later, then the buffer capacities computed with this dataflow model are still sufficient. This is because space will arrive in time to allow for consumption times according to the model, which implies that space arrives in time for these later consumption times. If the producer actually releases its data earlier, then also

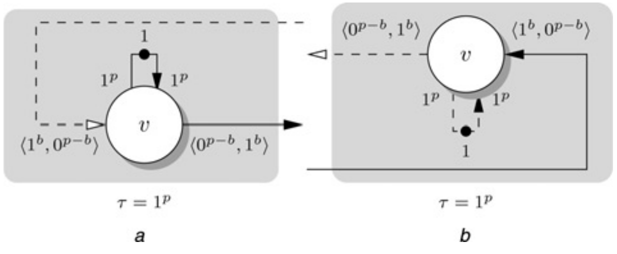
1
1^p
1^p
v
$\langle 1^b, 0^{p-b} \rangle$
$\langle 0^{p-b}, 1^b \rangle$
$\tau = 1^p$
a
$\langle 0^{p-b}, 1^b \rangle$
$\langle 1^b, 0^{p-b} \rangle$
v
1^p
1^p
1
$\tau = 1^p$
b

Figure 5 *IP models used for buffer-size comparison*

a Producer
b Consumer

the computed buffer capacities are sufficient. This is because an earlier production of data can only lead to an earlier enabling of the data consumer, that is, the NI. This is because of the one-to-one relation between components in the implementation and the model, together with the fact that the dataflow model is monotonic in the start and response times. The reasoning for the case in which the IP consumes instead of produces data is symmetric, because in the dataflow graph tokens model both data and space.

7.1.2 Synthetic benchmarks: To assess the performance over a broad range of designs, we choose to compare the different algorithms on a set of randomly generated use-cases. The benchmarks follow the communication patterns of real SoCs, with bottleneck communication, characterising designs with shared off-chip memory, involving a few cores in most communications. All generated designs have 40 cores, with an initiator and a target port, connected by 100 connections. Bandwidth and latency requirements are varied across four bins, respectively. This reflects, for example, a video SoC where video flows have high bandwidth requirements, audio has low bandwidth needs, and the control flows have low bandwidth needs but are latency critical. A total of 1000 benchmarks are evaluated, using the proposed channel model more than 200 000 times in total, with widely varying requirements.

Fig. 6*a* shows the distribution of the total buffering requirements, relative to [11]. As seen in the figure, both the simulation-based algorithm and the algorithms using our dataflow model result in significant reductions on the total buffer size. Averaging over all the benchmarks, we see a reduction of 36% using the dataflow approximation algorithm, 41% using the exhaustive simulation and 44% when applying the exact dataflow analysis. Moreover, the distribution of relative improvement is much wider for the simulation-based algorithm, ranging all the way from 5% up to 45%. The dataflow model, on the other hand, consistently results in an improvement of more than 30%, and even 35% in the case of an exact analysis. The large improvements stem from rather small slot tables (<32 slots), and hence a large bandwidth-allocation granularity (with 32 slots, each slot corresponding to roughly 63 Mbps). Although this leads to an increased burstiness and larger buffers using the analytical method, the dataflow analysis leverages the reduced response times, thereby reducing the buffer sizes.

The run times measured when deriving the aforementioned buffer capacities are also compared with [11], and the relative distribution is shown in Fig. 6*b*. It is clear from the experiment that the dataflow approximation algorithm is roughly one order of magnitude slower than the analytical approach, being on an average 11 times slower. Note though, that the run time is still below a second for an entire SoC design. The exact dataflow analysis and the simulation, on the other hand, are three orders of magnitude slower, averaging at 450 and 520 times the execution time of [11] (but faster algorithms for the exact dataflow analysis exist [18]). The run time of the simulation-based algorithm depends heavily on the period of the producer, network and consumer. As the generated designs use relatively small slot tables, the run times are in the order of minutes.

7.1.3 Mobile phone SoC: A phone SoC with telecom, storage, audio/video decoding, camera image encoding, image preview and 3D gaming constitutes our first design example. The system has 13 cores (27 ports distributed across an ARM, a TriMedia, two DSPs, a rendering engine etc.), one off-chip DDR memory, one on-chip SRAM plus a number of peripherals. Communication is done via memory, and the NoC runs at a frequency of 235 MHz.

The total buffer size and the time needed to derive all buffers, for all the use-cases, are shown in Table 3. As explained in [15], the buffer sizes are determined per use-case and then the maximum for every buffer is used in the architecture. Again, we see that the dataflow-based

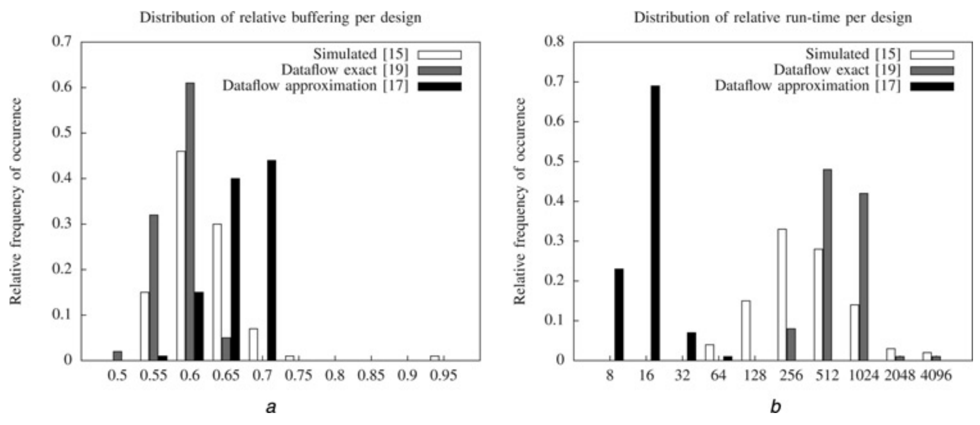

Figure 6 *Comparison of run time and total buffer size for the synthetic benchmarks*

a Relative buffer sizes compared with analytical [11]
b Relative run time compared to analytical [11]

Table 3 Buffer sizes for mobile phone system

Algorithm	Run time, s	Tot. buffers (words)	Impr., %
analytical [11]	0.05	1025	ref
simulated [15]	6845	799	12
dataflow approx. [17]	0.78	721	30
dataflow exact [19]	547	680	34

methods result in improvements of 30 and 34%, respectively. The simulation, however, only reduces the buffers by 12%, thus performing significantly worse than for the synthetic benchmarks. Moreover, although the dataflow approximation technique results in run times that are comparable to those seen in the synthetic benchmarks, the exact analysis and the simulation-based approach are, respectively, roughly 10 000 and 100 000 times slower than those in [11]. The reason for the increased run time is a large amount of low bandwidth connections with long periods.

7.1.4 Set-top box SoC: Our second design example is a set-top box with four different use-cases, all having hot-spots around three SDRAM ports and 100 to 250 connections. These connections deliver a total bandwidth of 1–2 Gbps to 75 ports distributed across 25 IP modules. With more than 500 connections to be analysed, this constitutes the largest example. The slot table size is also larger, with 67 slots, to accommodate a wide variety of bandwidth requirements.

As with the mobile phone SoC, we look at the total buffer capacity required across the use-cases and the run time needed to compute the buffer sizes. The results are presented in Table 4, except for the simulation-based ones that had not even finished the first use-case after running for 24 h. Simulating one least common multiple for every possible scheduling, interleaving is therefore often impractical for a design of this size. For the dataflow analysis, the approximation algorithm is almost as fast as

Table 4 Buffer sizes for set-top box system

Algorithm	Run time, s	Tot. buffers (words)	Impr., %
analytical [11]	6.10	9190	ref
simulated [15]	–	–	–
dataflow approx. [17]	6.87	7818	15
dataflow exact [19]	15 229	7170	22

[11], since the ratio computation and file I/O are far larger than for the previous examples. It should also be noted that this time also necessitated four invocations of the exact algorithm, as the heuristic failed to find a solution. Exclusive use of the exact algorithm, on the other hand, is considerably slower, again because of very large solution space. The slot table size for this design is also fairly large, leading to smaller discretisation effects, and this benefits the analytical algorithm that uses less conservative bounds. At the same time, the dataflow analysis does not have the opportunity to reduce the buffering requirements by exploiting lower response times, and we see a relative improvement of only 22% for the exact analysis.

Although 7170 words of buffering in the NIs might seem costly, it should be noted that most of this buffering is located close to the memory controller and its three ports. It is, thus, possible to use a few large dual-ported SRAMs rather than dedicated FIFOs. Thus, the roughly 24 kbits worth of buffering occupies only 0.2 mm^2 [38] in a 65 nm CMOS technology.

7.2 Application case study

In this section, we demonstrate how to model a complete application as a CSDF graph and, thus, enable verification of the end-to-end temporal behaviour and computation of NoC buffer sizes. For more examples, we refer the reader to for example, [24, 25].

The audio post-processing application, shown in Fig. 7*a*, comprises three tasks: first, the source analogue to digital conversion (ADC), periodically producing signed 16-bit pulse-code-modulated stereo samples; secondly, the actual filter task, executed on a statically scheduled VLIW

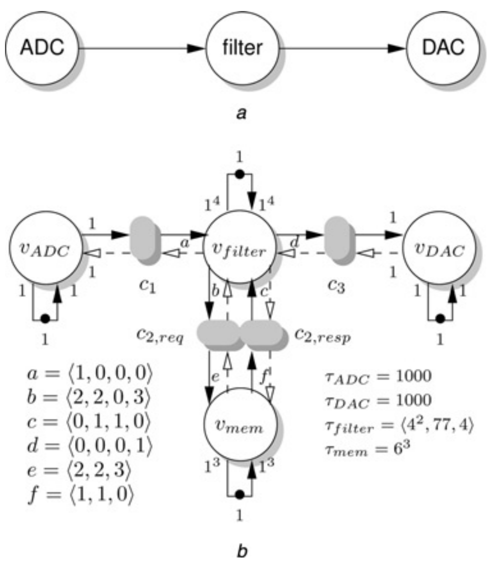
ADC
filter
DAC
a
v_{ADC}
v_{filter}
v_{DAC}
v_{mem}
c_1
c_3
$c_{2,req}$
$c_{2,resp}$
$a = \langle 1,0,0,0 \rangle$
$b = \langle 2,2,0,3 \rangle$
$c = \langle 0,1,1,0 \rangle$
$d = \langle 0,0,0,1 \rangle$
$e = \langle 2,2,3 \rangle$
$f = \langle 1,1,0 \rangle$
$\tau_{ADC} = 1000$
$\tau_{DAC} = 1000$
$\tau_{filter} = \langle 4^2, 77, 4 \rangle$
$\tau_{mem} = 6^3$
b

Figure 7 *Task graph and dataflow model of the audio post-processing filter*

a Task graph
b Dataflow model

without caches; and thirdly, the digital to analogue conversion (DAC), which acts as a periodic sink. Both the ADC and DAC have a sampling frequency of 48 kHz, and the NoC, processor and memory run at 48 MHz on an FPGA instance of the system. The filter task receives input samples from the ADC via the NoC and adds a two-tap reverberation and echo effect by mixing in past samples. The output is then sent both to the DAC and stored in the background memory for future mixing with the input. The filter application is firm real-time, as the failure to consume and produce samples in 48 kHz leads to noticeable clicks in the output.

The dataflow model of the filter application is shown in Fig. 7*b*, with time indicated in network cycles, that is, at 48 MHz. The ADC and DAC are modelled by single actors that have a response time of 1000 cycles. The filter application requires three connections, and a total of four channels, as indicated by the grey boxes in Fig. 7*b*. The processor uses FIFO streaming for the communication with the ADC and DAC, and these connections only have request channels. Shared memory communication is used for reading and writing reference samples in the background memory, using both a request and response channel, over which both reads and writes are sent. Every grey box corresponds to any one of the channel models in Fig. 4. The filter, together with its associated NI shell, is modelled by v_{filter}. Finally, the memory with its NI shell is modelled by v_{mem}, using the technique proposed in [39]. A detailed discussion of the last two actors follows.

The filter actor v_{filter} has four phases, corresponding to the two read operations, the actual calculation, and the writing of the output. The rates for the NoC channels are indicated by *a*, *b*, *c* and *d* in Fig. 7*b*. Note that the rate is the same for both the consumption (incoming edge) and production (outgoing edge). In the first phase, an input sample is read from the ADC, and a read request is sent to the memory ($h_{req,r}$). In the second phase, a new read request is sent, and the read response from the first phase returns one reference sample from memory ($1 + h_{resp,r}$). In the third phase, the read response from the second read returns. In the fourth and last phase, the output sample is written to memory ($1 + h_{req,w}$) and sent to the DAC. The response times of the different phases are determined by the analysis of the program flow [40]. The first two phases take four cycles each, partly because of the shell. Thereafter, 77 processor cycles are spent performing arithmetic operations and accessing local memory. Finally, another four cycles are spent writing the output.

The memory actor v_{mem} models an SRAM with a fixed access time of six cycles. The shell spends two cycles reassembling the bus transaction, and four cycles are for the pipelining of the controller and the TDM arbiter (more elaborate arbitration schemes are possible [36]). The production and consumption rates of the request and response channels are indicated by *e* and *f* in Fig. 7*b*. Reads and writes from the filter to the memory share the same connection. This is a complication, as the production and consumption rates of v_{mem} are different for the two types of operations (and also depends on the burst size). Therefore we model the memory with three phases, corresponding to the three accesses of the filter (read, read and write). This is only possible as the order of the operations is fixed, and not a generally applicable technique. There are, however, more elaborate dataflow models that enable variable-rate model [33], but this is outside the scope of this paper. In the first and second phases, a read request is received ($h_{req,r}$), and a reference sample is sent back to the filter ($1 + h_{resp,r}$). In the third phase, a write request is received ($1 + h_{req,w}$), without sending any response back (as $h_{resp_w} = 0$). The response time for all the phases is six cycles, as previously discussed.

The filter model in Fig. 7*b*, together with a channel model in Fig. 4 can now be used to determine an NoC configuration such that the ADC and DAC are guaranteed to produce and consume samples in 48 kHz. This simple example application, being just a pipeline with three tasks, demonstrates: (1) how the NoC channel model is included in a dataflow model of the application to include the temporal behaviour of inter-task communication, (2) how the mapping to an architecture gives rise to cyclic dependencies even though the application is a pipeline, (3) how coupling between NoC channels, for example, the request and response channels to memory, is taken into account in the dataflow model and (4) that more elaborate dataflow models, with variable rate, are needed also for very simple applications.

8 Conclusions and future work

A growing number of applications, often with real-time requirements, are integrated on the same SoC, in the form of hardware and software IP. NoC offer guaranteed throughput and latency to the communication between IPs. However, the guarantees only cover the router network and NI. To give performance guarantees on the application level, the buffers in the NIs must be sized according to the application behaviour. If these buffers are not sufficiently large, the performance requirements cannot be guaranteed. At the same time, the size must be kept at a minimum as the buffers are a major contributor to NoC power and silicon area.

In this work, we show how to construct a dataflow graph that conservatively models an NoC connection. The only requirements are that the NoC uses blocking flow control and that a connection is a latency-rate server. We show that an Æthereal guaranteed-throughput connection is a latency-rate server, and present a detailed model of such a connection. The proposed model is evaluated quantitatively by comparing with existing buffer-sizing approaches over a range of SoC designs. Buffer sizes are determined with a run time comparable to existing analytical methods, and results comparable to exhaustive simulation. In contrast to existing buffer-sizing methods that rely on coarse linear bounds or exhaustive simulation, the presented dataflow model allows for a more expressive application model, capturing cyclic dependencies and variation in execution times. Given that the application itself has a dataflow model, the presented channel model can be inserted to include the temporal behaviour of the communication between tasks. This enables the computation of NoC configurations that satisfy end-to-end temporal constraints.

9 References

[1] DUTTA S., JENSEN R., RIECKMANN A.: 'Viper: a multiprocessor SOC for advanced set-top box and digital TV systems', *IEEE Des. Test Comput.*, 2001, **18**, (5), pp. 21–31

[2] RUTTEN M., POL E.-J., VAN EIJNDHOVEN J., WALTERS K., ESSINK G.: 'Dynamic reconfiguration of streaming graphs on a heterogeneous multiprocessor architecture'. IS&T/SPIE Electron. Imag., 2005, vol. 5683

[3] ROWEN C., LEIBSON S.: 'Engineering the complex SOC: fast, flexible design with configurable processors' (Prentice-Hall PTR, 2004)

[4] BUTTAZO G.C.: 'Hard real-time computing systems: predictable scheduling algorithms and applications' (Kluwer Publishers, 1977)

[5] MOREIRA O., VALENTE F., BEKOOIJ M.: 'Scheduling multiple independent hard-real-time jobs on a heterogeneous multiprocessor'. Proc. EMSOFT, 2007

[6] HANSSON A., COENEN M., GOOSSENS K.: 'Undisrupted quality-of-service during reconfiguration of multiple applications in networks on chip'. Proc. DATE, 2007

[7] LIANG J., SWAMINATHAN S., TESSIER R.: 'aSOC: a scalable, single-chip communications architecture'. Proc. PACT, 2000, pp. 37–46

[8] JANTSCH A.: 'Models of computation for networks on chip'. Proc. ACSD, 2006

[9] GOOSSENS K., DIELISSEN J., RĂDULESCU A.: 'The Æthereal network on chip: concepts, architectures, and implementations', *IEEE Des. Test Comput.*, 2005, **22**, (5), pp. 21–31

[10] BJERREGAARD T., SPARSØ J.: 'A router architecture for connection-oriented service guarantees in the MANGO clockless network-on-chip'. Proc. DATE, 2005, pp. 1226–1231

[11] GANGWAL O., RADULESCU A., GOOSSENS K., PESTANA S., RIJPKEMA E.: 'Building predictable systems on chip: an analysis of guaranteed communication in the Æthereal network on chip', 'Dynamic and robust streaming in and between connected consumer-electronics devices' (Kluwer, 2005)

[12] RADULESCU A., DIELISSEN J., GOOSSENS K., RIJPKEMA E., WIELAGE P.: 'An efficient on-chip network interface offering guaranteed services, shared-memory abstraction, and flexible network programming', *IEEE Trans. CAD Int. Circuits. Syst.*, 2005, **24**, (1), pp. 4–17

[13] BJERREGAARD T., MAHADEVAN S., OLSEN R.G., SPARSØ J.: 'An OCP compliant network adapter for GALS-based SoC design using the MANGO network-on-chip'. Proc. SOC, 2005, pp. 171–174

[14] HANSSON A., WIGGERS M., MOONEN A., GOOSSENS K., BEKOOIJ M.: 'Applying dataflow analysis to dimension buffers for guaranteed performance in networks on chip'. Proc. NOCS, 2008

[15] COENEN M., MURALI S., RĂDULESCU A., GOOSSENS K., DE MICHELI G.: 'A buffer-sizing algorithm for networks on chip using TDMA and credit-based end-to-end flow control'. Proc. CODES + ISSS, 2006

[16] WIGGERS M., BEKOOIJ M., JANSESN P., SMIT G.: 'Efficient computation of buffer capacities for cyclo-static real-time systems with back-pressure'. Proc. RTAS, 2007

[17] BEKOOIJ M.J.G., SMIT G.J.M.: 'Efficient computation of buffer capacities for cyclo-static dataflow graphs'. Proc. DAC, 2007

[18] STUIJK S., GEILEN M., BASTEN T.: 'Throughput-buffering trade-off exploration for cyclo-static and synchronous dataflow graphs', *IEEE Trans. Comput.*, 2008, **57**, (10), pp. 1331–1345

[19] DASDAN A.: 'Experimental analysis of the fastest optimum cycle ratio and mean algorithms', *ACM TODAES*, 2004, **9**, (4), pp. 385–418

[20] KAVALDJIEV N.: 'A run-time reconfigurable network-on-chip for streaming dsp applications'. PhD dissertation, University of Twente, 2006

[21] WEBER W.-D., CHOU J., SWARBRICK I., WINGARD D.: 'A quality-of-service mechanism for interconnection networks in system-on-chips'. Proc. DATE, 2005

[22] GOOSSENS K., DIELISSEN J., GANGWAL O.P., GONZÁLEZ PESTANA S., RĂDULESCU A., RIJPKEMA E.: 'A design flow for application-specific networks on chip with guaranteed performance to accelerate SOC design and verification'. Proc. DATE, 2005

[23] HANSSON A., GOOSSENS K., RĂDULESCU A.: 'A unified approach to constrained mapping and routing on network-on-chip architectures'. Proc. CODES + ISSS, 2005

[24] HOLZENSPIES P., HURINK J., KUPER J., SMIT G.: 'Run-time spatial mapping of streaming applications to a heterogeneous multi-processor system-on-chip MPSoC'. Proc. DATE, 2008

[25] MOONEN A.: 'Modelling and simulation of guaranteed throughput channels of a hard real-time multiprocessor system'. Master's thesis, Eindhoven University of Technology, 2004

[26] MURALI S., DE MICHELI G.: 'An application-specific design methodology for STbus crossbar generation'. Proc. DATE, 2005

[27] HU J., MARCULESCU R.: 'Application-specific buffer space allocation for networks-on-chip router design'. Proc. ICCAD, 2004

[28] SRIRAM S., BHATTACHARYYA S.: 'Embedded multiprocessors: scheduling and synchronization' (CRC Press, 2000)

[29] BILSEN G., ENGELS M., LAUWEREINS R., PEPERSTRAETE J.: 'Cyclo-static dataflow', *IEEE Trans. Signal. Process.*, 1996, **44**, (2), pp. 397–408

[30] AMBA AXI Protocol Specification: 'ARM Limited', 2003

[31] OCP Specification 2.2: 'OCP International Partnership', 2007

[32] WINGARD D.: 'Socket-based design using decoupled interconnects', in NURMI J., TENHUNEN H., ISOAHO J., JANTSCH A. (EDS.): 'Interconnect-centric design for SoC and NoC' (Kluwer, 2004)

[33] WIGGERS M.H., BEKOOIJ M.J., SMIT G.J.: 'Buffer capacity computation for throughput constrained streaming applications with data-dependent inter-task communication'. Proc. RTAS, 2008

[34] BAMBHA N., KIANZAD V., KHANDELIA M., BHATTACHARYYA S.: 'Intermediate representations for design automation of multiprocessor dsp systems', *Des. Autom. Embedded Syst.*, 2002, **7**, (4), pp. 307–323

[35] MOREIRA O., BEKOOIJ M.: 'Self-timed scheduling analysis for real-time applications', *EURASIP J. Adv. Signal Process.*, 2007

[36] WIGGERS M., BEKOOIJ M., SMIT G.: 'Modelling run-time arbitration by latency-rate servers in dataflow graphs'. Proc. SCOPES, 2007

[37] STILIADIS D., VARMA A.: 'Latency-rate servers: a general model for analysis of traffic scheduling algorithms', *IEEE/ACM Trans. Netw.*, 1998, **6**, (5), pp. 611–624

[38] OHBAYASHI S., YABUUCHI M., NII K., *ET AL.*: 'A 65-nm SoC embedded 6T-SRAM designed for manufacturability with read and write operation stabilizing circuits', *IEEE J. Solid-State Circuits*, 2007, **42**, (4), pp. 820–829

[39] STUIJK S., BASTEN T., MESMAN B., GEILEN M.: 'Predictable embedding of large data structures in multiprocessor networks-on-chip'. Proc. Euromicro Symp. Digital System Design, 2005

[40] CHEN K., MALIK S., AUGUST D.: 'Retargetable static timing analysis for embedded software'. Int. Symp. System Synthesis (ISSS), 2001, pp. 39–44

doi: 10.1049/iet-cdt.2008.0093