Synchronous dataflow graph (SDFG) modeling and performance analysis of multiprocessor NoC based system on chip (SoC)

Hassoun, M.

Award date:
2009

TECHNISCHE UNIVERSITEIT EINDHOVEN

Department of Mathematics and Computer Science

Master's thesis

Synchronous Dataflow Graph (SDFG) Modeling and Performance Analysis of Multiprocessor NoC Based System on Chip (SoC)

by
Maissa Hassoun

Committee:
prof. dr. ir. J. J. Lukkien (TU/e)
dr.ir. M. C. W. Geilen (TU/e)
ir. J. Boonstra (NXP Semiconductors)

Eindhoven / High Tech Campus
December, 2008

Abstract

Design of *systems-on-chip* (SoCs) for modern consumer electronic devices is getting more and more complexer with the advancing semiconductors technology. Many software and hardware intellectual property components (IP cores) are integrated on a single sub-micron chip. Communication between these large numbers of cores in modern SoC is realized by *Network-on-Chip* (NoC). NoC became the paradigm for designing scalable SoC and is common in the implementation of SoC for many application areas like real time multimedia applications. These applications share resources on-chip and often share off-chip memory with real-time requirements, therefore the NoC needs to provide performance guarantees (throughput and latency) to fulfil the performance constraints of these real-time applications. Tooling for analyzing the performance of NoC based SoC are still in the area of research and development. In this thesis we introduce a method to model a NoC based SoC for a multimedia application by constructing a *Synchronous Dataflow Graph* (SDFG) model. We analyze the throughput of the constructed model, make trade-off between throughput and buffer size, and compute latency. We apply the method on a video decoder H.263 and analyse its performance. We improve the performance results further by applying the Latency-Rate (LR) analysis to the shared memory controller. We show the impact of of the arbitration schemes on the latency computation and modeling in general. Our method in modeling and analyzing the performance of NoC based SoC is independent of the scheduling algorithm and platform architecture.

General Terms:
Modeling, Performance analysis

Key words:
SoC, NoC, SDF, WCET, Latency, Throughput, Arbitration

Acknowledgments

I would like to take the opportunity to thank all who have supported me with my master thesis at the TU/e and NXP Semiconductors, especially to Joep Boonstra for his supervision and feedback during the progress meetings, to my colleagues Henk Hamoen and Joost Laarakkers for offering me this assignment and for their support.

Many thanks to Sander Stuik for his input in the initial phase of this master thesis and to Johan Lukkien for taking a place in my graduation commission and for his feedback and suggestions on the presented work.

My appreciation and gratitude to my gradation supervisor at the TU/e, Marc Geilen, for the opportunity to do my master thesis under his guidance. His suggestions, feedback and support have been of great value throughout the graduation period and writing of this thesis.

Finally, warmest thank to my family abroad for their moral support, and to my husband and best friend Kazik for being my greatest support during those years of study. Without his encouragement and confidence in me I would not have been able to accomplish this work.

Contents

Chapter 1

Introduction

NXP Semiconductors (former Philips Semiconductors) is developing a state of the art network-on-chip known as: Universal Network Interconnect on-Chip (U-NIC) which will replace the legacy interconnect (dedicated point-to-point signal wires, shared buses, or segmented buses with bridges) in the design of future SoC for multimedia and automotive applications.
U-NIC needs to provide performance guarantees for the real-time applications that will be integrated in the future SoC.

1.1 Purpose of the assignment

Systems-on-chip (SoC) are composed of multiple processing elements or IP cores (e.g. in a multimedia SoC the processing elements represents the application's code segments) that interconnect through communication elements (e.g. network-on-chip). A challenging aspect in the design of SoC is to guarantee that the SoC fulfills the application's hard real-time requirements and that sufficient bandwidth is offered by the interconnect (NoC) to each of the processing elements at all times as required. In order to configure the network and evaluate the performance of the SoC, there is a need for an adequate tooling to perform these tasks at the design time. The purpose of this assignment is to provide a method to configure and analyse the performance of multimedia applications when mapped to NoC-based SoC which helps the SoC designers to map, schedule and analyse the performance of a multimedia NoC based SoC at the design time.

The main contribution in this report is a method to construct an SDFG model for U-NIC based multimedia SoC (U-NIC interconnect, a multimedia application mapped to U-NIC platform and an external memory controller) and analyzing the performance of the SoC based on the SDFG Design Flow and Tooling SDF^3 [26]. Our method fits in a NoC design flow which consists of iterative steps to configure and analyse the performance of a network based on a given application, network topology and performance constraints. The steps will be repeated until the performance evaluation fulfills the applications constraints. Our method provides worst-case performance guarantees since it requires that upper bounds

on the latency (Worst Case Execution Times) are known upfront for each of the processing elements as well as the the network components and the shared memory.

1.2 Related work

Available design flows and tools such as SDFG-based SDF^3 (SDFG For Free) [26] and UMARS [19] provide solutions for NoC generation and performance analysis at design time. UMARS requires that time-division-multiple-access (TDMA) time-slots are allocated on the network to guarantee that the application's constraints are met. The same perquisite applies to the SDF^3 design flow. If the network architecture can fulfil this requirement then either of these design flows can be used with minor modifications. However, this limiting factor makes these design flow unsuited for the U-NIC based SoC design since U-NIC architecture that is studied in this report does not implement TDMA time-slot allocation. Our method doesn't put any limitations on the network architecture, however it requires that the latency at the network elements are known. This will be discussed in greater details in the next chapters.

1.3 Structure of this thesis

This report is organized as follows: chapter 2 provides a general introduction to NXP U-NIC interconnect and arbitration schemes. Chapter 3 describes the structure of the SDFG and provides related definitions and notions. Chapter 4 gives an overview of the SDF^3 design flow, tooling and analysis algorithms with illustration on the differences between the proposed platform in [26] and U-NIC platform. Chapter 5 introduces the method for modeling a SoC in SDFG. Chapter 6 provides details on the performed experiment on a multimedia application (H.263) mapped onto U-NIC based SoC. Chapter 7 gives the results of the performance analyses applied to the case study. Chapter 8 proposes a memory controller model based on a combination of network calculus and dataflow analyses. Chapter 9 proposes an improved model of the H.263 case study and performance results. Chapter 10 provides summary and conclusions with recommendations for future work. Chapter 11 provides a guide for tool installation and running the analysis algorithms.

Chapter 2

Universal Network Interconnect on-Chip (U-NIC)

In this chapter we provide a general overview of U-NIC interconnect architecture and components. Additionally, we give some preliminaries on network-on-chip (NoC) arbitration and performance analysis that are relevant to the scope of this assignment.

U-NIC is a high performance on-chip interconnect that will gradually replace legacy interconnects in future NXP systems-on-chip (SoCs). Figure 2.1 shows U-NIC topology that consists of network interfaces (NIs) on the boundary of the network and switches within the network. There are two types of network interfaces: *target* (tNI) which allows the master IP (processing element) to initiate a transaction request on the network; and *initiator* (iNI) which allows the slave IP, typically an off-chip shared memory controller, to respond to the transactions on the network. U-NIC supports connection of IP devices with AXI [7], AHB [4], DTL [5], MTL [6] interfaces.
U-NIC supports 32-bit memory-mapped data transactions. If the transmitted data is not 32-bit wide, data-width conversion to 32-bit (U-NIC word) takes place at the tNI, similarly conversion from 32-bit to different data-width occurs at the iNI.

U-NIC is a layered, packet based protocol, based on an OSI stack model [31]. The basic idea of layering is that each layer adds value to services provided by the set of lower layers in such a way that the highest layer is offered the set of services needed to run distributed applications according to the OSI model. On the top of the stack U-NIC has the application interfaces like AHB, DTL and AXI. On the bottom the stack a component connects to another component through a U-NIC link as shown in figure 2.2.
When transmitting, while going down the stack each layer encapsulates the transmitted packet of the upper layer and forwards the encapsulated packet to the next layer. When receiving, while going up the stack each layer strips the encapsulation information and forwards the stripped packet. Each layer can add new packets for communication between same level layers. At the *Application* layer, U-NIC network interfaces provide network access for master and slave IPs. At the *Transport* layer U-NIC network interfaces take

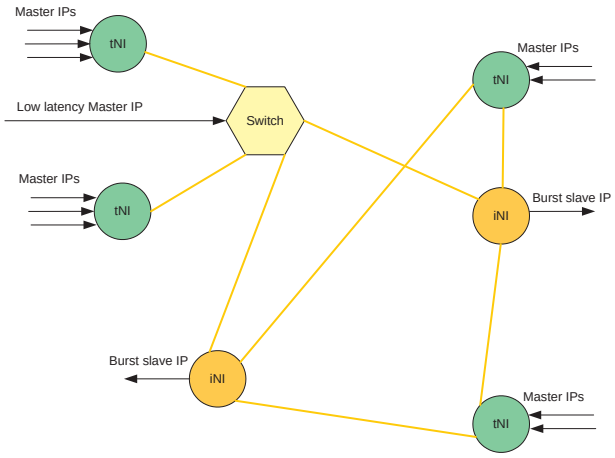

Figure 2.1: U-NIC Network Topology

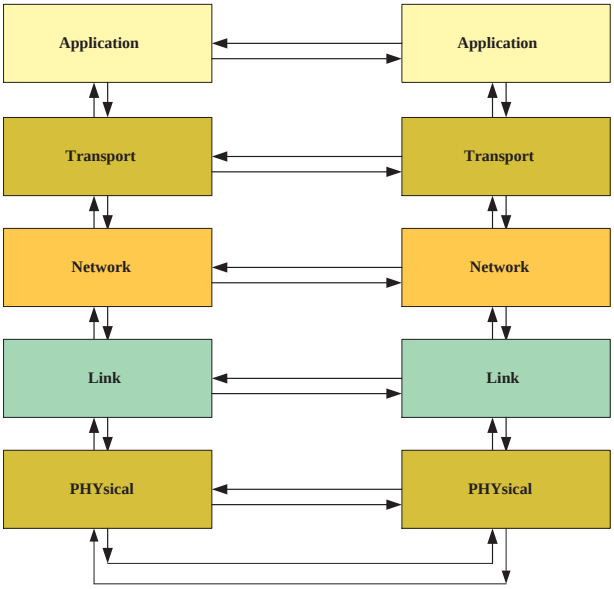

Figure 2.2: U-NIC OSI Model

packets from different resources, buffering the packets, mapping the packets to virtual channels and arbitrate between multiple sources that are mapped to the same virtual channel and forward packets to the data-link. At the *Network* layers, U-NIC switches take care of packets routing through the network where the route through the network is determined at the source of the packet (source routing). At the *Data Link* layer, U-NIC links take care of robust link-level data transfers, flow control and data interleaving. At

the *Physical* layer, U-NIC PHY links convey the data streams through the network at the electrical level by encoding it to PHY symbols so that it can be transmitted over a hardware transmission medium.

2.1 Performance

The performance of NoC based SoCs is known to be the throughput and latency that a network-on-chip can offer as guarantees to fulfill the requirements of the applications that are mapped on these systems. We distinguish between two types of performances, worst-case and average-case. In worst-case performance, guarantees are provided for real-time critical applications where the application tasks should be performed within strict deadlines, while in average-case performance, best-effort for non-critical applications is given. For the worst-case, it is possible to analyse the performance analytically, however for the average-case a probability theory can help in analyzing the performance since it is difficult to predict what is the average traffic [29]. Alternatively, simulation can provide average-case performance analysis but it gives no performance guarantees.

2.2 Quality of service

An on-chip communication network should provide deterministic bounds on delays and throughput for communication among pairs of cores on the chip. This is generally achieved by special designs of network routers which have capabilities of reserving channels or bandwidth for certain traffic and/or by allowing packets transmission with different priorities [11].

U-NIC is a virtual channel (VC) based network. Per physical link there are max 16 virtual channels, which share the link in time-division-multiple-access fashion.
Virtual channels can lead to an increase in the throughput of the network and avoid the occurrence of deadlock. This is because a blocked packet in one virtual channel does not affect a packet progressing on another virtual channel [11]. However, the more virtual channels the more buffers are required and accordingly the more silicon area which results in increasing the SoC price and of course the power consumption.

2.3 Arbitration

Systems-on-chip (SoCs) contain many applications with real-time requirements. Resources like the interconnect and memory are often shared between the applications to reduce the costs of a SoC. However, sharing the resources between the applications means that the processing elements representing the applications will compete to use these resources therefore arbitration is required to organize access to the shared resources in the SoC. For a NoC, arbitration influences the latency of the network elements. Each of the arbitration points contributes to the delay on the message passing through the network. Latency is an important parameter for the performance analysis.

In our SDF modeling method, we provide hard guarantees for real-time applications, therefore the knowledge of the worst-case execution time is essential for providing an accurate performance analysis of the system. Arbitration have impact on computing the worst-case execution times. We will briefly present some of the most common arbitration/scheduling schemes and show the impact on the performance analysis.

2.3.1 Time-division-multiple-access (TDMA) scheduling

A time-division-multiple-access (TDMA) arbiter uses a time-wheel that rotates periodically and consists of time-slots that are fractions of the time-wheel. A task (code segment of an application) is assigned slot(s) in the time-wheel that would allow the task to execute when the scheduler is within the allocated time slot(s) and if needed multiple slots can be allocated to the task to obtain more bandwidth (weighted TDMA). TDMA provides worst-case guarantees on the maximum latency [12]. Per TDMA wheel rotation a predefined number of words (network words) can be sent over the network, the latency of a request due to TDMA arbitration depends on the number of TDMA wheel rotations that are necessary for processing a request. However, TDMA provides conservative upper bounds on the latency for average-case traffic because even when part of the time wheel is not allocated, a scheduled task has to wait for its allocated time-slot to execute and cannot make use of the free time-slots that might belong to an idle task. We can compute the time required by a task (read or write request) on a TDMA scheduler if we know the size of the request, the TDMA time-wheel and the size of the TDMA time-slot. The number of time-slots required by a task i to submit its read or write request can be computed as follows:

$$N_{tsi} = \lceil \frac{S_i(byte)}{S_{ts}(byte)} \rceil$$

Where N_{tsi} is the number of time-slots required by request i, S_i is the size of request i in bytes and S_{ts} is the size of the time-slot in bytes. The number of time-slots required by request i is rounded up incase the size of the request is not a product of the time-slot size.

The latency of request i depends on its time-slots allocation on the TDMA time wheel. If the needed time-slots per request is a multiple of the allocated slots in the time-wheel, then the worst-case latency can be computed as follows:

$$L_i = (\frac{S_{tw}(slots)}{AL_{tsi}(slots)} \times N_{tsi}) \times N_{cc/ts}$$

Where L_i is the latency of request i, S_{tw} is the size of TDMA time-wheel in slots, AL_{tsi} is the allocated time-slots for task i and $N_{cc/ts}$ is the number of cc per time-slot.

A more general formula for computing the worst-case latency at a scheduler when the needed time-slots per request is not a multiple of the allocated slots in the time-wheel, independent of the slot distribution across the time-wheel is as follows:

$$L_i = (\lceil \frac{N_{tsi}(slots)}{AL_{tsi}(slots)} \rceil \times S_{tw} - (\lceil \frac{N_{tsi}(slots)}{AL_{tsi}(slots)} \rceil \times AL_{tsi} - N_{tsi})) \times N_{cc/ts}$$

If we have no knowledge about the slots distribution in a time-wheel, we consider the slots to be located at the end of the time-wheel, in this way we guarantee the worst-case latency, since in the worst-case the last slot in the time-wheel is needed. However, if in the last time-wheel cycle less slots than the allocated ones are needed by the request, then these un-used slots (which reside at the end of the time-wheel) are excluded from the worst-case latency computation as shown in the second term of the formula.

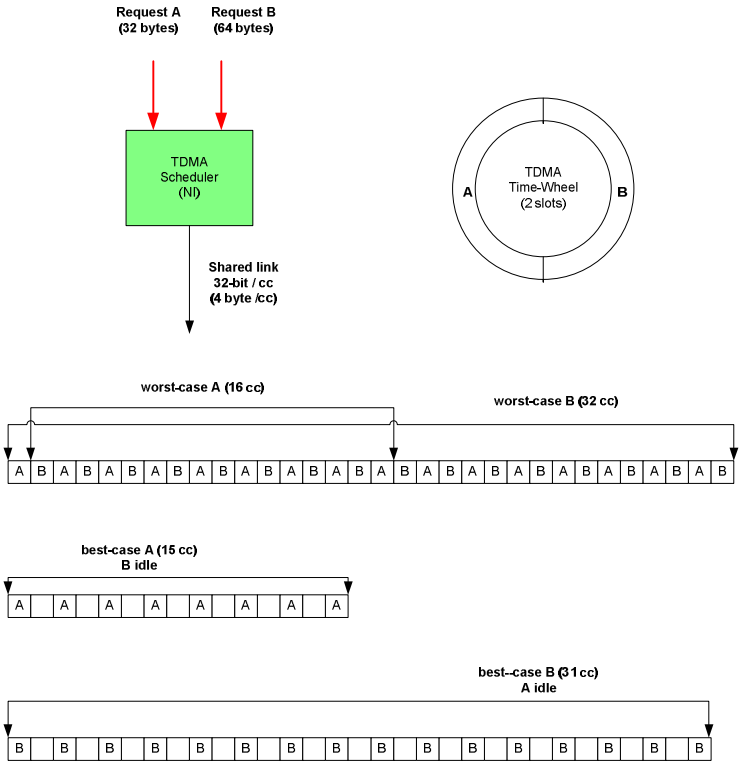

Figure 2.3: Example 2 requests on TDMA scheduler

Figure 2.3 provides an example of two concurrent requests A and B that are scheduled on a TDMA scheduler with request size 32 byte and 64 byte respectively. We consider a time-wheel with 2 time-slots, each time-slot has the size of 4 bytes (which is in this case the link bandwidth that the 2 requests want to share), each request is allocated 1 time-slot in the time-wheel, the wheel turns every clock cycle 1 time-slot. Table 2.1 shows the worst-case latency of request A and B computed according to the above formulas. If we want to compute the best-case latency where for example request A is idle, we notice that the best-case latency is almost equal (depends on the arrival of task B) to the worst-case latency since the reserved time-slot for request A will not be available for the use by request B. The same applies if request B is idle, request A will still have a

Table 2.1: **Latency of example requests on TDMA scheduler**

Task	N_{ts} (slots)	AL_{ts} (slots)	S_{tw}(slots)	WCase latency (cc)	BCase latency (cc)
A	8	1	2	16	15
B	16	1	2	32	31

best-case latency that is almost equal to the worst-case latency as indicated in Figure 2.3. The average-case latency is between the worst-case and the best-case, which is for TDMA almost the value of the worst-case latency. Table 2.1 provides the computed worst-case and best-case for example requests A and B when scheduled on TDMA scheduler.

2.3.2 Round-robin (RR) scheduling

Round-robin is known to be the simplest scheduling algorithm where equal time-slots are assigned in order to the tasks that share the same resource, without priority assignment to the tasks. Round-robin scheduling is both simple and easy to implement, and starvation-free. In round-robin the highest performance is guaranteed when the tasks have uniform size. If the tasks have variable sizes, then round-robin is not desired since a task with large size would be favored over other tasks [1]. For tasks with varying sizes, time-slots are assigned so that a task can be interrupted in one round if it uses the allocated bandwidth and it continues in the next round.
In round-robin the average-case latency depends on the number of the 'active' tasks that are scheduled on the same resource and the tasks arrivals in the time wheel. If a task is 'inactive'/idle, its allocated slot(s) can be used by the other requests which are scheduled on the same scheduler. The average-case performance in round-robin is better than that of TDMA scheduler, since in an average-case performance the available time-slots for a request might exceed its original allocation.
The worst-case latency for a request scheduled on round-robin is identical to that of TDMA scheduler, because in the worst-case performance all the scheduled requests on round-robin scheduler make use of their allocated time-slots (no idle requests). The formulas used in computing the latency for TDMA scheduler apply to round-robin.

Figure 2.4 provides an example of the previous requests A (size 32 bytes) and B (size 64 bytes) when scheduled on round-robin scheduler. For 2 requests there are 2 time-slots that are allocated fairly in order for request A and B.

The best-case latency of task A occurs when task B is idle = 8cc. Similarly, the best-case latency of task B is when task A idle = 16cc. Table 2.2 provides the computed worst-case and best-case of the example requests A and B when scheduled on round-robin scheduler. The average-case latency is between the worst-case and best-case latency.

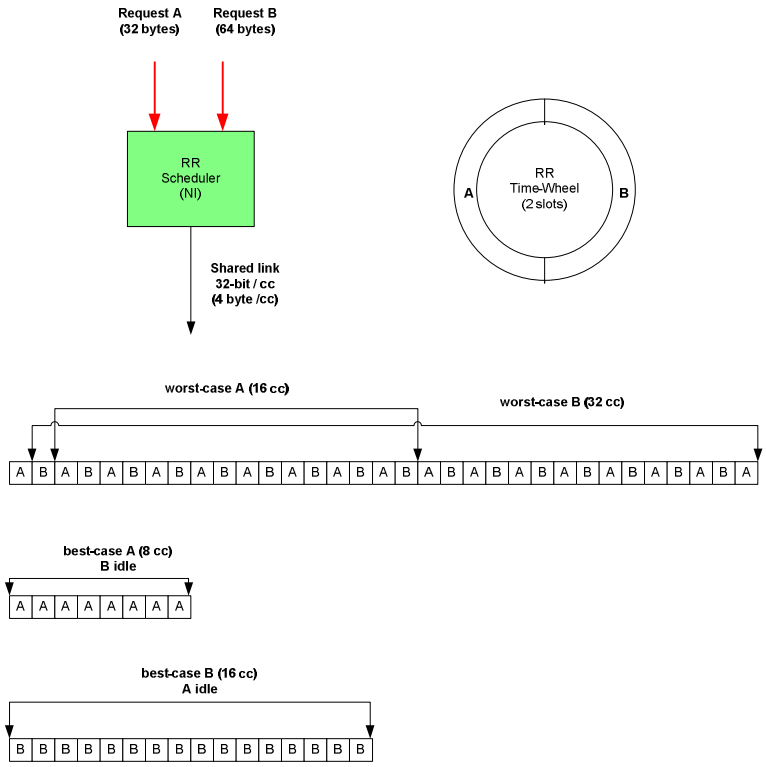

Figure 2.4: Example 2 requests on round-robin scheduler

2.3.3 Weighted round-robin (WRR) scheduling

Weighted round-robin scheduling is used for traffic of the same priority, with weights proportional to the bandwidth allocations of the tasks. The weights determine the number of bytes that a task is allowed to send at each round. A variation of weighted round-robin is that if a task sends more than its allocated bandwidth, the same task receives less bandwidth in the next round [14]. The formulas used in computing the latency for TDMA scheduler apply to weighted round-robin.

Figure 2.5 provides and example of the same 2 requests A and B on weighted round-robin scheduler whith 3 time-slots with allocation of 1 time-slot for taks A and 2 time-slots for

Table 2.2: **Latency of example requests on RR scheduler**

Task	N_{ts} (slots)	AL_{ts} (slots)	S_{tw}(slots)	WCase latency (cc)	BCase latency (cc)
A	8	1	2	16	8
B	16	1	2	32	16

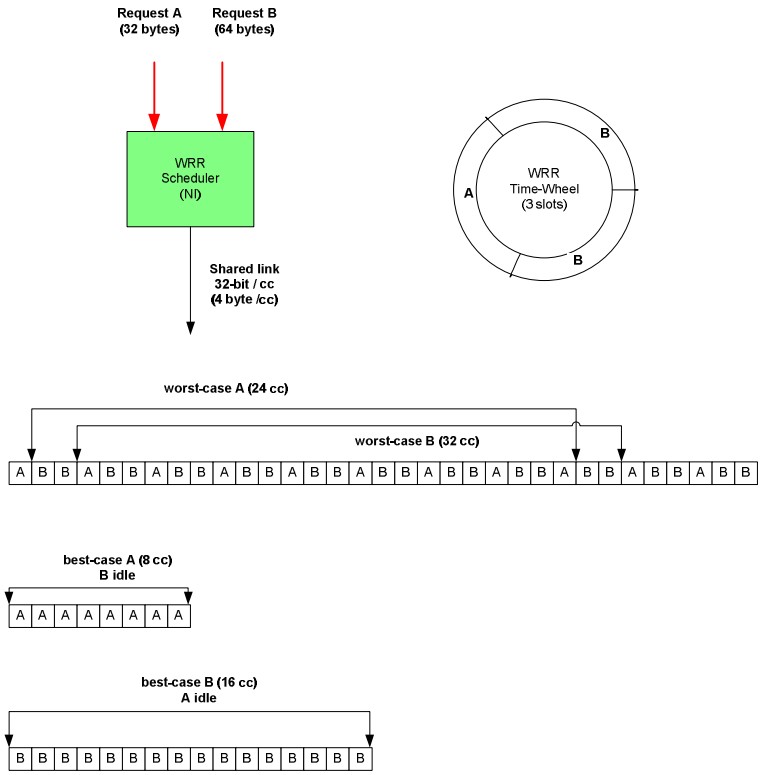

Figure 2.5: Example 2 requests on weighted round-robin scheduler

task B. Table 2.3 provides the worst-case and best-case latency for tasks A and B where the best-case latency computation of task A we assume that task B is idle and vice versa. The value of the average-case latency is between the worst-case and best-case as explained above.

2.3.4 Priority-based scheduling

In a priority-based scheduling each task is given a predefined priority to share the network resources. Tasks with the highest priority have the least latency. Often, tasks with stringent real-time requirements receive the highest priority, while tasks with less critical

Table 2.3: **Latency of example requests on WRR scheduler**

Task	N_{ts} (slots)	AL_{ts} (slots)	S_{tw}(slots)	WCase latency (cc)	BCase latency (cc)
A	8	1	3	24	8
B	16	2	3	24	16

real-time requirements get lower priorities. Accordingly, upper bounds on latency can be put for high priority tasks, while the latency computation for low priority tasks is harder because of the dependency on the arrival of higher priority traffic. The queueing theory [2] provides means to compute the waiting time of a lower priority traffic based on the arrivals of higher priority traffic and the available requests with higher priority in the queue of the shared resource.
A variation of priority-based scheduling is the rate monotonic algorithm (RMS) which maximizes the schedulability of tasks by assigning fixed priorities to the tasks so that a set of tasks is considered chedulabile if all the tasks in the set meet all their deadlines at all times. The RMS is a simple algorithm based on assigning the priority of each task according to its period, thus the higher priority is given to the tasks with the shortest period. More about the RMS can be found in [24].
[9] presents a static-priority scheduler which provides guaranteed bandwidth allocation and a maximum latency bound that is decoupled from the allocated bandwidth. The computation of the maximum delay of a request with priority is out of the scope of this report, more details can be found in [9].

In U-NIC round-robin scheduling [22] is implemented in several arbitration points, in the switch and in the network interfaces (iNI and tNI) as follows:

- Intra VC arbitration: Between requests on the same VC. Implements weighted round-robin scheme.
- Inter VC arbitration: Between multi VC wanting to transmit packets on the same link. Access will be granted to requests with highest priority.

The choice of round-robin scheduling is mainly to have a better average-case latency compared to TDMA scheduling, while worst-case latency remains comparable to that provided by the TDMA as shown earlier.

2.4 Buffering

In a network-on-chip, throughput guarantees depend on the availability of sufficient buffer space in the network. Buffers are required for the data communication through the network. Buffers can at the same time contribute to reducing the latency as it influences the time needed to process the data (pipelining). Thus increasing the buffer size results in increased throughput and possibly reduced latency in the network. At the same time, increasing the buffer size mean that more memory is required and accordingly increase in the area and power costs. Therefore, the system designer should try to minimize the buffering while meeting the throughput and latency constraints. Throughput-buffering trade-off is studied in [27].

The above applies to U-NIC, since buffers are required in the network components to guarantee the application's desired throughput, however for cost reasons there is a need

to reduce the number of buffers. In every U-NIC component (NI or Switch), there is a FIFO buffer per virtual channel that supports data transfer between these network components. It is also possible that a NI contains 2 SRAM one for requests (TX) and one for responses (RX) since write requests and read response require significant buffer size. iNI resides at the memory side and therefore should have enough buffer capacity to buffer data for all outstanding read and write requests to the memory.

Chapter 3

Synchronous Data Flow Graph (SDFG)

In this chapter we provide formal and informal definitions of a Synchronous Data Flow Graph and the notions that are used in this report. The formal definitions are obtained from the literature [26] [15] [16].

3.1 Informal definition

An SDFG consists of finite sets of *actors* and *channels*. An actor can be seen as a task of an application for example. A *channel* in an SDFG refers to either *dependency edge* or *sequence edge*. The dependency edge between two actors models a data dependency while the sequence edge models the execution order. The data sent on the edges are called *tokens* and the execution of the task is called *firing*. A channel can carry an unbounded number of tokens called *initial* tokens. The initial tokens are required for an actor to start its firing. Communication between two actors is done by sending data via the edge that connects the output of the source actor with the input of the destination actor.

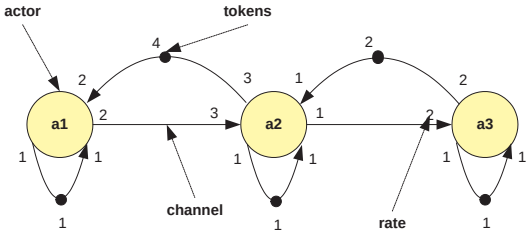

Figure 3.1: Simple SDF graph

When a source actor fires it consumes tokens from its input channel(s), performs computation on these tokens and produces tokens on its output channel(s), the amount of

produced and consumed tokens are fixed and called *rates*. The produced tokens on the output edge(s) of the source actor are consumed by the destination actor and so on. Actors in the SDFG synchronize only by communicating tokens over edges.

In Figure 3.1 we show an example of a dataflow graph with three actors. Each edge is annotated with the port rate of the corresponding source and destination actors. The single token of the self-edge of an actor prevents multiple firings of the actor simultaneously ensuring that the next firing of the actor will occur only after the execution of the previous firing is finished. The backwards edge models dependency such that an actor can fire only when there is enough storage space at the destination actor, this is called the *firing rules* of an actor. The number of initial tokens on the backwards referrers to the buffer size of the destination actor.

3.2 Formal definition

Definition 1. [Actor] *An actor is a tuple (I,O) that consists of a set of input ports* $I \subseteq$ *Ports and a set of output ports* $O \subseteq$ *Ports where an input port cannot be an output port at the same time* $I \cap O = \emptyset$.

Definition 2. [SDFG] *An SDFG is a tuple (A,C) with a finite set of A of Actors and a finite set* $C \subseteq Ports^2$ *of channels (or edges). The source of a channel is the output port of some actor and the destination is an input port of some actor. A port is connected to exactly one channel and a channel is connected to ports of some actor. For every actor* $a = (I,O) \in A$ *we denote the set of all channels that are connected to ports in I(O) by InC(a) (OutC(a).*

A channel in an SDFG referrers to either dependency edge or sequence edge. The dependency edge between two actors shows data dependency while the sequence edge shows the execution order. A channel can carry an unbounded number of tokens called initial tokens. The initial tokens are required for an actor to start its firing.
An example of an SDFG with three actors a1, a2 and a3 is shown in Figure 3.2. Actors a1 and a2 are connected via edge c1. Actors a2 and a3 are connected via edge c2. The execution of an actor is defined in terms of *firings*. When an actor a starts its firing, it removes Rate(q) tokens from all (p,q) $\in$ InC(a). When the firing ends, it produces Rate(p) tokens on every (p,q) $\in$ OutC(a). The rates determine how often the actors have to fire with respect to each other so that the distribution of tokens over all the channels is not changed. Actors synchronies by communicating tokens over the channels.
The backward edge models dependency such that an actor can fire only when there is enough storage space at the destination actor (firing rules of the source actor). The single token of the self-edge of an actor prevents multiple firings of the actor simultaneously ensuring that the next firing of the actor will occur only after the execution of the previous firing is finished.

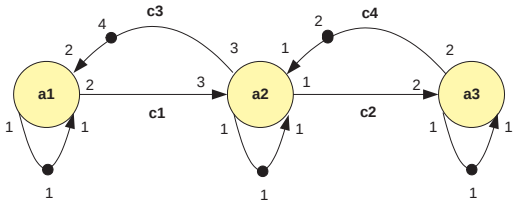

Figure 3.2: Example of an SDFG

Definition 3. [Repetition vector and consistency] *A repetition vector of vector q of an SDFG (A,C) is a function A → 0 in a way that for each channel (i,o) ∈ C from actor a ∈ A to b ∈ A, Rate(o) · γ(a) = Rate(i) · γ(b). A repetition vector γ is called non-trivial if and only if q(a) > 0 for all a ∈ A. An SDFG is called consistent if it has a non-trivial repetition vector. For a consistent graph, there is a unique smallest non-trivial repetition vector, which is designated as the repetition vector of the SDFG.*

The repetition vector of an SDFG determines the firing frequencies of its actors. The repetition vector of the SDFG actors (a1,a2,a3) in Figure 3.2 are (3,2,1) respectively. The repetition vector of the graph is non-trivial meaning that the graph is consistent. Consistency and deadlock free are important properties for an SDFG. Deadlock results when there are insufficient number of tokens in a cycle of the graph.

Definition 4. (Timed SDFG) *A timed SDFG is a triple (A,C,ϒ) consisting of an SDFG (A,C) and a function ϒ : A → 0 that assigns to every actor a ∈ A the time it takes to execute the actor once.*

In Figure 3.3, the execution time of each actor is denoted with a number in the actor.

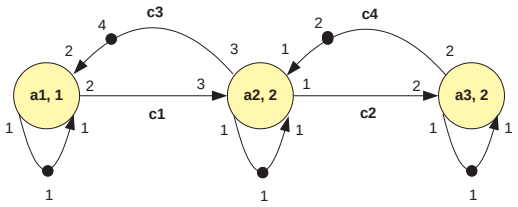

Figure 3.3: Example of a Timed SDFG

Definition 5. [State] T*he state of a timed SDFG (A,C,ϒ) is a pair (δ, υ). Edge quantity δ associates with each dependency edge c ∈ C the amount of tokens in that edge in that state. To keep track of time progress, an actor status υ : A → N^N associates with each actor a ∈ A a multiset of numbers representing the remaining times of different ongoing*

firings of a. We assume that the initial state of an SDFG is given by some initial token distribution δ, which means the initial state equals (δ,{(a,{ }) | a ∈ A}) *(with* { } *denoting the empty multiset).*

Definition 6. [Self-timed execution] *An execution is self timed if and only if clock transitions only occur when no start transitions are enabled.*

In the self timed execution of an SDFG, from one clock transition to the next, there can be some interleaving of simultaneously enabled start and/or end transitions. However, because these start and end transitions are completely independent of each other, independent of the order in which these transitions are applied, the final state before each clock transition, and hence also the state after each clock transition, is always the same. Self timed SDFG behavior is therefore deterministic in the sense that all the states immediately before and after clock transitions are completely determined and independent of the selected execution. In a self timed SDFG an actor fires as soon it can (i.e. as soon as it is enabled).

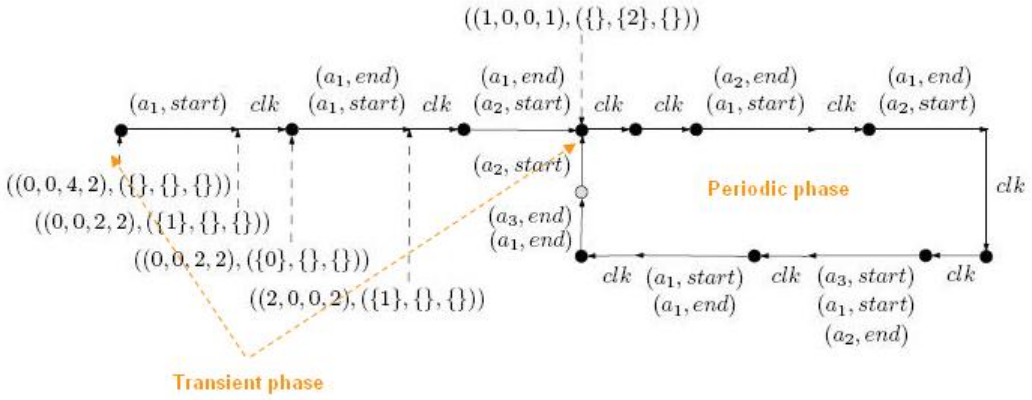

Figure 3.4: State-space of example SDFG showing the transient and periodic phases

Figure 3.4 shows the transition system of the self timed execution of the timed SDFG shown in Figure 3.3. All clock transitions are shown explicitly. Between clock transitions, there can be multiple start and/or end transitions enabled simultaneously. These start and end transitions are independent of each other. Independent of the order in which they are applied, the final state before each clock transition, and the first state after each clock transition, are always the same. Therefore, all start and end transitions are shown as one annotated step. A transition system consists of a finite sequence of states and transitions called the *transient phase* followed a sequence of states and transitions that is repeated infinitely often called the *periodic phase* as marked in Figure 3.4.

Definition 7. [Strongly-connected SDFG] *A strongly connected SDFG is a graph of which every actor depends in its firing on tokens from another actor.*

Strongly connected self timed SDFGs, the *transient phase* and *periodic phase* are important in the throughput calculation which will be discussed in the next chapter.

Definition 8. [Throughput] *The throughput Th(a) of an actor a for the self-timed execution σ of a consistent and strongly connected timed SDFG G = (A,D,Υ) is defined as the average number of the firings of actor a per time unit in σ. The throughput of G is defined as*:

$$Th(G) = \frac{Th(a)}{\gamma(a)}$$

where γ is the repetition vector (in Definition 3) of G and a *an arbitrary actor. The throughput of G gives the average number of iterations of the graph per time unit in σ.*

Definition 9. [Latency] *Latency between two actors* a *and* b *is defined as the time delay between the k-th firing of the source (src) actor* a *and the corresponding fining of the destination (dst) actor* b *within an execution σ*:

$$\mathrm{L}_k\sigma = \mathrm{F}_{dst,cf(src,k,dst)}\sigma \text{ - } \mathrm{F}_{src,k}\sigma$$

In the above formula, F *refers to the finishing time of the k-th firing of an actor and σ represents an execution. When computing the latency, only an execution of a complete iteration is considered.*

Chapter 4

SDF^3 Design flow and tooling

The SDF^3 tool is the implementation of the SDFG-based design flow and algorithms presented in [26] which provides techniques for mapping streaming applications with time constraints to a NoC-based MP-SoC.
The design flow objective is to minimize the resource usage (processing, memory, communication bandwidth) while at the same time offering guarantees on the throughput of the application when mapped to the system. The design flow consists of the following four iterative phases:

- *Memory dimensioning:* In this phase enough storage-space is made available for the tokens that communicate over the edges of the SDFG. This includes modeling the access to a memory tile when the token does not fit in the local memory of a processing tile. In this phase trade-off is made between the storage-space that is allocated to the edges in the SDFG and the maximal throughout that can be achieved under these allocation. The trad-off space is used to put constraints on the storage-space of the edges in the application SDFG.
- *Constraint refinement:* In this phase the storage constraints defined in the previous phase are used to compute latency and bandwidth constraints on the edges of the SDFG. The constrains all together are used in the next phase to bind the application actors to the tiles of the MP-SoC architecture.
- *Tile binding and scheduling:* In this phase, application actors that are bound to the same tile receive static-order scheduling to perform their tasks. TDMA time-slots on the tiles are also allocated in this phase and the storage-space allocation is minimized when possible.
- *NoC routing and scheduling:* In this phase TDMA time-slots are allocated on the links of the NoC by first extracting the NoC scheduling from the previous phase of the flow and then find a solution for the NoC scheduling, then the actual bandwidth used by the NoC schedule is computed.

All the information from the previous steps are used to determine the resources that are available for the next application that need to be mapped to the same NoC-based MP-SoC

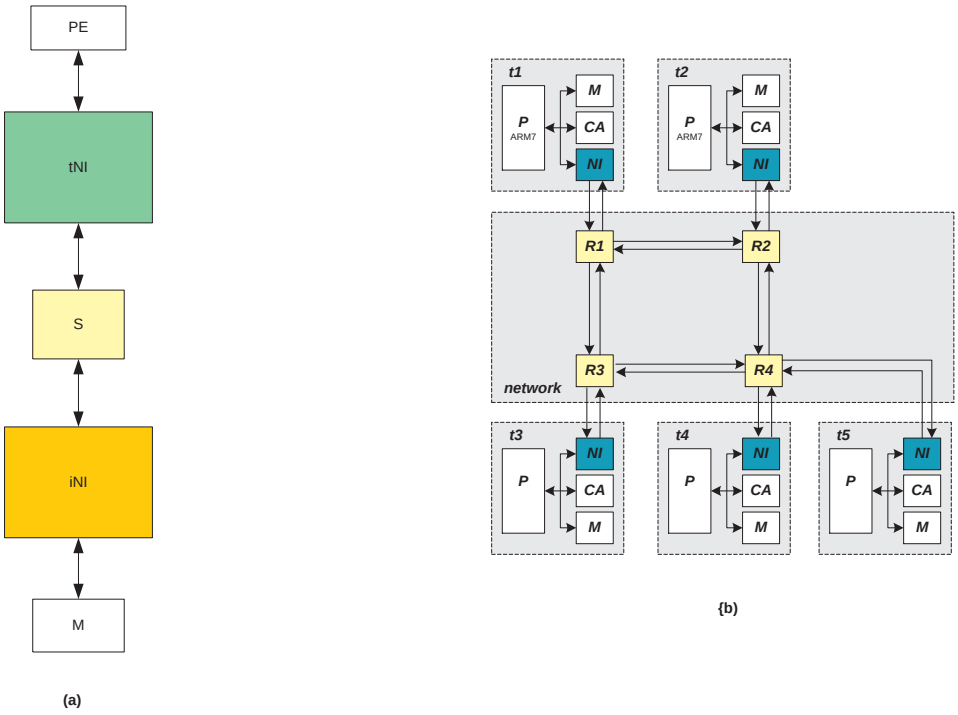

Figure 4.1: Platform Architecture

architecture.

In order to use the SDF^3 design flow, an application needs to be modeled as an SDFG specifying the execution time of the application's tasks (actors) when executed on a defined processor. In addition to that, the size of tokens communicated over the edges should be known. It is also required to provide the SDFG for the platform and interconnect. In [26] the platform architecture consists of processor tiles and memory tiles, each tile contains the NI of the NoC interconnect, computation and storage elements. Tiles communicate through links via the interconnect routers as shown in figure 4.1.b.
The result of the design flow is a multiprocessor SoC configuration that specifies a binding of the actors to the processor tiles and memory tiles, a schedule for the actors on the processors and a schedule for the token communication over the NoC.

The presented platform architecture and interconnect differ from U-NIC platform architecture in Figure 4.1.a. In U-NIC there are no fixed tiles structure, instead a processing element is connected to a network interface. Another difference is the scheduling algorithm of the NoC. U-NIC uses virtual channels for resource sharing and round-robin schemes for scheduling tasks on the same resource, while in the proposed design flow Æthereal [18] NoC architecture is used, which is based on TDMA time-slot allocations for sharing the network resources.

In order to use the design flow for U-NIC based SoC, minor modification to the platform architect can be conducted since every processing element can have its own internal memory and the NI in every tile can be mapped to the same NI, but the scheduling remains a major difference that require substantial modification in the proposed design flow. However, if U-NIC arbiters can provide the same guarantees of the TDMA scheduler, then the design flow can be modified for U-NIC use.
Due to the mentioned differences, the proposed design flow and the SDF^3 tool capabilities in generating the design space of a SoC cannot be used for U-NIC based SoC design as is in this assignment. However, the SDF^3 tool can still be used for analyzing the throughput and latency of a system that is modeled in an SDFG.
In this report we present a method to model, map and schedule a multimedia application onto U-NIC based SoC. The results is an SDFG of the SoC that can be analyzed via the SDF^3 tool.

4.1 SDF^3 Tool

SDF^3 is the implementation of the SDFG-based design flow and analysis algorithms in [26]. The input to the tool is the streaming application SDFG and platform architecture both in XML format. Example of the XML input files are included in the tool software package [28]. The tool performs automatically the steps described in the design flow and as output it generates a complete state of the design flow in XML. The outputted design flow provides the tile binding, NoC routing and scheduling. It is possible for the designer to modify the output file manually in order to change the design flow suggested by the tool.
SDF^3 implements various techniques and algorithms. The algorithms are outside the scope of this report, however for the purpose of the experiments performed in this report, a brief introduction to the algorithms for the throughput and latency analysis is provided, where more details can be found in [26] [16] [27] [15].

4.2 SDF^3 Performance analysis

4.2.1 Throughput

In multimedia applications, throughput is an important indicator to the performance constraints. The throughput is influenced by the available buffers in the system. More buffer means more data can be transmitted through the system and accordingly an increase in the throughput. However, more buffers means more silicon area (high cost) and power consumption (undesired), therefore trade-off between the throughput and buffer size [27] need to be made. The SDFG model allows the SoC designer to explore this trade-off using the throughput analysis capabilities of the SDF^3 tool as will be shown in a later chapter.
In [26], "the throughput of an SDFG refers to how often the actor produces an output token". Traditionally the SDFG throughput is defined as 1 over the maximal cycle mean (MCM) of the corresponding Homogeneous Synchronous Data Flow Graph (HSDFG) [23].

The corresponding HSDFG of a SDFG is an SDF graph where the execution of any actor in the graph results in the consumption of one token from each incoming edge of the actor and the production of one token on each outgoing edge. The transformation of an SDFG to a HSDFG can lead to an exponentially larger size than the size of the original graph in terms of the number of actors and edges, and accordingly this traditional method for analyzing the SDFG throughput is inefficient. As an alternative, [15] presented a method for analyzing the throughput of an SDFG based on state-space exploration of a self-timed strongly-connected SDFG. This method is implemented in the SDF^3 tool. The method states that the notion in Definition 8 is equivalent to 1 over the maximal cycle mean of the corresponding HSDFG by using two propositions related to the transient and periodic phase in Figure 3.4.

Proposition 1. *For every consistent and strongly connected timed SDFG, the self-timed state-space consists of a transient phase, followed by a periodic phase.*

Proposition 2. *For every consistent and strongly connected timed SDFG (A,C,Υ), the throughput of an actor $a \in A$ is equal to the average number of firings per time-unit in the periodic part of the self timed state space.*

The buffer capacity of an actor is modeled in SDF by the number of initial tokens on a backwrds edge from the actor to the source actor that attempts to communicate with it. In Figure 4.2 we show how the buffer capacity can be modeled in SDF graph by the backwards edge.

The throughput of an SDFG is limited by its critical cycle [23]. The critical cycle in an SDFG is the cycle with the maximal cycle mean, that is the total execution time over the number of tokens in that cycle.
By knowing the critical cycle of an SDFG, we know which components (actors, channels) in the graph that are limiting the throughput. In order to increase the throughput of the graph we can look at these limiting components and try to improve either the execution time of the involved actors or the capacity of the channels. By increasing the channel capacity, the flow of more tokens through the channel is utilized and accordingly the throughput increases.
Although the SDF^3 finds the critical cycles in the throughput computation, it does not indicate them in the output of the tool. Possible method to find a critical cycle in an SDFG is by examining the effect on the graph throughput when the execution time of an actor is modified. If no major change to the throughput, then that specific actor is not in the path of the critical cycle. In our experiment with U-NIC we will use this method to find the critical path in the obtained SDFG model.

4.2.2 Latency

In real-time applications that execute concurrently on NoC based SoC, the latency is an important performance indicator. In [16] the latency is defined by "the time delay between

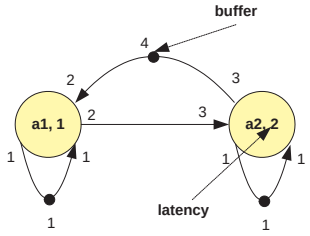

Figure 4.2: Buffer and latency model in SDF

the moment that a stimulus occurs and the moment that its effect begins or ends". In an SDFG, the actors firings and producing of tokens respond to the stimuli, while their effects are the consumption of the produced tokens by other actors.
The exsection order of the actors in an SDFG influences the latency. Since the execution order of the actors in an SDFG can be fixed at the design time (static scheduling), this makes the SDFGs statically analyzable. The execution order of the graph actors is called "the class of static order schedulers" [16].
An algorithm to determine the minimal achievable latency between the executions of any two actors in an SDFG is described in [16] and implemented in the SDF^3 tool. Latency computation is performed on a timed SDFG where the minimum latency is achieved by obtaining a class of static order schedules.
Resource arbitration strategy that uses a static order schedule starts with waiting till the first task in the sequence is ready to execute. After executing this task, the scheduler executes the next task in the sequence if it is ready, or it waits till the task becomes ready to be executed. Once this task is executed, it continues with the next task. This process is repeated till the schedule is finished or it continues indefinitely in the case of an infinite schedule.
Latency is modeled in SDF by the execution time of the actors in the graph as indicated in Figure 4.2.

Chapter 5

Method for modeling NoC based SoC

In this chapter we will present a method to construct an SDFG model for a multimedia system-on-chip (SoC). The method will help the SoC designers to model and analyse the performance of the system with the SDF^3 tool at design time.

In a multimedia SoC, the applications (e.g. video, audio and graphics) are implemented in tasks. These tasks are in turn implemented as software on programable processors or as hardware IPs known as processing elements (PEs). The processing elements in a SoC require certain data to perform their computation. For that data Read requests are submitted to a shared memory, then after performing the computation the data is updated by the PEs, for that data Write requests are submitted to a shared memory. Often this is an off-chip shared memory due to the size of data that need to be processed by the application, like video frames for example. Access to the shared memory is arranged by the network-on-chip and memory controller.

To be able to model a multimedia SoC, we need to understand the applications that are running on the system, their communication patterns and traffic characteristics when accessing the shared memory through the network.
Another aspect that needs to be considered is the buffer capacity in the system that ensures sufficient throughput to fulfil the application's constraints.

As a start point, we will build basic SDFG models for the application, the network and the memory, then connect the graphs based on the communication pattern of the application. Essential prerequisites for building the models are:

- the communication patterns and traffic characteristics of the application are known upfront;
- the statistical analysis of the application (worst case execution times of the processing elements) is available;

- worst case execution times (WCETs) at the network elements and memory are available.

The above perquisites are necessary when mapping the application actors to the network and when allocating the memory resources. The size of data that is communicated through the network via the external memory influences the allocated network resources and the required buffering space. Also the execution time of each task (code segment) is required to perform the throughput and latency analysis.

The WCETs at the network elements depend on the scheduling algorithms implementation and can be provided either by careful analysis of the implemented scheduling algorithms or from simulation results, however, simulation provides no guarantees or upper bounds on the latency.
Static-code analysis of the application source code can be extracted by specialized tools [21]. These tools take an application as input and analyze its memory and execution time requirements.

The method incorporates the following steps:

1. model the application's main code tasks
2. model the network-on-chip request and response paths
3. build basic model for the external memory
4. construct the SoC model: analyse and map the application to the network based on the application traffic characteristics.

5.1 Formal definitions

The following definitions will be used in describing the above steps.

Definition 10. [Request] *A Request is either a read or write transition from the application actors through the network.*

Definition 11. [Master actor] *A master actor is an actor that issues read or write requests to the network.*

Definition 12. [Slave actor] *A slave actor is an actor that receives a read or write request from a master actor through the network.*

Definition 13. [Request path] *A request path is the path that a request takes through the network from the master actor to the slave actor.*

Definition 14. [Response path] *A request path is the path that a request takes through the network from the slave actor to the master actor.*

Definition 15. [Execution actor] *An execution actor is an application actor that performs the computation of a code with the actual execution time as estimated in the static-code analysis.*

Definition 16. [Auxiliary actor] *An auxiliary actor is an actor with an execution time equals to 0. Often required to mimic the internal communications between some other actors.*

Definition 17. [Sequence edge] *A sequence edge is a channel from a source actor to a destination actor.*

Definition 18. [Dependency edge] *A dependency edge is a channel with tokens from a destination actor to a source actor.*

5.2 Application model

We start by analyzing the application to determine the main processing elements (PEs) and the communication characteristics between the PEs, mainly the size of the communicated data, the sequence of which data is communicated and the data dependency between the PEs.
Per each processing element, we assign an actor named by the processing element and repeat until all PEs have assigned actor. The actors are connected by channels where the in/out port rates of the actors depends on the size of communicated data in tokens. Figure 5.1 shows an SDFG model of an application that consists of three processing elements (ABC). Actor A performs its computation on 64 data units, each data unit is presented by a token. Actors B and C perform computation on 1 data element.

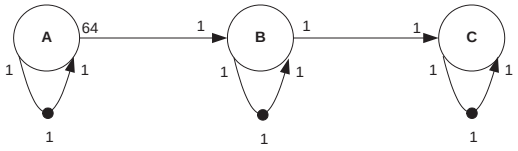

Figure 5.1: SDFG of an example application

In Chapter 2 we showed that for round-robin arbitration (as for TDMA and weighted round-robin) the worst-case latency of one request at any shared resource can be computed independently of the other requests that share the same resource. Based on this observation, we can model each read/write request (application task) to the shared memory through the network as if it communicate via a separate network. Therfore, the

application SDFG is fine grained (see Figure 5.2) by replacing each actor that communicates data through the network by five actors: (a) three actors for the read request of which one *execution actor* performs the actual computation on the data and two *auxiliary actors*; (b) two actors for simulating the write request of which both are *auxiliary actors* since in the write request no computation is performed. The *auxiliary actors* simulate the read/write request/response as well as for the sequence and data dependency between the read and write requests. The arrows to and from the NoC represents the requests and responses respectively. The horizontal thick arrows represent the sequence and dependency edges. Figure 5.2.a. shows the fine grained SDFG for actor A where actor *a1* sends read request of 64 data elements to the network; actor *a2* receives the read responses of 1 data element at a time and sends 1 token on its output edge to actor *aexe*; actor *aexe* requires 1 token to start its computation then it fires 1 token on its output edge to actor *a3* indicating the completion of its computation on one data element so that actor *a3* can send write request through the network, actor *a4* receives the write response and sends 1 token on its output edge to *aexe* indicating that the next data element can be sent. The edge from *aexe* to *a3* is a *sequence edge* while the edge from *a4* to *aexe* is a *dependency edge*.
If an actor doesn't communicate data through the network, it will be replaced by three actors, one actor for the actual computation and two *auxiliary actors* for the sequence and data dependency with the other actors in the graph. Figure 5.2.b gives the fine grained SDFG for actor C which doesn't communicate data via the network.

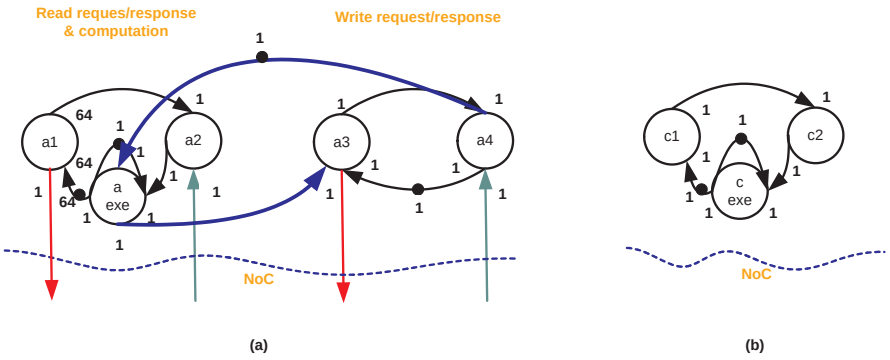

Figure 5.2: Fine grained SDFG of actors A and C in the example application

5.3 Network model

Most network-on-chip (NoC) architectures consist of network interfaces (NI) and switches or routers [18] [10]. The network model consists of two type actors, NI and Switch. The NI connects *master* and *slave* actors to the network, while the switch connects NIs. Figure 5.3 provides a basic network channel model for an example NoC which consists of two net-

work interfaces and one switch. The NI are connected to *master* and *slave* actors. Both request and response paths are illustrated. NoC with different topology, can be modeled in a similar way. The only difference is that the channel model shall contain multiple switches depending on the route that is taken through the network. Figure 5.4 shows an example of a network channel model for multiple routers mesh topology. In this example the request and response follow the same path as the case in the study network U-NIC, but in general the response path can be different depending on the network architecture and routing schemes.

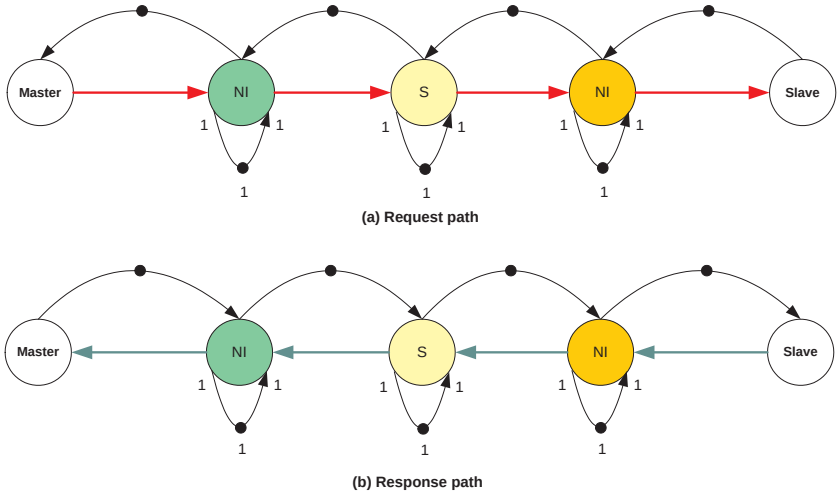

Figure 5.3: NoC channel model

The backwards edges with tokens model the buffer capacity of a network elements. In the network, buffers are needed for the data and the flow control communication between the network elements. Tokens are sent from a source actor to a destination actor when there is sufficient buffer space at the destination actor, flow control provides the information on the available buffer space at the destination actor so that the source actor can send more tokens depending on the free buffer space.

5.4 Memory model

The memory model consists of two actors, one actor for receiving the read/write requests from the network and processing the request and one *auxiliary actor* for sending the read/write responses via a network as shown in Figure 5.5.

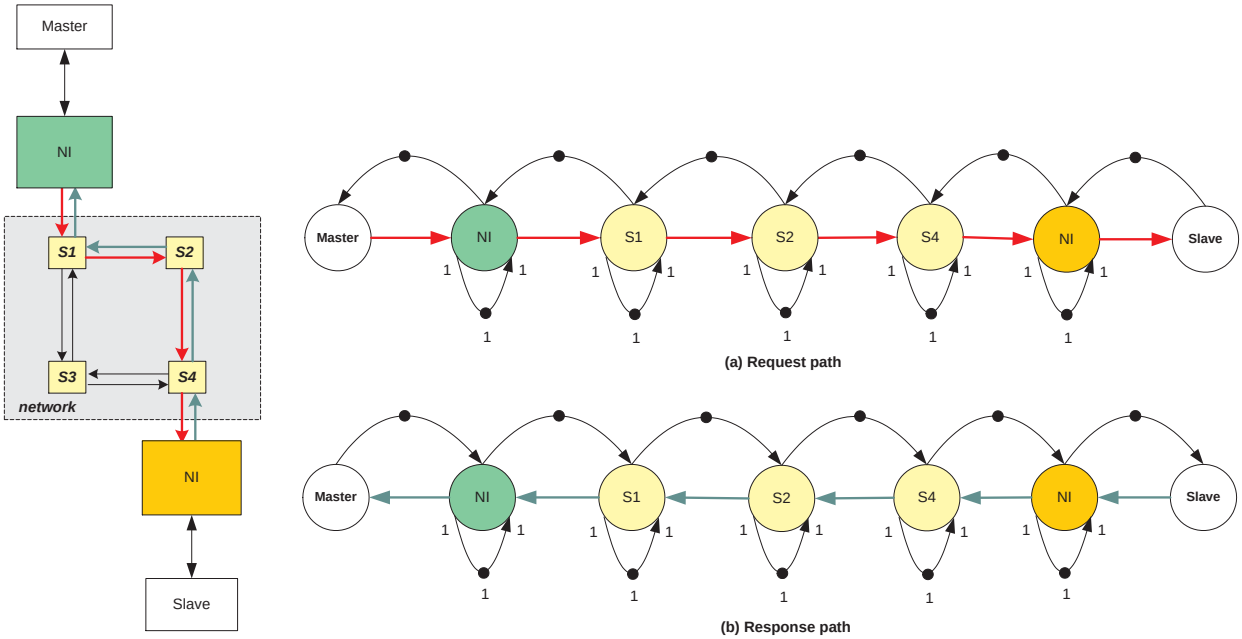

Figure 5.4: NoC channel model for multiple mesh topology

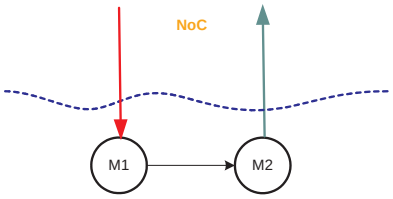

Figure 5.5: Basic Memory model

5.5 Construct the SoC model

5.5.1 Application mapping

First we analyse the application's traffic characteristics and identify the actors (*master actors*) that communicate data through the network. We proceed by building a simple model of the SoC to reflect the application's communication patterns through the NoC on a high level where the network is modeled by one actor for simplicity. In the example application, actors A and B communicate by sending data to a shared memory (*slave actor*) through the network. This is illustrated in Figure 5.6.

For every main actor in the application, we built a fine grained SDFG. The arrows that represent the read/write requests and responses through the network are replaced by the

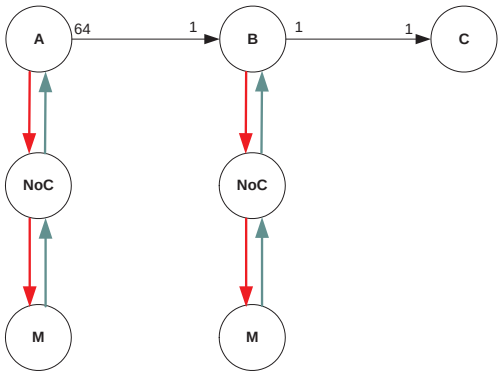

Figure 5.6: Abstract SoC SDFG for example application

network channel models.

5.5.2 Application scheduling

We schedule the execution sequence of the application actors via sequence and dependency edges as indicated previously in Figure 5.2.a. The scheduling is based on the execution sequence of the code segments and the size of communicated data. The dependency edges are used as control channels between the code main actors similar to the use of flags in the code.

5.5.3 Port rate specification

Every actor in the SoC SDFG has a total number of (input and output) ports equals to the number of input and output channels that are connected to the actor. A port rate depends on the read/write request size, the network data width (word size) and the communication unit (tokens) representation. Often the read/write requests are converted to different data size before sending it to the network. In Figure 5.2.a of the example application, actor A sends read request to the shared memory of 64 bytes, performs computation and sends write request of 64 bytes to the shared memory. If we consider the network word size 64-bit=1 token, then for writing 64 bytes to the memory, a total of $(64 \times 8 \times \frac{1}{64(bits)}) = 8$ transactions are required which means actor that issues the write request of 64 bytes has an output port rate on the channel to the network equals to 8 tokens.

5.5.4 Minimum buffer allocation

To avoid any deadlock in the dataflow graph sufficient tokens should be available on every cycle in the graph. The minimum number of the initial tokens on a dependency edge from destination actor b to source actor a must be $\geq$ the output port rate of the source actor a on the channel to the destination actor b. However, when we consider the resources sharing, not only the availability of sufficient tokens on an actor input can guarantee its

immediate firing, but also the schedule that is used on the shared resources. In an SDFG an actor fires when it has sufficient tokens and when the actor's firing is scheduled on the shared resource. This property is referred to by *causal dependency* in [30]. When this property holds for all actors sharing resources in the graph, then we can guarantee by providing sufficient tokens on every cycle in the graph that it will not deadlock. In case more than one task send data on the same channel (share the same resource), then the buffer size of the shared resource in the implementation should be the sum of the sized buffers required by each of the concurrent tasks.

Finally to complete the SDFG model of the example application SoC, the worst case execution times are added to the SoC actors. In the case study of the next chapter the above steps are discussed in more details.

5.6 Place in a design flow

Typical NoC implementation design flow consists of iterative steps to generate, configure and analyse the network performance [17]. A NoC design flow proposed in NXP incorporates similar iterative steps where the required inputs to the design flow are the application communication requirements, the network topology and performance constraints (throughput, latency, power, area). Our method to model and analyse the performance of NoC based SoC can be placed on top of the implementation design flow as shown in Figure 5.7. The SoC designer can construct an SDFG model of the system as described above and run the performance analysis to check if the application constraints are met, if not, then the designer can check the critical cycle in the model and try to resize the buffers in the path of the critical cycle or change the application mapping or scheduling until the constraints are met.
The performance of the constructed SDFG provides guarantees on throughput and latency provided that the latency is accurately computed and that no component that can influence the latency computation is missed in the model.
Once the required performance constraints are achieved, the implementation phase can start. In the implementation phase the network is generated with NXP SoC design environment using the RTL and SystemC libraries. The generated RTL and SystemC views of the network are used to verify the network performance through simulation (SystemC and RTL VHD).

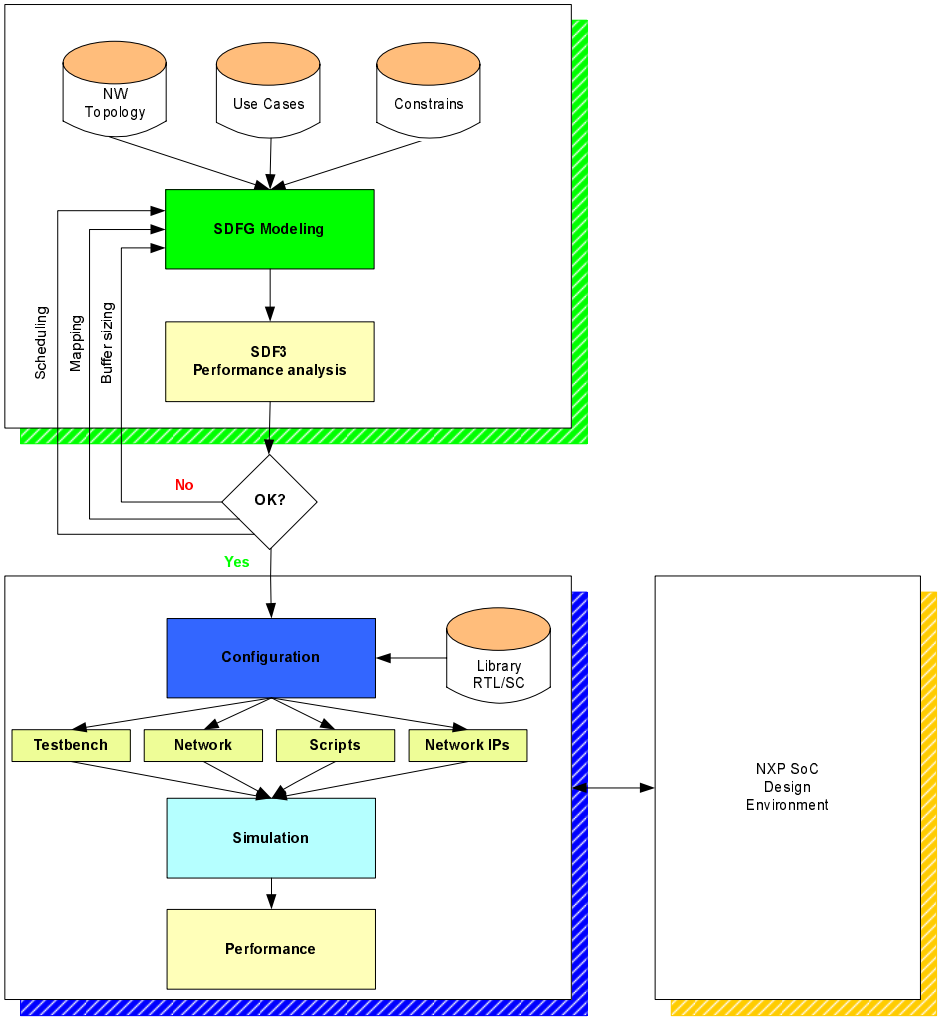

Figure 5.7: Place in NXP NoC design flow

Chapter 6

H263 Video Decoder Case Study

We apply the method to a multimedia application (H.263 video decoder) running on a SoC. The application consists of four cores (processing elements) performing the code computation on U-NIC based SoC and an external memory.

Based on the Synchronous Dataflow (SDF) model of computation [26], we measure the throughput and latency of H.263 video application when mapped on a U-NIC base line network using the analysis capabilities of the SDF^3 tool. The full XML file of the SDFG model is enclosed in appendix 1.

6.1 Model H.263 video decoder

The H.263 standard is intended for video compression of streams with low resolutions. The decoder performs the reversed operations of a H.263 encoder [3] in order to reconstruct the original video stream.
In this experiment, we assume that the H.263 decoder operates on frames with QCIF resolution (176 $\times$ 144 pixels) and that the received video stream has a maximal frame rate of 15 frames per second known as ***throughput constraints*** of the application.
In [26] the SDFG for the H.263 application is modeled by 4 actors; each actor corresponds to one of the code main tasks and edges for the tasks to communicate as shown in Figure 6.1. The H.263 application has the following main tasks:

- Variable Length Decoder (VLD) decompresses the data streams
- Inverse Quantization (IQ) de-quantizes elements in a block of 8X8 data elements
- Inverse Discrete Cosine Transform (IDCT) transforms blocks from frequency domain to space domain.
- Motion Compensation (MC) reconstructs the original video frame

Model description
The VLD actor, on each firing, produces decompressed data or encoded macro blocks

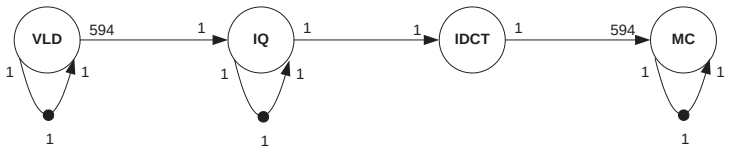

Figure 6.1: H.263 SDFG Model

(MBs) for a complete video frame. Each macro block captures the image data for a region of 16 × 16 pixels. Inside the macro block, the image data is divided to 6 blocks, each block of 8 × 8 = 64 data elements. 4 blocks contain the luminance values of the pixels inside the MB and 2 blocks contain the chrominance values as shown in Figure 6.2. The macro block represents 6 × 8 × 8 = 384 data elements. In a video frame with QCIF resolution, we have in total 11 × 9 = 99 MBs and accordingly 99 × 6 = 594 blocks.

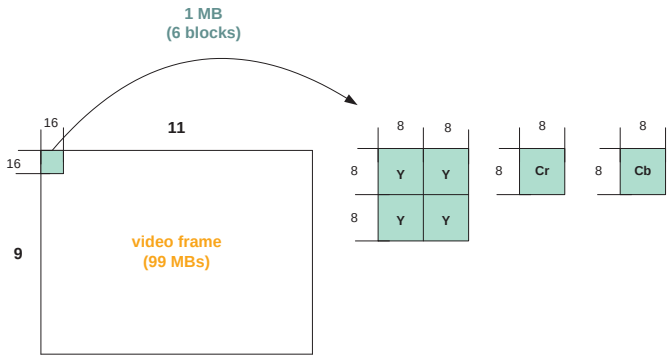

Figure 6.2: Video frame size

The inverse quantization (IQ) and inverse discrete cosine transformation (IDCT) actors revert the MB encoding.
The IQ and IDCT actors operate on a single block of encoded pixel data instead of a complete MB. This allows smaller memory requirements for these actors when compared to the full frame decoding.
The motion compensation (Motion comp.) actor takes a group of 594 blocks to reconstruct the original video frame.
The self-edges with one token on VLD, IQ and MC actors prevent simultaneous firings of the corresponding actor and ensure that the next firing of the actor will occur only after the execution of the previous firing has finished.

6.2 Model U-NIC baseline

U-NIC baseline platform which is considered in this experiment consists of:

- Target network interface (tNI): Interfaces with the processing element
- Initiator network interface (iNI): Interfaces with the external shared memory
- Switch (S): Interfaces between the iNI and tNI

Figure 6.3 shows the SDF model for U-NIC network connection with a processing element and an external memory. For the purpose of this case study, and since U-NIC architecture doesn't allow the requests and responses to be on the same VC, we consider the use of two separate virtual channels (VCs), one VC for the Read/Write requests (*request path*) to the memory and one VC for the Read/Write responses (*response path*) from the memory. Since the worst-case latency at any shared resource is computed independently from the other requests that share the same resource (discussed in Chapter 2), we can model the requests sharing the network resources independently in a SDFG as if each request is connected to a separate network.

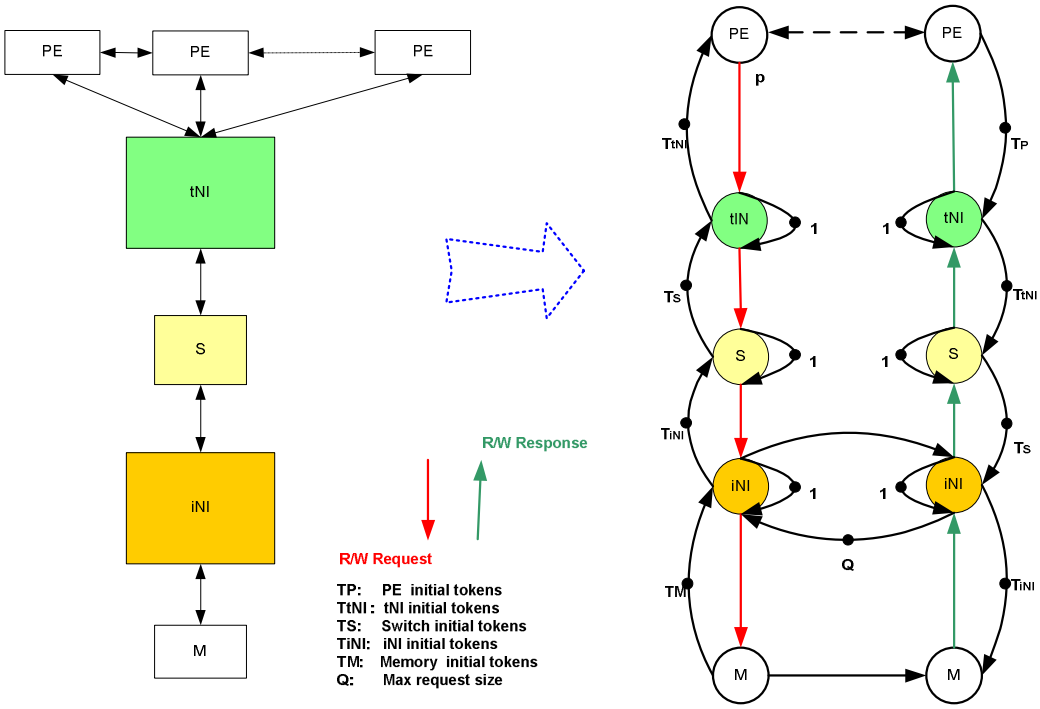

Figure 6.3: U-NIC SDFG Model

Model Description

The network components are modeled in the SDF graph as actors named by the corresponding network component (tNI, S, iNI) as shown in Figure 6.3. The processing

elements and the memory are actors labeled by PE and M respectively. Each network actor has self-edge with one token to ensure that tokens are processed one by one; in this way auto-concurrency is prevented.
Communication between the actors is modeled by edges. Each edge is annotated with the port rates of the corresponding output/input actors. These rates refer to the produced and consumed tokens when the actor executes/fires (In Figure 6.3 the rates of the ports are not indicated). The dotted arrow corresponds to the internal communications between the PE actors.
U-NIC link between any two actors can process only 1 word (32-bit word) per cycle. The port rates of U-NIC actors (tNI, S, iNI) is equal to 1.
When the processing element actor fires it produces 'p' tokens, the fired tokens represents either read or write request to the memory. The number of fired tokens depends on the size of the read or write request. In this base line model, the tNI actor requires 1 token on all its input ports to fire. In order to ensure that the graph will not deadlock, which is an important property for the SDFG [26], sufficient number of initial tokens on each cycle in the graph should be available assuming that the *causal dependency* property which is discussed in the previous chapter holds.
The number of initial tokens on the edges influences the actors' firings. In Figure 6.3 the backwards edges are labeled with the number of initial tokens equal to the buffer capacity of the destination actors respectively. This is to guarantee that no overflow will occur in any of the network components. In other words, tokens may be transmitted from one actor to another if there is enough buffer space at the receiving actor. Buffer size is illustrated by the number of tokens Tx (x: actor name) on the backward edge. When multiple processing elements share the same resource in parallel execution, then the buffer size of the shared resource in the implementation is the sum of the required buffers for each task.
U-NIC requires that the iNI at the response side has sufficient buffer capacity to accommodate the full request before the tokens of the full request are sent to the memory, therefore in order to fulfill this requirement edges between the iNI on the response channel and the iNI on the request channel are added to the model (horizontal dependency edges). The initial number of tokens on the horizontal edge is at least equal to the maximum request size of the application (Q).

6.3 SDFG model of the H.263 SoC

The communication pattern between the application actors and the memory actors through the network is essential in building the SoC SDFG model, therefore we discuss in details how the application communicates through the network.

6.3.1 Application communication pattern through the interconnect

The following communication scenario of the video decoder H.263 is considered:

- VLD sends 1 read request to the memory to read a frame of 594 blocks. The memory sends read responses to the VLD, in every response, 1 word (32 bit) is sent until

the full frame of 594 blocks is read. Per a block of $8 \times 8 = 64$ bytes, the memory fires $(64 \times 8 \times \frac{1}{32(bits)}) = 16$ times, which means that the memory fires in total $(594 \times 16) = 9504$ times as read response. Once 6 blocks (1 Macro Block) are received from the memory, the VLD decompresses the blocks and stores the decompressed blocks in the memory. When the full frame is decompressed by the VLD, a token is sent to the IQ indicating that the frame is decompressed and stored in the memory.

- The IQ reads the frame 1 block at a time, decodes it and sends it to the IDCT.
- The IDCT processes the blocks one by one to the MC.
- Once 6 blocks are decoded by the IQ and IDCT, the MC reconstructs the MB and stores the reconstructed MB in the memory until all the frame blocks are processed.

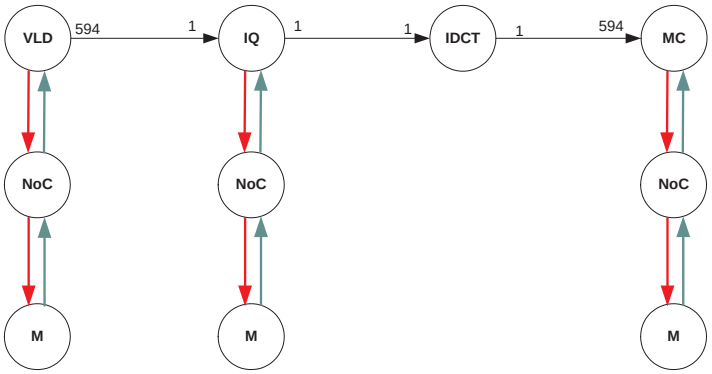

Figure 6.4: Abstract SoC SDFG for H.263

6.3.2 Model construction

First we identify which main code segments require access to the memory through the interconnect. Considering the above communication scenario, Figure 6.4 provides an abstract SoC model for the H.263 decoder where the VLD, IQ and MC send read and/or write requests to the external shared memory via the network, while the IDCT makes use of the local memory and thus has no connection with the network.
We construct the complete model of the H.263 video application when mapped to U-NIC base line platform by building sub-models representing the execution of each code segment then binding the resulted sub-models.
To keep the graph readable, the port rate is mentioned only when the consumed or produced tokens are > 1. If no value is given to the port rate, then it is by default 1.

VLD Read SDFG model

In order to mimic the read request, read response and the MB decompression of the VLD code, three actors *vld1*, *vld2* and vld_{exe} are introduced as shown in Figure 6.5, where

vld1 and *vld2* are *auxiliary actors* while vld_{exe} is the *execution actor* which performs the actual VLD computation. Communication order: *vld1* sends one read request through the network *request path* to read a full frame (594 blocks) from the external memory. The memory receives the read request and starts sending the frame word by word via the *response path*. The memory is modeled by two *auxiliary actors Mvld1* and *Mvld2*, where *Mvld1* receives and processes the read request and *Mvld2* sends the read response. For each block of 8×8 data elements, 16 U-NIC words of 32-bit are sent from the memory through the network (1 byte/data element in the memory). Meaning that the memory will fire in total $594 \times 16 = 9504$ times as read response. Actor *vld2* at the response side receives the frame data sent from the memory in words and sends it to actor vld_{exe}, upon receiving a total of one MB ($6 \times 16 = 96$ words), vld_{exe} decompresses it and produces 96 tokens on the edge to *vld1*. Once *vld1* receives in total 9504 tokens it sends a read request of the next frame and so on.
The horizontal dependency edges between the iNI on the request and the response channels are to model the buffer size at the response side since U-NIC requires that the iNI at the response side has sufficient buffer capacity to accommodate the full request before the tokens of the full request are sent to the memory. The initial number of tokens on the horizontal edge is equal 9504 words which is the frame size in words.

VLD SDF model

In the same way, in order to model the write request/response of the VLD, two *auxiliary actors vld3* and *vld4*) are introduced communicating as follows: *vld3* sends write request of one MB ($6 \times 16 = 96$ words) to the memory via the network *request path*. Once the memory receives the full MB, it will fire one time to actor *vld4* through the *response path* indicating the write response of 1 MB. This will repeat until the full processed frame is written in the memory.

The VLD model consists of the VLD read and write sub-models communicating via the horizontal sequence and dependency edges in blue as illustrated in Figure 6.6. Actor vld_{exe} sends 1 token to *vld3* per processed MB. This will instruct *vld3* to send a write request of 1 MB to the memory. When write response is received by *vld4*, it sends one token to actor vld_{exe} indicating that the next MB can be processed. Actor vld_{exe} fires and executes once the number of tokens that will be consumed is available on all its input edges. The initial token on the edge from *vld4* to vld_{exe} ensures that vld_{exe} starts and that the actors fire sequentially.

VLD IQ SDFG model

Like in the VLD read model, the IQ read request is modeled by three actors *iq1*, *iq1* and iq_{exe}. The size of the IQ read request is one block of 8×8 data elements. For the read request of one block 16 U-NIC words of 32-bit are sent from the memory through the network.
The VLD and IQ models are connected via an additional actor *vld0*. This actor is required

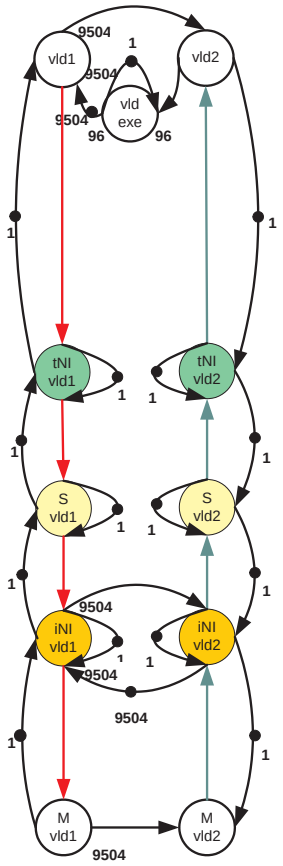

Figure 6.5: VLD Read SDFG Model

to ensure that the full frame (594 blocks) after being processed by the VLD and stored in the memory before it is read and processed by the IQ block by block as shown in Figure 6.7.

VLD, IQ IDCT SDFG model

The IDCT model consists of actors *idct1*, *idct2* and $idct_{exe}$. Since the IDCT executes per block, it can start execution immediately after the IQ as the block that is processed by the IQ can be temporarily stored on the local memory. The communication with the local memory is not explicitly modeled in SDF as the latency resulted from this internal communication is added to the involved actors execution times. The IDCT has no read or write request to the shared memory. Actor iq_{exe} sends 16 tokens to *idct1* per processed block of 8×8 data elements. This will instruct *idct1* to fire 16 tokens to *idct2* which fires in its turn 16 times to $idct_{exe}$, where the actual execution takes place, and to the iq_{exe} indicating that another block can be processed by the IQ. Figure 6.8 shows the model of

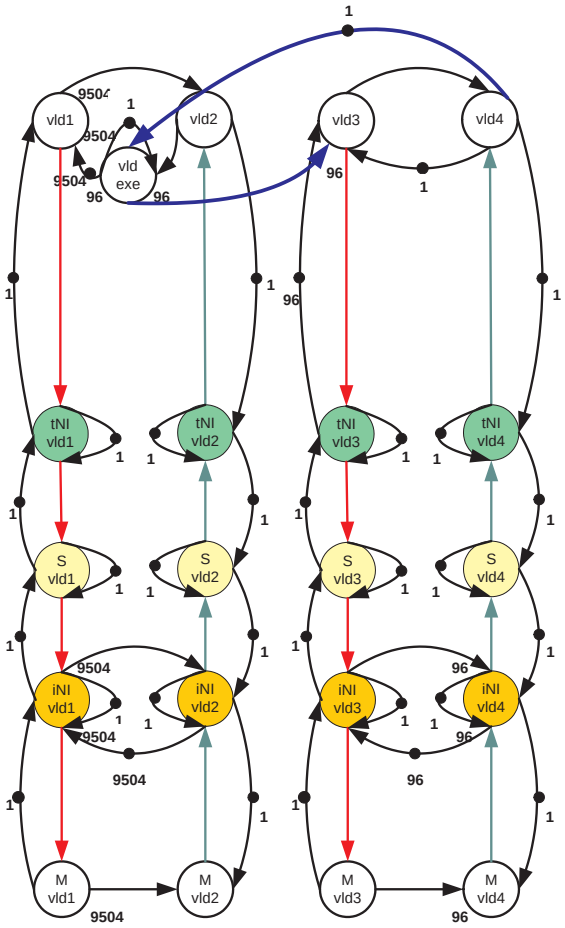

Figure 6.6: VLD SDFG Model

the VLD and IQ connected with the IDCT model via the horizontal edges in blue which model the communication sequence.

Full SDFG model

The MC model consists of actors *mc1*, *mc2*, mc_{exe}, *mc3* and *mc4*. Similar to the IDCT execution, actors *mc1*, *mc2* and mc_{exe} execute on a macro block (MB) following 6 consecutive executions of the IDCT. The 6 blocks that are processed by the IDCT are stored on the local memory. Actor mc_{exe} sends 1 token to *mc3* per processed MB. This will instruct *mc3* to send write request of 1 MB to the memory via actors *mc3* and *mc4* similar to the write request model of the VLD.
Figure 6.9 shows the full model of the H.263 system where the VLD, IQ and IDCT models are connected with the MC model via the horizontal edges in blue which model the communication sequence.

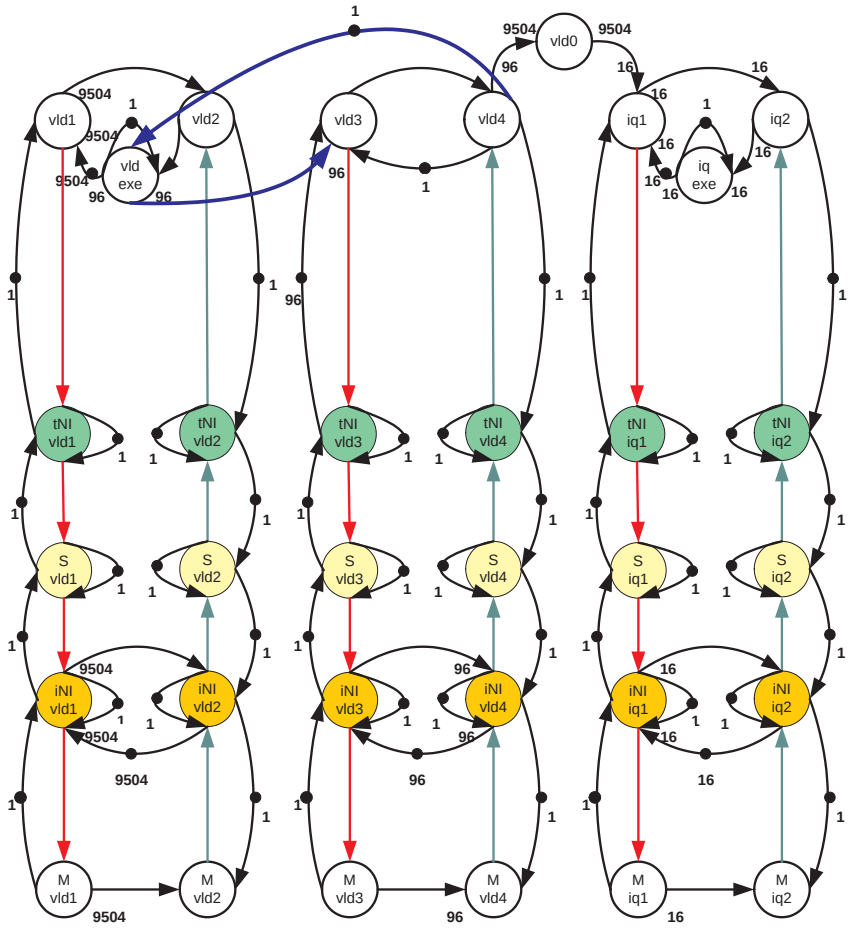

Figure 6.7: VLD IQ SDFG Model

In Figure 6.10 a backward edge is added connecting actor *mc4* with actor *vld1*. This backward edge has initial tokens equivalent to the size of a video frame indicating the reserved frame buffer in the memory. In case the initial buffer space is two video frames or more, parallel execution of the code segments can take place leading to an increase in the throughput as will be shown in the performance analysis in the next chapter.

6.3.3 Worst Case Execution Time assignment

In order to analyse the system performance we need to provide some statistics on the execution times of the actors in the constructed SoC SDFG model.

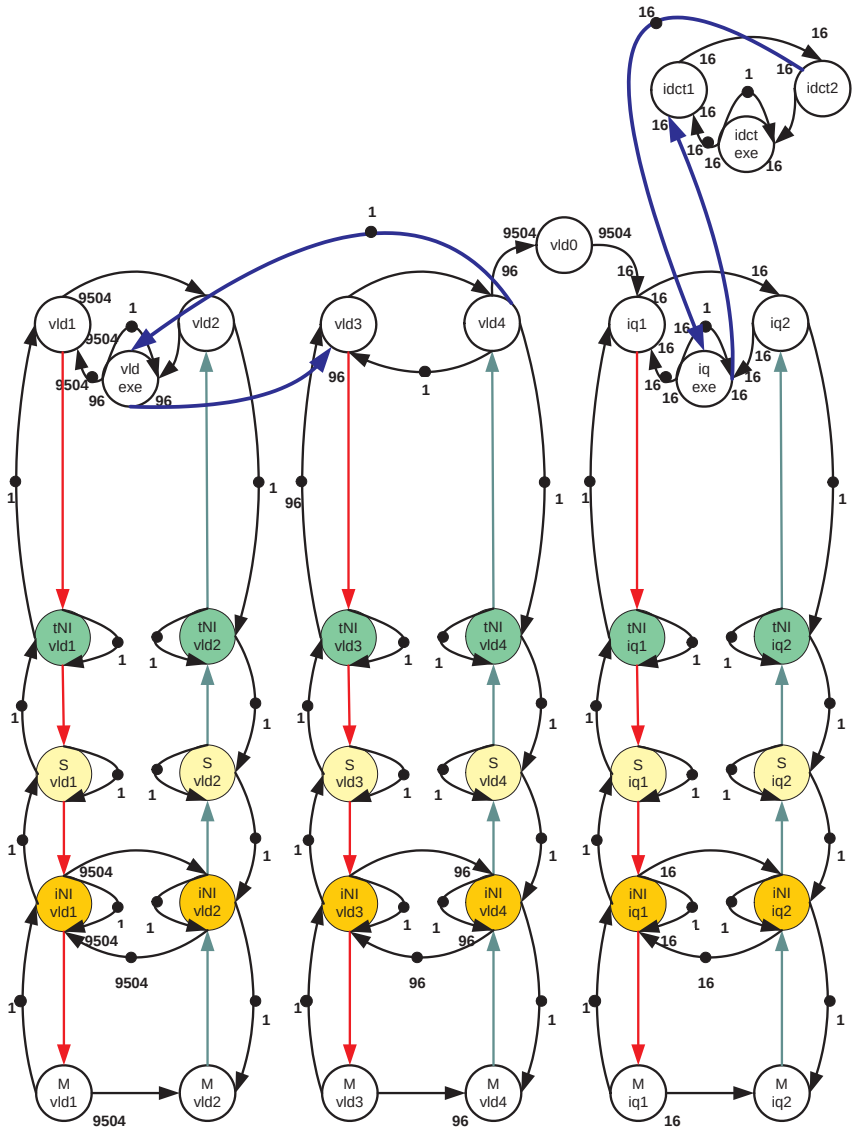

Figure 6.8: VLD, IQ IDCT SDFG Model

Execution times of H.263 actors

Worst case execution times of the H.263 main code segments shown in Tabel 6.1 are based on the codes execution on an ARM7 processor with clock frequency 100MHz [26] by applying static-code analysis of the application source code via specialized tools [21]. The codes segments IQ, IDCT and MC execute on the same processor, therefore arbitration inside the processor shall take place to schedule the execution of these codes on the same shared resource. The latency due to this arbitration is assumed to be included in the code

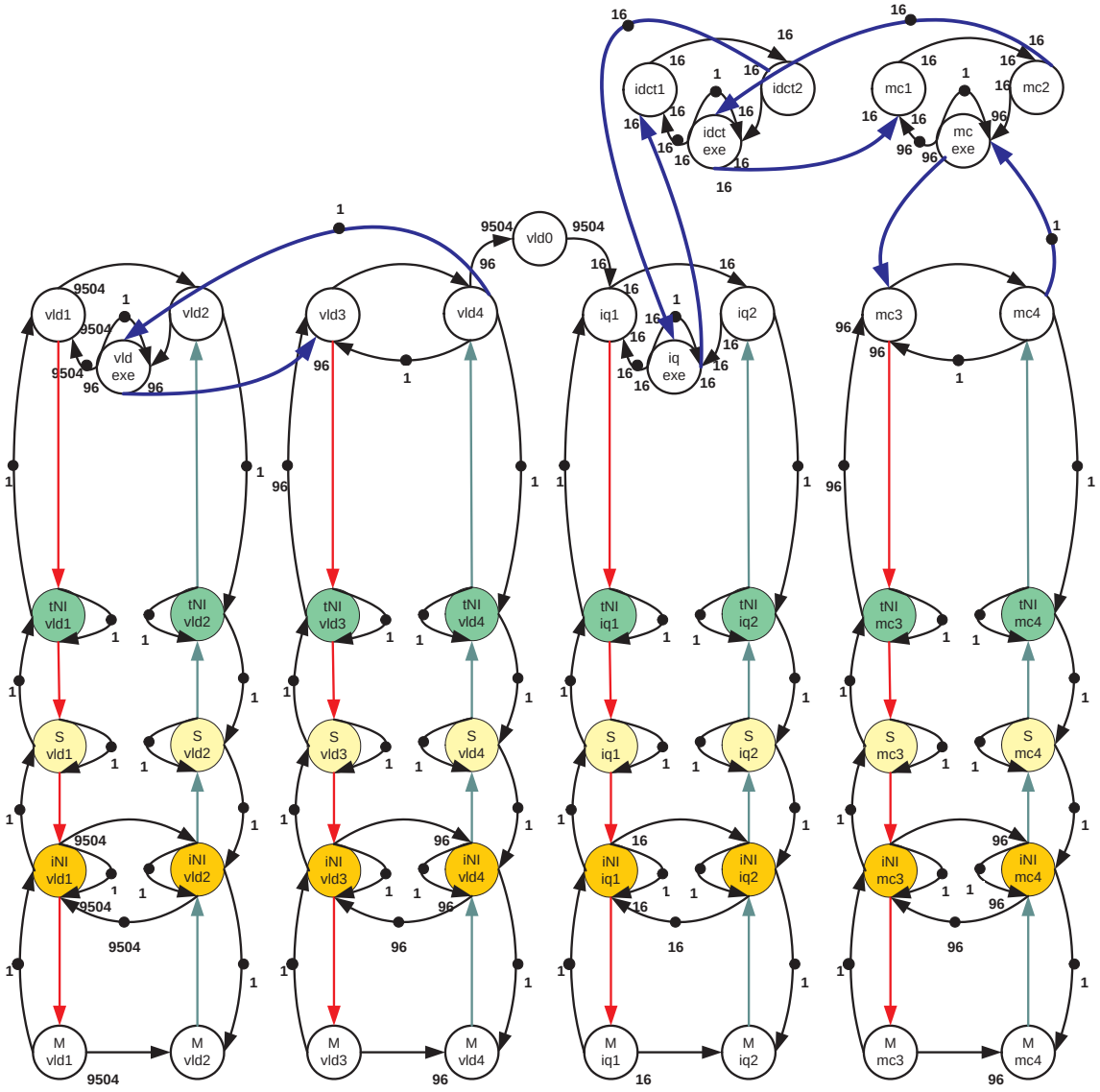

Figure 6.9: h263 SoC SDFG Model

segments WCETs.

Execution times of U-NIC actors

The worst case execution time of each U-NIC component is influenced by its arbitration. When a request is sent by the PE to the tNI it will be written to a FIFO inside the tNI. If more than one VC is used, there can be multiple FIFOs in the tNI and accordingly an

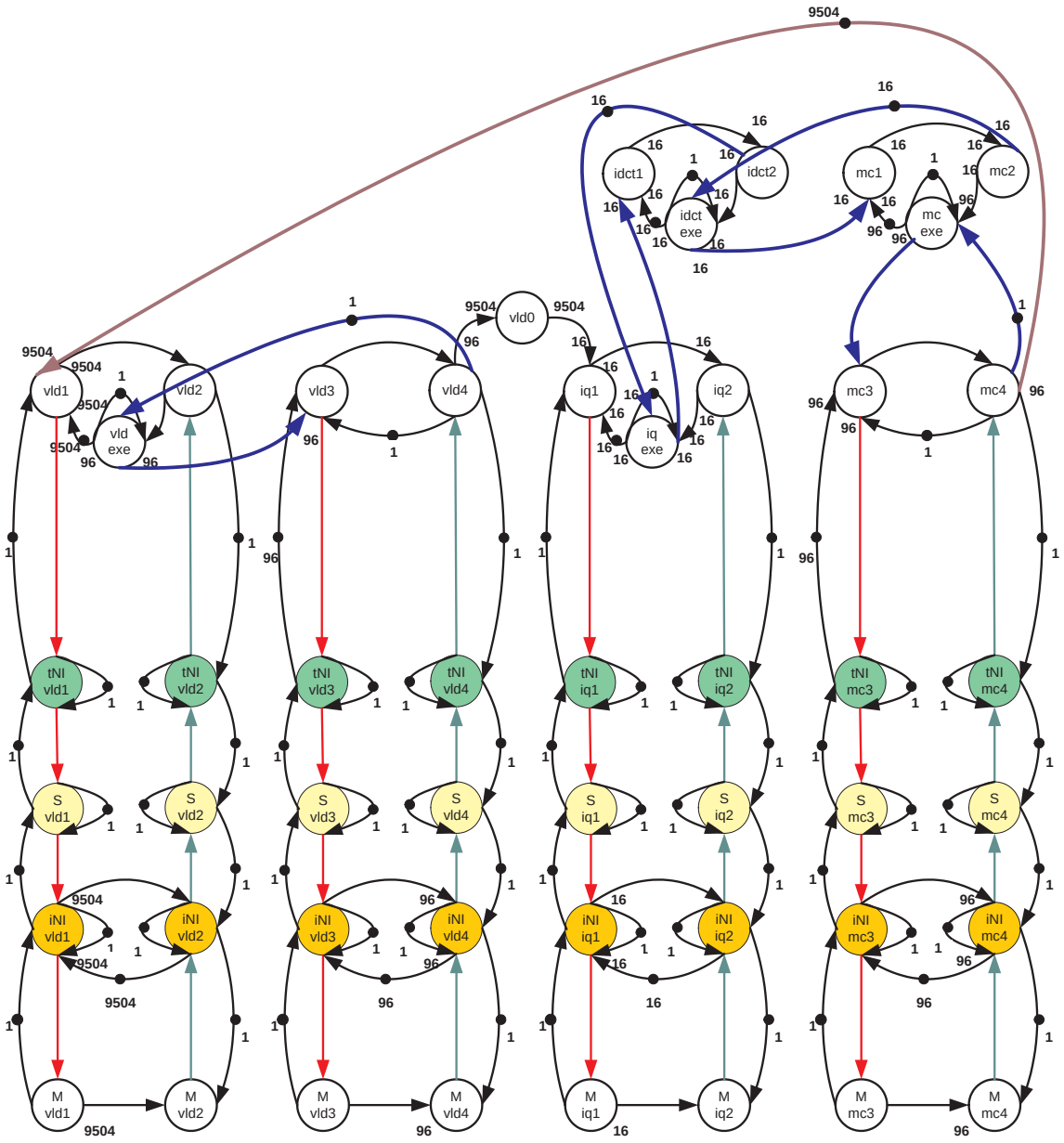

Figure 6.10: Final SDFG Model

inter-vc arbitration is needed to select one of these FIFOs to write the data to. If more than one request share the same VC, an intra-vc arbitration is required to select which request to write to the selected VC FIFO. Then the data must be scheduled on the output link of the tNI, if more than one VC is used, an inter-vc arbitration is need to select one FIFO.

Table 6.1: **H.263 WCET**

Actor	Execution Time	
	cc	**nsec**
vld_{exe}	26018	260180
iq_{exe}	559	5590
idct_{exe}	586	4860
mc_{exe}	10958	109580

In the Switch there can be more than one port and per port more than one VC, accordingly multiple FIFOs might exist, therefore inter-vc arbitration is required at the switch to select one FIFO. At the iNI if more than one request share the same virtual channel, an intra-vc arbitration is required and if more than one virtual channel is used, an inter-vc arbitration is required as well. In addition to that a delay due to the memory access waiting time at the iNI. On the request side, packets going through the network components suffer from delay due to the several arbitration points as explained above. On the response path, once the memory starts producing data, the time spent for the response to get back at the master is a constant since the network is faster than the memory there is no contention in the network and the delay of the arbiters does not change for the iNI, S and tNI at the response path. Computation of the WCET of U-NIC depends on the implementation of the arbiters in the various arbitration points in the network interfaces and switches. For this specific study case we assume that round-robin arbitration is used to schedule the sharing of the network resources. We compute the latency according to the formulas provided in Chapter 2. The formulas are recalled hereunder:

$$N_{tsi} = \lceil \frac{S_i(byte)}{S_{ts}(byte)} \rceil$$

Where N_{tsi} is the number of time-slots required by request i, S_i is the size of request i in bytes and S_{ts} is the size of the time-slot in bytes. The number of time-slots required by request i is rounded up incase the size of the request is not a product of the time-slot size.

The latency of request i depends on its time-slots allocation on the TDMA time wheel. If the needed time-slots per request is a multiple of the allocated slots in the time-wheel, then the worst-case latency can be computed as follows:

$$L_i = (\frac{S_{tw}(slots)}{AL_{tsi}(slots)} \times N_{tsi}) \times N_{cc/ts}$$

Where L_i is the latency of request i, S_{tw} is the size of TDMA time-wheel in slots, AL_{tsi} is the allocated time-slots for task i and $N_{cc/ts}$ is the number of cc per time-slot.

Considering round-robin scheduling with 4 time-slots and 2 concurrent requests. The allocation of the time-slots is 2 per request per round. The size of the time-slot equals to the link bandwidth = 32-bit (4 bytes), the wheel turns every clock cycle 1 time-slot.

According to the application's communication patterns, VLD read and VLD write executes in parallel, therefore round-robin arbitration takes place at the tNI, S and iNI. Actor *tNIvld1* represents the delay for VLD read request due to the arbitration at the tNI. The read request has only a header but no real data (payload), the header size is 1 word (32-bit). The required number of time-slots for the VLD read request ($\frac{4bytes}{4bytes}$)= 1 slot. Accordingly, the latency at *tNIvld1* is ($\frac{4slots}{2slots} \times 1$) = 2 cc.

Actor *tNIvld3* represents the delay of VLD write requests due to the arbitration at the tNI. The write request size is 96 words. The required number of time-slots for the VLD write request ($\frac{96 \times 4bytes}{4bytes}$)= 96 slots. The latency at *tNIvld3* is ($\frac{4slots}{2slots} \times 96$) = 192 cc.

It can be concluded that the latency for IQ read request at *tNIiq1* is equal to the latency of *tNIvld1* and the latency of MC write request at *tNImc3* is equal to the latency of *tNIvld3*. In a similar way we compute the latency of the read and write request in the network and memory elements depending on the requests sizes. Table 6.2 provides the WCETs at the network arbitration points. The network runs at frequency 500 MHz, 1 cc = 2 nsec.

Execution times of the memory actors

Delay at the memory is the time from a read or write request from the network interface is accepted until the last data in the response has been returned to the network interface. We compute the latency at the memory for the read and write requests/responses at the memory according to the above formulas. We consider that the word size of the memory is also 32-bit (4 bytes). WCETs at the memory are provided in Table 6.2.

Table 6.2: **U-NIC and memory WCET**

On The Request Path	
Actor	WCET (nsec)
tNIvld1	4
Svld1	4
iNIvld1	4
Mvld1	38016
tNIvld3	384
Svld3	4
iNIvld3	4
Mvld3	4
tNIiq1	4
Siq1	4
iNIiq1	4
Miq1	32
tNImc3	384
Smc3	4
iNImc3	4
Mmc3	4
On The Response Path	
Actor	Execution Time (nsec)
tNIvld2	4
Svld2	4
iNIvld2	4
Mvld2	0
tNIvld4	4
Svld4	4
iNIvld4	4
Mvld4	0
tNIiq2	4
Siq2	4
iNIiq2	4
Miq2	0
tNImc4	4
Smc4	4
iNImc4	4
Mmc4	0

Chapter 7

Experimental results

In this chapter we provide the results of the throughput and latency analysis of the constructed SDFG model for the H.263 SoC performed by the SDF^3 tool.

7.1 Throughput analysis

The initial size of the video frame buffer that is allocated in the memory influences the application's execution pattern. If initially the frame buffer size equals to one video frame, the application will run as explained in the previous chapter. If we double the size of the initial frame buffer, we would allow the application actors to run in parallel, where the VLD will start processing the second video frame while the IQ, IDCT and MC are still processing the blocks of the first video frame. This in turn can lead to an increase in the throughput of the system [27].

We measure the throughput with the above frame buffer size variations. When the application's actors run in parallel (with initial frame buffer equal to two video frames), the WCETs of the network components and memory are doubled to reflect the resources sharing (each request will have 1 time-slot allocated in round-robin scheduler instead of 2 time-slots). The resulted throughput satisfies the H.263 throughput constraints of 15 frames/sec. It is noticeable that increasing the initial buffer size further will have no affect on the throughput value because the throughput of an SDFG is known to be limited by its critical cycle [23]. The execution times of the actors in the critical cycle influences the throughput computation as well, therefore increasing the buffer size further doesn't help in reducing the the maximal cycle mean of the graph. Throughput results are shown in Table 7.1.

7.2 Critical cycle

The critical cycle of the SDFG determines the achieved throughput as explained in Chapter 4. One way to find the critical cycle in our model is by changing the execution time of the application *execution actors* and analyze the throughput results after every change.

Table 7.1: Performance analysis

Frame bufer (Tokens)	Thr (frame/sec)	Latency (nsec)
9504	22.6	4.40646e+07
19008	30.0	5.14845e+07

Table 7.2: Critical cycle

WCET (nsec)	VLD	IQ	IDCT	MC
			Thr (frame/sec)	
Original value	22.6	22.6	22.6	22.6
+ 10%	**21.4**	22.6	22.6	22.1
+ 20%	**20.3**	22.6	22.6	21.6
+ 30%	**19.3**	22.6	22.6	21.1
+ 40%	**18.3**	22.6	22.6	20.6
+ 50%	**17.5**	22.6	22.6	20.2

We increase the WCET of one actor at a time in small steps 10% per step while keeping the WCET of the other actors unchanged, then run the throughput analysis algorithm. The results in Table 7.2 when compared to the results from the original throughput analysis (for the original WCET of the actors) indicate that the VLD is in the critical cycle, which means that if we can improve the throughput of the VLD sub-model the system's throughput will increase as well. One way of increasing the throughput of the VLD sub-model is by increasing the buffer size of the channel from *vld4* to *vldexe*. The current value is 1 token, by increasing the channel capacity to 2 tokens, we notice that the throughput of the system is increased due to the fact that the VLD execution actor can start processing the second Macro Block (MB) immediately without waiting for the first processed MB to be written in the shared memory. Tabel 7.3 provides the throughput results when the channel capacity is increased. This change in the channel capacity means that the buffer of the VLD processing element should be larger and accordingly an increase in the silicon area and power costs. Another way to improve the throughput of the critical cycle by reducing the latency in the path of the critical cycle. This can be done by for example allocating more time-slots for the VLD requests at the several arbitration points in the network and memory (weighted round-robin), however in this study case we did not explore this option.

7.3 Latency analysis

The latency computation depends on the execution times of the network components, the application main functions and the access time to the shared memory via the memory controller due to the arbitration between the requests that share the same network and memory resources. To minimize the system latency, the previously mentioned execution

Table 7.3: Performance analysis vs VLD channel capacity

VLD channel (Tokens)	Thr (frame/sec)	Latency (nsec)
1	22.6	4.40646e+07
2	24.7	4.04123e+07
3	24.7	4.04123e+07

times need to be reduced. The choice of the arbitration algorithms in the network and the memory controller is crucial for the latency computation. For example weighted round-robin allows the allocation of additional slots to the request that require high bandwidth, which can result in a latency reduction.
in Table 7.1 we show the latency computation between the actors *vld1* and *mc4* of the SDFG. It is noticeable that the latency increased when the initial frame buffer is equal to two video frames because the WCETs of the network and memory components are increased at the same time. In Tabel 7.3 the latency has decreased when the initial tokens are increased for the VLD sub-model on the channel from *vld4* to *vldexe*, this is due to the fact that the VLD processes the second MB without waiting for the first processed MB to be written to the memory.

In the next chapter we will construct a Latency-Rate model for the external memory controller to compute the latency bounds and provide more accurate worst case response time of the shared memory controller. We will also consider the restrictions on the burst size by the network and the memory controller.

Chapter 8

Latency-Rate SDFG model

In a multi-processor SoC tasks share resources. Some of the shared resources are off-chip, like the memory in the previous case study. Access to the shared memory is organized through a memory controller. The time required to access the shared memory depends on the previous requests scheduled at the memory controller, which makes it difficult to predict the latency at design time [9]. Latency-Rate analysis [25] which is a Network Calculus [13] provides a methodology for computing the delay accurately.
Latency-Rate (LR) schedulers can be modeled as a dataflow graph [30], therefore we will replace the memory actors in the previously constructed model with LR actors in a dataflow model so that we can give latency bounds when accessing the shared memory.

8.1 Latency-rate server

Latency-Rate servers are models of run-time schedulers used to analyze the traffic scheduling algorithms in packet networks by providing means to describe the worst case behavior of the scheduling algorithm [25]. LR servers allow the computation of tight upper bounds on latency, internal burstiness and buffer requirements in a heterogeneous network, where different scheduling algorithms and traffic models are supported [25]. TDMA and round-robin are examples of LR servers.

The behavior of a LR scheduler is determined by the *latency* θ and *allocated rate* ρ. The latency is the worst case delay experienced by the first packet in a request (word in a burst). This latency can be calculated from the latencies of the individual schedulers on the path of the packet through the network of schedulers which means that the choice of scheduling algorithms will have direct impact on the worst case delay computation. Low latency schedulers lead to lower delay of a request (burst) through the network of schedulers. In a LR server, the average rate of service offered by the scheduler to a request (burst) after the first experienced delay should be at least equal to the rate reserved for the request in the scheduler. Figure 8.1 provides an example of the LR server behavior. In other words, bandwidth guarantees should be given by the LR scheduler to a request after an initial delay.

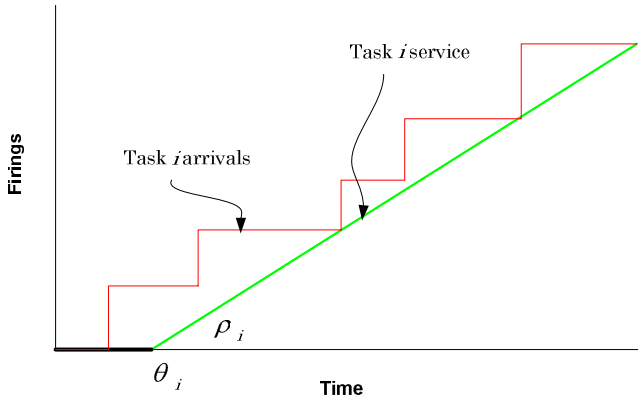

Figure 8.1: An example of a LR server behavior

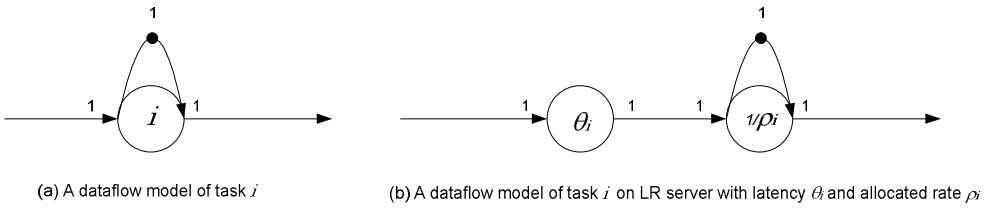

Figure 8.2: Dataflow actors of task i on LR server

Figure 8.2 shows the traditional dataflow graph model of a task executed on a network scheduler (TDMA for example) and the LR model of the same task when executed on a latency rate server. In the traditional dataflow model the worst-case latency due to the scheduler arbitration is computed for every word that is sent through the scheduler, which is a conservative model. In the LR model, the worst-case latency is experienced one time by the actor θ_i without self-edge, while actor ρ_i bounds the rate of which the data is sent. The self-edge with one token on the ρ_i actor provides a conservative bounds on the worst-case latency that is required to serve one word after an initial latency θ_i, which is the the period of the TDMA wheel divided by the reserved rate during this time period. In the next section we will explain how to compute the delay of a scheduler and the average rate reservation for a request.

8.2 Memory controller LR model

In section 6 we have constructed an SDFG where the memory is represented by one actor on the request side of the network and one actor on the response side of the network. In practice, the memory is accessed through a memory controller which can be a class of LR server depending on the used arbitration (e.g. TDMA, RR). We will replace the memory

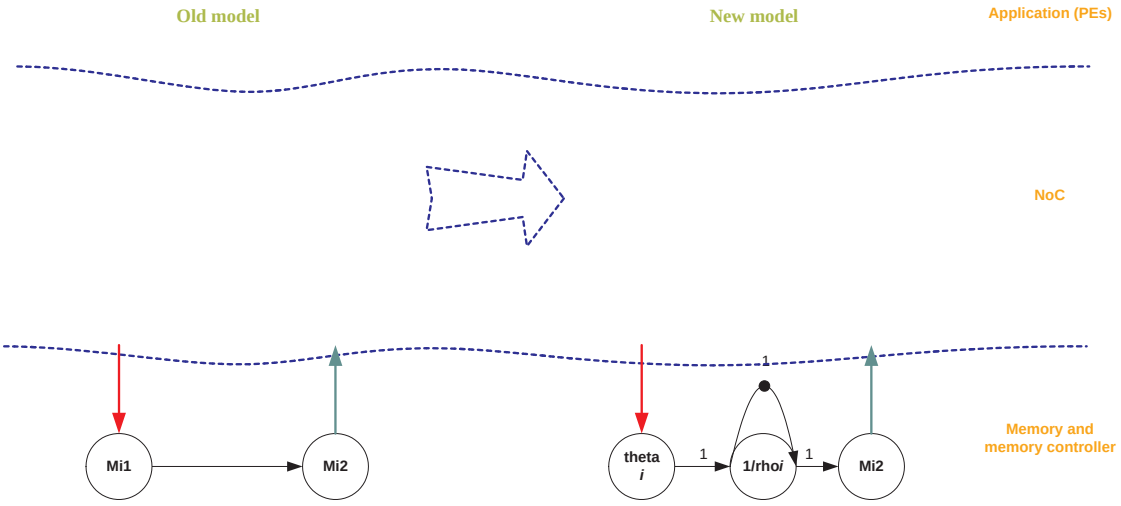

Figure 8.3: Memory controller LR SDFG model

actor on the request side with two actors one for the *latency* and one for the *rate*, in this way we have better model for the worst case behavior of the memory controller.

The *latency* actor has a worst case execution time θ_i that is the sum of all latencies found on the request path from the time a request is submitted to the memory by the network interface until the time the memory starts processing the request. The latency value θ_i for a request depends on the latencies of the individual schedulers on the path through the memory controller [25].
The *rate* actor represents the service/net bandwidth ρ_i allocated in the memory for an execution of a task i which represents the Read or Write request. This bandwidth is guaranteed after the initial service latency θ_i. The total number of allocated bandwidth of all the tasks that are scheduled to access the memory through the memory controller should not exceed the total bandwidth of the memory.
Figure 8.3 provides the old SDFG memory model and the new LR SDFG model of the memory controller. The absence of the self-edge on the latency node ensures that the latency will be experienced only one time per burst once a requestor is granted access to the memory. The self-edge on the rate actor ensures that only one token at a time will be consumed and produced. This is to avoid concurrency as explained earlier in this report.
In order to have an accurate computation of the *latency* and *rate* of the memory controller for the H.263 case study, we first need to study the memory controller architecture and the internal communication which have impact on the latency and rate computation. We will review the architecture of a Double Data Rate Synchronous Dynamic Random Access Memory controller (DDR SDRAM) similar to the memory controller developed by NXP known as ip_2032.

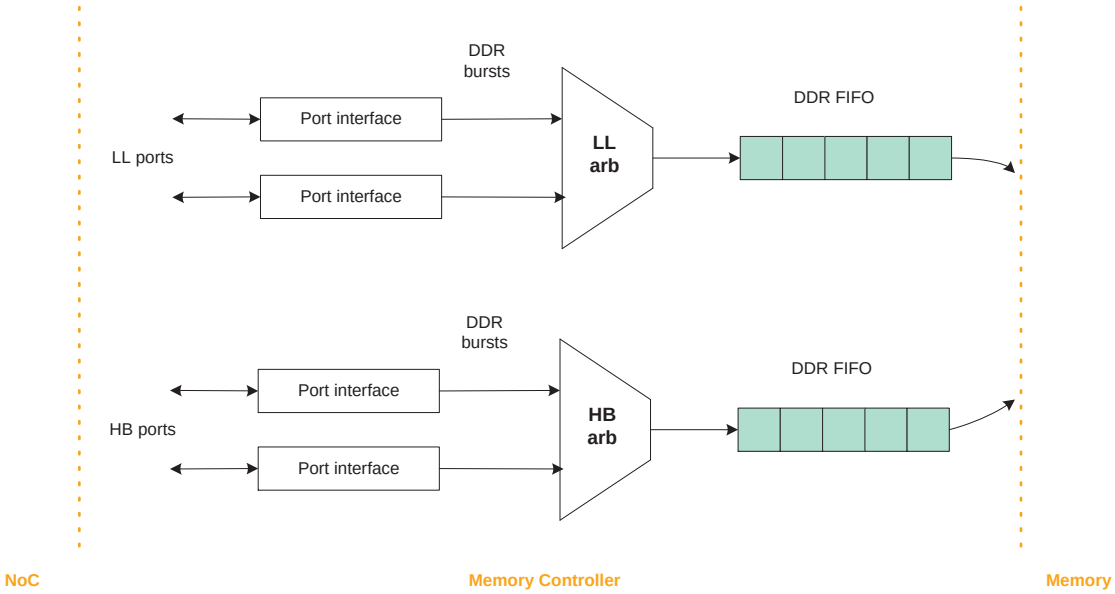

Figure 8.4: Memory controller partial block diagram

8.3 DDR SDRAM preliminaries

A double data rate synchronous dynamic memory has 4 banks [8] and achieves the double bandwidth of a single rate SDRAM since data is transferred on the rising and falling edges of the clock signal. The transfer rate of a DDR SDRAM is determined by its data width, often 32-bit or 64-bit, and its clock frequency. The transfer rate can be computed as follows:

$$\text{(number of bits transferred)} \times 2 \text{ (for double rate)} \times \text{(clock frequency)} \times \frac{1}{8(bits/byte)}$$

The SDRAM memory controller (ip_2032) has multiple ports for traffic class low latency (LL) and high bandwidth (HB). Each port has a buffer that can hold two (Read or Write) requests. The request size is restricted to 128-byte in order to use the memory efficiently [9]. There are two TDMA arbiters one to arbitrate between requests arriving at the LL ports and the other to arbitrate between requests arriving at the HB ports as illustrated in Figure 8.4. Inside the memory controller there are two DDR FIFO buffers where the accepted LL and HB requests are buffered after converting a request to DDR bursts of size 32-byte. Another arbiter exists after the DDR FIFO buffers to arbitrate between the LL and HB DDR requests. A 128-byte transaction is split into 4 DDR bursts, one on each bank of the SDRAM. This will explained in the next section.

8.4 Request path through U-NIC to memory controller

The read and write requests that are submitted by the PEs are passed to the memory through a memory controller which schedules the request access to the memory. These read/write requests from the PEs to the memory controller are passed through the network. At the network adapter, the requests are chopped to bursts of size 128-byte then sent to the tNI. The reason for shaping the requests to 128-byte bursts is to make use of all 4 banks in the SDRAM so that the preparation of one bank can overlap with the data transfer from other bank. This will ensure an efficient use of the shared SDRAM [9]. The time required for chopping requests depends on the size of the request. One clock cycle (cc) is required per 128 bytes.
At the tNI a 128-byte burst is converted to the network data-size of 32-bit (1 *word*). On the request path of U-NIC network. Afterwards, per clock cycle a *word* is sent towards the Switch (S) and buffered in a FIFO inside the switch. Then the *word* is passed to the iNI. At the iNI another conversion occurs to convert the network *words* to the original 128-byte request size and pass it to the memory. A maximum of two 128-byte requests can be buffered at the iNI. Once the request is sent to the memory controller by the iNI, it will be first buffered at either the LL port buffer or the HB port buffer depending on its traffic class. The port buffer can hold maximum two requests each of size 128-byte. In the memory controller, a request is divided to four DDR requests size 32-byte and then buffered in the DDR request FIFO buffer. The DDR request buffer can hold maximum of five DDR requests. Each DDR request is finally converted to four data units (for double data rate this is eight data units). The latency at the memory controller is due to the TDMA arbitration in case more than one port is activated. If requests access the same port, with the same traffic class, then Round Robin (RR) scheduling takes place. The latency that a request experiences at the memory controller can be computed as follows:

$$\theta_i = \sum_j \tau_i^j$$

Where θ_i is the memory controller latency of task i, τ_i is the latency at arbitration point/scheduler j.

Finally the DDR requests are passed to the SDRAM with transfer rate depending on the reserved bandwidth of the task ρ_i. The reserved rate and bandwidth can be computed as follows:

$$Tbw_i = \frac{Ts_i}{M_{bw}} \times 100\%$$

$$\rho_i = \frac{Tbw_i}{100} \times M_{tr}\%$$

Where T_{bwi} is the percentage of reserved bandwidth for task i in relation with the total memory bandwidth, Ts_i is the required memory allocation for task i in MB, M_{bw} is the memory bandwidth in MB/sec, ρ_i is the rate of task i and M_{tr} is the memory transfer rate.
Figure 8.5 provides an improved SDFG for the VLD Read request incorporating the

derived LR server model of the memory controller. To keep the graph readable, the port rate is mentioned only when the consumed or produced tokens is > 1. If no value is given to the port rate, then it is by default 1. The latency and rate computation will be discussed in the next section.

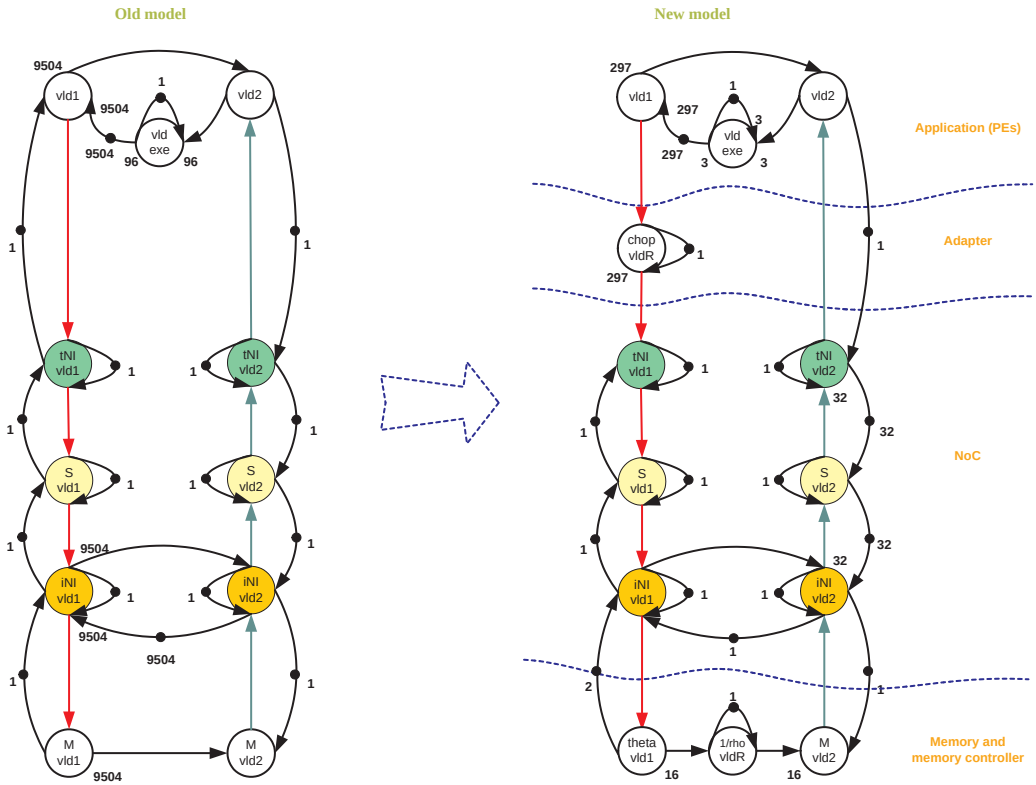

Figure 8.5: VLD Read Improved SDFG Model

Chapter 9

H263 Video Decoder Case Study - Revisited

In the first experiment we did not consider the memory controller access requirements related to the request size and number of requests accepted at each of the memory controller ports, which resulted in an approximate model. In this experiment, based on the details of the requests issued by the processing elements and how these requests pass through the network to the memory controller, we construct an improved SDFG model for the case study application. The full XML file of the improved SDFG model is enclosed in appendix 2.

9.1 Improved SDFG model for H263 video decoder

In this experiment, we apply the improved SDFG model to the H.263 case study as shown in Figure 9.1. The differences from the old model studied in Chapter 6 are mainly the LR model of the memory controller and the data-width conversion through the network. WCETs at the network and memory are computed in the previous case study.

We consider a 128-MB SDRAM that runs at 300 MHz and has 32-bit DDR data-width, thus the transfer rate is $(32 \times 2 \times 300 \times \frac{1}{8}) = 2400$ MB/sec and an SDRAM memory controller with only one port for LL traffic class where the requests are scheduled according to RR scheme.

Table 9.1: **Tasks transfer rate at SDRAM**

Actor	Memory Space (bytes)	% Reserved BW	Rate MB/sec
rho_{vldR}	38016	0.02800	0.6700
rho_{vldW}	384	0.00028	0.0067
rho_{iqR}	64	0.00005	0.0011
rho_{mcR}	384	0.00028	0.0067

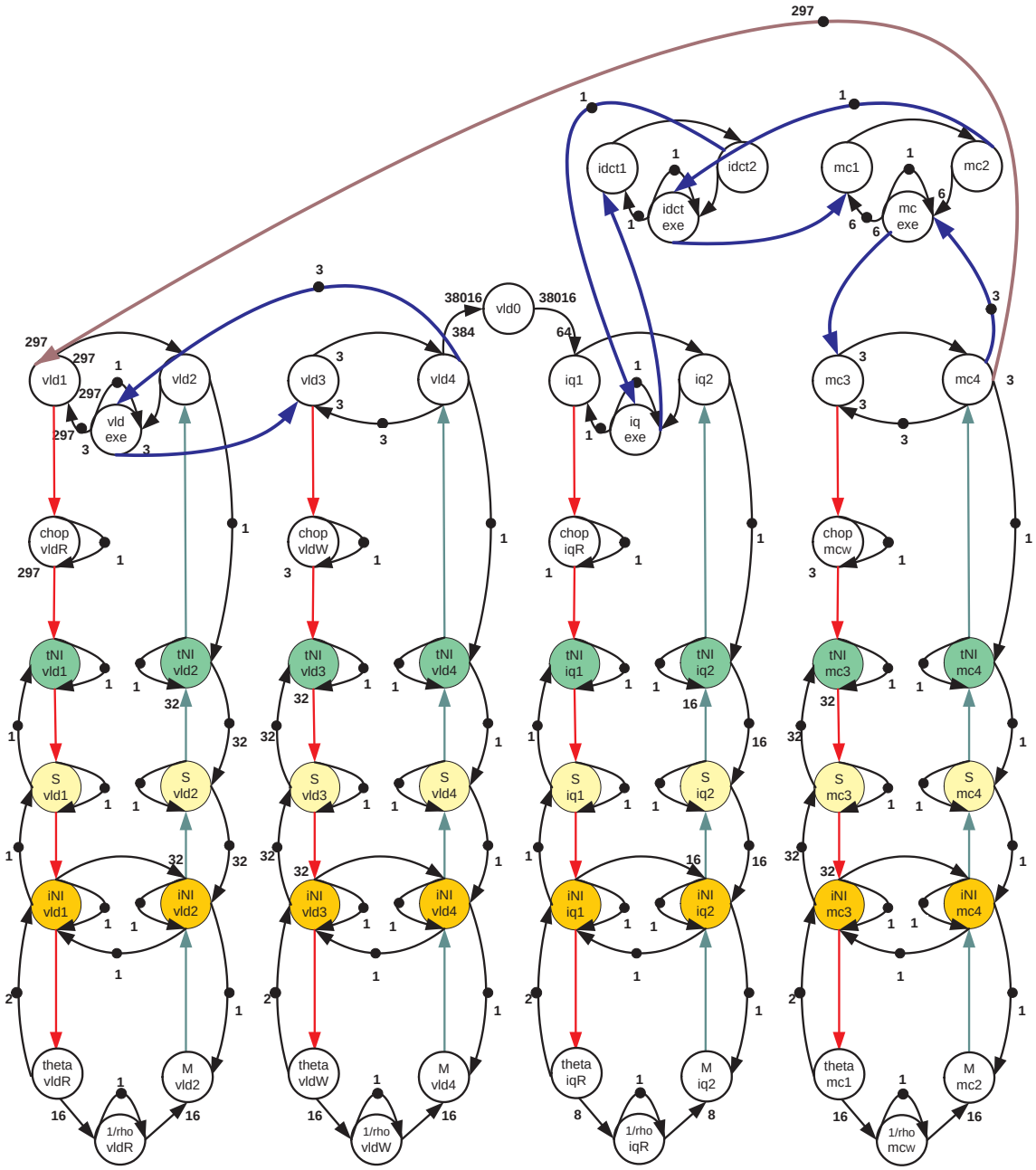

Figure 9.1: Improved SDFG Model

In the studied application, initially two tasks execute in parallel, and accordingly share the same network and memory resources. The communication scenario between the application main tasks that is explained in details in Chapter 6 show that VLD read and write

tasks perform their computation on the full video frame in parallel. Then the IQ, IDCT and MC start computation. The IDCT and MC read tasks do not send any requests to the shared memory controller through the network, there are no resources sharing from the network and memory by these tasks, whereas the IQ read and MC write tasks do share the network and memory resources. Since we use the base line platform of U-NIC, the requests to the memory go through the same iNI. As discussed previously, the iNI can hold only two 128-byte requests. Considering the request sizes of the H.263 tasks in Table 9.1, the choice of the parallel execution of two tasks at a time is reasonable when the tasks are mapped to one iNI. Of course we can have parallel execution of all tasks, but this can lead to delay increase due to the access limitations of the Memory Controller which is modeled by a dependency edge with two initial tokens from the latency actor (theta) to the iNI of every read and write request. Tabel 9.2 provides the WCET of the network actors including the network adapter for the NoC architecture used in the H.263 case study.
Table 9.1 provides the computation of the reserved bandwidth and rates of the main H.263 tasks for an SDRAM with transfer rate 2400 MB/sec and 128 MB bandwidth.
Table 9.4 gives the WCET of the memory controller. The WCET of the rate actors are considerably high due to the low reserved bandwidth. For a 128 MB SDRAM, only a small fraction (0.025%) of the memory bandwidth is reserved for the VLD read request, accordingly the transfer rate for the VLD read request is 0.67 MB/sec. In Table 9.3 the reserved bandwidth of each task is provided.
Knowing that in this specific case study only two tasks at a time execute in parallel, we can make more efficient use of the memory bandwidth and transfer rate. In the performance analysis results we will show how this suggestion will improve the latency computation results of the system.

9.2 Performance analysis results

We analyse the throughput and latency of the improved model similar to the previous experience. The performance results in Table 9.5 show that the throughput is decreased compared to the results from the first case study. This is due to the conservative memory allocation for the tasks which lead to very low transfer rate at the memory and accordingly the throughput is negatively affected. We obviously did not make efficient use of the total memory transfer rate.

Considering the concurrency in the system, we can relax the memory allocation by dividing the total bandwidth between the concurrent requests, while giving guarantees that none of the tasks will receive less than the required bandwidth.
In our model, two concurrent requests execute at a time, therefore we allocate 50% of the total transfer rate (2400 MB/sec) to each task.

$$\text{WCET}_{rho} = 50\% \times \tfrac{1}{2.4} \times 2 \text{ (concurrent requests)} = 1.66 \text{ nsec}$$

As a result the WCET of the rate actors are improved and therefore the throughput

Table 9.2: **Network actors WCET**

On The Request Path		
Actor	WCET (nsec)	Reasoning
$chop_{vldR}$	594	Request size 38016 bytes /128
$chop_{vldW}$	6	Request size 384 bytes /128
$chop_{iqR}$	2	Request size 64 bytes /128
$chop_{mcW}$	6	Request size 384 bytes /128
tNIvld1	4	
Svld1	4	
iNIvld1	4	
tNIvld3	384	
Svld3	4	
iNIvld3	4	
tNIiq1	4	
Siq1	4	
iNIiq1	4	
tNImc3	384	
Smc3	4	
iNImc3	4	
On The Response Path		
Actor	Execution Time (nsec)	Reasoning
tNIvld2	4	
Svld2	4	
iNIvld2	4	
Mvld2	0	
tNIvld4	4	
Svld4	4	
iNIvld4	4	
Mvld4	0	
tNIiq2	4	
Siq2	4	
iNIiq2	4	
Miq2	0	
tNImc4	4	
Smc4	4	
iNImc4	4	
Mmc4	0	

Table 9.3: **H.263 memory allocation**

Task	Memory Space (bytes)	Reasoning
vldR	38016	99 (MB) × 6 (blocks/MB) × 8×8 (block size) × 1 (byte/pixel)
vldW	384	6 (blocks/MB) × 8×8 (block size) × 1 (byte/pixel)
iqR	64	8×8 (block size) × 1 (byte/pixel)
mcR	384	6 (blocks/MB) × 8×8 (block size) × 1 (byte/pixel)

Table 9.4: **Memory controller WCET**

Actor	WCET (nsec)	Reasoning
$theta_{vldR}$	64	request size is 16 mem words
$theta_{vldW}$	64	request size is 16 mem words
$theta_{iqR}$	32	request size is 8 mem words
$theta_{mcR}$	64	request size is 16 mem words
$1/rho_{vldR}$	1492	Rate 0.6700 MB/sec
$1/rho_{vldW}$	149254	Rate 0.0067 MB/sec
$1/rho_{iqR}$	909090	Rate 0.0011 MB/sec
$1/rho_{mcW}$	149254	Rate 0.0067 MB/sec
M	0	This is *auxiliary actor*

results as shown in Table 9.6. When we doubled the initial frame buffer size, the latency increased due to the fact that the the WCETs at the network and memory components have been doubled to reflect the parallel execution of the code segments as explained in the previous section.

We notice that he throughput of the improved model is increased by 10.7 % while the latency is decreased by 11.4% compared to the model in the previous case study.

In [20], even a further refinement is presented to model a network channel by splitting the data forwarding in the network interface from the credits injection due to the flow control so that the latency due to credits injection is the sum of the delay caused from updating the credit counter on the data consumption and the delay until the credits are seen by the network interface. Then for each of the forward data and credit injection actors a LR model is constructed. Finally the latency due to the data routing and credits routing is added to the model. Via this refinement, more accurate bounds on latency are achieved for the NI especially when end-to-end flow control (between source and destination) is

Table 9.5: Performance analysis with conservative memory allocation

Frame buffer (byte)	Thr (frame/sec)	Latency (nsec)
9504	1	9.95904e+07

Table 9.6: Performance analysis with relaxed memory allocation

Frame buffer (byte)	Thr (frame/sec)	Latency (nsec)
9504	25.3	3.94896e+07
19008	38.5	3.96264e+07

implemented.
This method can be followed to model U-NIC channels and achieve more accurate upper bounds on latency. However, in U-NIC the flow control occurs between two consecutive components by means of returned tokens indicating the free buffer space after every time a network component fires and consumes token which frees space in its own buffer, therefore splitting the data forward from the credits injection can be ignored. In the presented case-study, we assumed that the flow control is instantaneous, However, the latency due to the flow control can be included in the computation of the latency at each of the network component.

Chapter 10

Summary and conclusion

The design of a NoC based system-on-chip for future multimedia applications for consumer electronic devices is a challenging task for the SoC engineers. System level performance analysis at the design time contributes to the design decisions at an early stage of the design traject and helps in reducing the implementation costs while giving the required quality of service and performance guarantees to the applications that are running on the system.

In this report, a method for modeling a multimedia NoC based system-on-chip is presented, which enables the performance analysis and state space exploration of the system at the design time. The method is based on the Synchronous Dataflow Graph model of computation (SoC); it allows the SoC designer to configure an application-specific NoC based system manually, build an SDFG model of the system and convert the model to an XML file that can be used later to analyse the performance of the system automatically using the capabilities of the SDF^3 tool. The performance analysis can help the designer to identify the bottleneck of the system by finding the critical cycle in the model and explore the trade-off between the system throughput and the buffer size in the path of the critical cycle to improve the throughput or by modifying the mapping and scheduling of the application tasks. Important perquisites in our method are the availability of the static analysis of the application to obtain a prior knowledge of the communication patterns and the worst case execution time of the application main functions; additionally, the delay of the network and memory elements is essential for the performance analysis of the system. The presented method can be placed at the top of a traditional design flow and can be used iteratively until the application performance constrains are met. Our method can be applied to any NoC and is independent of the platform architecture of the SoC.

The method is applied to a video decoder U-NIC based system-on-chip by constructing an SDFG of the system that is built out of basic sub-modules of the application, the network and the external memory then connecting the sub-models via dependency and sequence edges. A static scheduling is applied for the resource sharing according to the

application's communication patterns and traffic characteristics. A minimum buffer size is allocated for the network components that fulfills the application's requirements. Finally system's performance is analyzed to check if the the application's constraints are met. The performance analyses results show that it is possible to increase the throughput of the system by increasing the initial buffer size, but this can lead to an increase in the silicon area and power costs. The delay computation depends on the number of arbitration points and the implemented arbitration scheme(s) in case a mix of scheduling algorithms are used. Computation of the latency becomes more complex when levels of arbitration are implemented, like round-robin and priority scheduling schemes. TDMA as well as round-robin scheduling provide guarantees on throughput and latency for worst-case performance, however, TDMA is more restrictive than round-robin for average-case performance which makes round-robin favorite for average-case traffic.

The results are further improved by applying the Latency-Rate (LR) model to the memory controller that schedules the requests access to the shared memory. The LR model of the memory controller provides upper bounds on the latency that is experienced by a request and guarantees that a request will receive guaranteed service (memory bandwidth) after an initial delay. The throughput in the improved model is increased by 10.7 % while the latency is decreased by 11.4 %. Further refinement of the model could be achieved by applying the LR model to the network components which can result in even better performance results, but this is out of the scope of this report.
To conclude, the SDFG model of computation is a suitable method for modeling a multimedia NoC based system-on-chip and for worst-case performance analyses of the system via the SDF^3 tool. The worst-case execution times of the application and at each of the system components (network and memory) are crucial for performing theses analyses. Performance guarantees depends on the network and memory controller architectures, more precisely on the implementation of the arbitration and the number of arbiters in the path of a request through the network to the external shared memory. TDMA and round-robin schedulers provide upper bounds on latency, where round-robin makes more efficient use of the available resources for average-case traffic than TDMA schedulers.
Via the latency-rate model, the SDFG of a NoC based system-on-chip provides a realistic model for worst-case performance analyses. The latency in the latency-rate SDFG model is experienced only one time per burst, which results in a more accurate throughput and latency analyses of the system.

Recommendation for future work

- Automate the generation of the SDFG XML file;
- Build LR model for U-NIC interconnect.

Chapter 11

Practical Guide SDF3

11.1 Tool Installation

The software package of SDF3 tool can be downloaded from http://www.es.ele.tue.nl/sdf3/download.php
Download the latest version of the tool and check the 'README' file for instructions on
the installation. Documentation on the SDF3 tool can be found at http://www.es.ele.tue.nl/sdf3

11.2 SDFG to XML Transformation

The SDF3 tool accepts as input file in XML format. The SDFG model we created in Chapter 6 needs to be transformed to XML format so that the analysis algorithms provided by the tool can be executed.
Every actor shall have entries in the XML input file specifying the actor name, its input and output ports, ports names and rates. Next to the actor name, each actor needs to have a unique name. For example, actor vld1 and actor vldexe in the model constructed in Chapter 6 will have the following XML format:

<actor name="vld1" type="A">
<port name="p0" type="in" rate="1">
<port name="p1" type="out" rate="1">
<port name="p2" type="in" rate="9504">
<port name="p3" type="out" rate="9504">
<port name="p4" type="in" rate="9504">
</actor>
<actor name="vldexe" type="A">
<port name="p0" type="in" rate="96">
<port name="p1" type="out" rate="96">
<port name="p2" type="in" rate="1">
<port name="p3" type="out" rate="1">
<port name="p4" type="in" rate="1">
<port name="p5" type="out" rate="1">

</actor>
Every edge shall have one line in the XML input file referenced by channel. A channel connects 2 actors and has unique channel name, source actor, destination actor, source port, destination port and the number of initial tokens in case initial tokens are carried by the edge. For example the edge connecting actor vldexe with actor vld1 will have the following xml format:

<channel name="vldexe2vld1" srcActor="vldexe" srcPort="p1" dstActor="vld1" dstPort="p2" initialTokens='9504'/>

The name of the channel is descriptive to indicate which actors are communicating and the source/destination.

Each actor shall have sdf properties in the XML input file. The actors properties are listed under the tag <sdfProperties>. The property section consists of the actor name, processor type to indicate the execution time of the actor when running on that specific processor type and the execution time. For example, actor vldexe in our constructed model shall have the following actor properties:

<actorProperties actor="vldexe">
<processor type="arm" default="true">
<executionTime time="260180"/>
</processor>
</actorProperties>

11.3 Graph Consistency

Before computing the throughput and latency, it is essential to check the consistency of the graph. A graph is consistent if an actor produces insufficient number of tokens that allows the infinite execution of the graph. Consistency can be checked by executing the following command:

$ ../sdf3/build/release/Linux/bin/sdf3analysis-sdf –graph h263dec4unic.xml –algo consistency

If the above command returns 'Graph is not consistent', it is a good indication for checking the port rates.

Note
There will be no returned error when checking throughput due to inconsistency which makes it difficult to quickly find which port is causing the problem.

11.4 Throughput Analysis

Throughput is computed by the following command:

$ ../sdf3/build/release/Linux/bin/sdf3analysis-sdf –graph h263dec4unic.xml –algo throughput

Note
If the above command returns no results (hangs), it might be due to inconsistency in the graph. If the above command returns the value 0, it might be an indication to deadlock due to insufficient number of initial tokens. In this case check the edges that carry initial tokens and valuate the number of tokens against the port rate of the destination actor. The number of initial tokens should be >= the number of tokens required by the destination actor to fire. For example, actor mc3 cannot fire unless the number of initial tokens on the edge from tNI to mc3 is at least equal to 96, which is the number of fired tokens by mc3 for write request of 1 MB to the memory.

11.5 Latency Analysis

Latency between a source and a destination actor is computed by the following command:

$ ../sdf3/build/release/Linux/bin/sdf3analysis-sdf –graph h263dec4unic.xml –algo "latency (st,<src>,<dest>)"

st: Latency method Self Timed
src: Source actor
dest: Destination actor

Appendices

Appendix 1: XML file for initial SDFG model

```
<?xml version="1.0" encoding="UTF-8"?>
<sdf3 type="sdf" version="1.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="http://www.es.ele.tue.nl/sdf3/xsd/sdf3-sdf.xsd">
  <applicationGraph name='h263dec4unic'>
    <sdf name="h263dec4unic" type="H263dec4unic">

<!-- Application Actors -->
      <!-- VLD Function -->
      <actor name="vld1" type="A">
        <port name="p0" type="in" rate="1"/>
        <port name="p1" type="out" rate="1"/>
        <port name="p2" type="in" rate="9504"/>
        <port name="p3" type="out" rate="9504"/>
        <port name="p4" type="in" rate="9504"/>
      </actor>
      <actor name="vld2" type="A">
        <port name="p0" type="in" rate="1"/>
        <port name="p1" type="out" rate="1"/>
        <port name="p2" type="in" rate="1"/>
        <port name="p3" type="out" rate="1"/>
      </actor>
      <actor name="vldexe" type="A">
        <port name="p0" type="in" rate="96"/>
        <port name="p1" type="out" rate="96"/>
        <port name="p2" type="in" rate="1"/>
        <port name="p3" type="out" rate="1"/>
        <port name="p4" type="in" rate="1"/>
        <port name="p5" type="out" rate="1"/>
      </actor>
      <actor name="vld3" type="A">
        <port name="p0" type="in" rate="96"/>
        <port name="p1" type="out" rate="1"/>
        <port name="p2" type="in" rate="1"/>
        <port name="p3" type="out" rate="96"/>
        <port name="p4" type="in" rate="1"/>
      </actor>
      <actor name="vld4" type="A">
        <port name="p0" type="in" rate="1"/>
        <port name="p1" type="out" rate="1"/>
        <port name="p2" type="in" rate="1"/>
        <port name="p3" type="out" rate="1"/>
        <port name="p5" type="out" rate="1"/>
        <port name="p7" type="out" rate="96"/>
      </actor>
      <actor name="vld0" type="A">
```

```
    <port name="p0" type="in" rate="9504"/>
    <port name="p1" type="out" rate="9504"/>
  </actor>

  <!-- IQ Function -->
  <actor name="iq1" type="A">
    <port name="p0" type="in" rate="1"/>
    <port name="p1" type="out" rate="1"/>
    <port name="p2" type="in" rate="16"/>
    <port name="p3" type="out" rate="16"/>
<port name="p4" type="in" rate="16"/>
  </actor>
  <actor name="iq2" type="A">
    <port name="p0" type="in" rate="1"/>
    <port name="p1" type="out" rate="1"/>
    <port name="p2" type="in" rate="1"/>
    <port name="p3" type="out" rate="1"/>
  </actor>
  <actor name="iqexe" type="A">
    <port name="p0" type="in" rate="16"/>
    <port name="p1" type="out" rate="16"/>
    <port name="p2" type="in" rate="1"/>
    <port name="p3" type="out" rate="1"/>
<port name="p4" type="in" rate="16"/>
    <port name="p5" type="out" rate="16"/>
  </actor>

  <!-- IDCT Function -->
  <actor name="idct1" type="A">
    <port name="p0" type="in" rate="16"/>
    <port name="p1" type="out" rate="16"/>
    <port name="p2" type="in" rate="16"/>
  </actor>
  <actor name="idct2" type="A">
    <port name="p0" type="in" rate="16"/>
    <port name="p1" type="out" rate="16"/>
<port name="p3" type="out" rate="16"/>
  </actor>
  <actor name="idctexe" type="A">
    <port name="p0" type="in" rate="16"/>
    <port name="p1" type="out" rate="16"/>
    <port name="p2" type="in" rate="16"/>
    <port name="p3" type="out" rate="16"/>
  </actor>

  <!-- MC Function -->
  <actor name="mc1" type="A">
    <port name="p2" type="in" rate="16"/>
    <port name="p3" type="out" rate="16"/>
<port name="p4" type="in" rate="16"/>
  </actor>
    <actor name="mc2" type="A">
    <port name="p0" type="in" rate="16"/>
    <port name="p1" type="out" rate="16"/>
<port name="p3" type="out" rate="16"/>
  </actor>
  <actor name="mcexe" type="A">
    <port name="p0" type="in" rate="96"/>
    <port name="p1" type="out" rate="96"/>
    <port name="p2" type="in" rate="1"/>
    <port name="p3" type="out" rate="1"/>
    <port name="p4" type="in" rate="1"/>
```

```
        <port name="p5" type="out" rate="1"/>
      </actor>
      <actor name="mc3" type="A">
        <port name="p0" type="in" rate="96"/>
        <port name="p1" type="out" rate="1"/>
        <port name="p2" type="in" rate="1"/>
        <port name="p3" type="out" rate="96"/>
        <port name="p4" type="in" rate="1"/>
      </actor>
      <actor name="mc4" type="A">
        <port name="p0" type="in" rate="1"/>
        <port name="p1" type="out" rate="1"/>
        <port name="p2" type="in" rate="1"/>
        <port name="p3" type="out" rate="1"/>
        <port name="p5" type="out" rate="1"/>
    <port name="p7" type="out" rate="96"/>
      </actor>


<!-- NoC Actors -->
      <!-- VLD Function -->
      <actor name="tNIvld1" type="NI">
        <port name="p0" type="in" rate="1"/>
        <port name="p1" type="out" rate="1"/>
        <port name="p2" type="in" rate="1"/>
        <port name="p3" type="out" rate="1"/>
    <port name="p4" type="in" rate="1"/>
        <port name="p5" type="out" rate="1"/>
      </actor>
      <actor name="Svld1" type="S">
        <port name="p0" type="in" rate="1"/>
        <port name="p1" type="out" rate="1"/>
        <port name="p2" type="in" rate="1"/>
        <port name="p3" type="out" rate="1"/>
    <port name="p4" type="in" rate="1"/>
        <port name="p5" type="out" rate="1"/>
      </actor>
      <actor name="iNIvld1" type="NI">
        <port name="p0" type="in" rate="1"/>
        <port name="p1" type="out" rate="1"/>
        <port name="p2" type="in" rate="1"/>
        <port name="p3" type="out" rate="1"/>
    <port name="p4" type="in" rate="1"/>
        <port name="p5" type="out" rate="1"/>
        <port name="p6" type="in" rate="9504"/>
        <port name="p7" type="out" rate="9504"/>
      </actor>
      <actor name="tNIvld2" type="NI">
        <port name="p0" type="in" rate="1"/>
        <port name="p1" type="out" rate="1"/>
        <port name="p2" type="in" rate="1"/>
        <port name="p3" type="out" rate="1"/>
    <port name="p4" type="in" rate="1"/>
        <port name="p5" type="out" rate="1"/>
      </actor>
      <actor name="Svld2" type="S">
        <port name="p0" type="in" rate="1"/>
        <port name="p1" type="out" rate="1"/>
        <port name="p2" type="in" rate="1"/>
        <port name="p3" type="out" rate="1"/>
    <port name="p4" type="in" rate="1"/>
        <port name="p5" type="out" rate="1"/>
      </actor>
```

```
  <actor name="iNIvld2" type="NI">
    <port name="p0" type="in" rate="1"/>
    <port name="p1" type="out" rate="1"/>
    <port name="p2" type="in" rate="1"/>
    <port name="p3" type="out" rate="1"/>
<port name="p4" type="in" rate="1"/>
    <port name="p5" type="out" rate="1"/>
<port name="p6" type="in" rate="1"/>
    <port name="p7" type="out" rate="1"/>
  </actor>


  <actor name="tNIvld3" type="NI">
    <port name="p0" type="in" rate="1"/>
    <port name="p1" type="out" rate="1"/>
    <port name="p2" type="in" rate="1"/>
    <port name="p3" type="out" rate="1"/>
<port name="p4" type="in" rate="1"/>
    <port name="p5" type="out" rate="1"/>
  </actor>
  <actor name="Svld3" type="S">
    <port name="p0" type="in" rate="1"/>
    <port name="p1" type="out" rate="1"/>
    <port name="p2" type="in" rate="1"/>
    <port name="p3" type="out" rate="1"/>
<port name="p4" type="in" rate="1"/>
    <port name="p5" type="out" rate="1"/>
  </actor>
  <actor name="iNIvld3" type="NI">
    <port name="p0" type="in" rate="1"/>
    <port name="p1" type="out" rate="1"/>
    <port name="p2" type="in" rate="1"/>
    <port name="p3" type="out" rate="1"/>
<port name="p4" type="in" rate="1"/>
    <port name="p5" type="out" rate="1"/>
    <port name="p6" type="in" rate="1"/>
    <port name="p7" type="out" rate="1"/>
  </actor>
  <actor name="tNIvld4" type="NI">
    <port name="p0" type="in" rate="1"/>
    <port name="p1" type="out" rate="1"/>
    <port name="p2" type="in" rate="1"/>
    <port name="p3" type="out" rate="1"/>
<port name="p4" type="in" rate="1"/>
    <port name="p5" type="out" rate="1"/>
  </actor>
  <actor name="Svld4" type="S">
    <port name="p0" type="in" rate="1"/>
    <port name="p1" type="out" rate="1"/>
    <port name="p2" type="in" rate="1"/>
    <port name="p3" type="out" rate="1"/>
<port name="p4" type="in" rate="1"/>
    <port name="p5" type="out" rate="1"/>
  </actor>
  <actor name="iNIvld4" type="NI">
    <port name="p0" type="in" rate="1"/>
    <port name="p1" type="out" rate="1"/>
    <port name="p2" type="in" rate="1"/>
    <port name="p3" type="out" rate="1"/>
<port name="p4" type="in" rate="1"/>
    <port name="p5" type="out" rate="1"/>
<port name="p6" type="in" rate="96"/>
    <port name="p7" type="out" rate="96"/>
```

```
  </actor>

  <!-- IQ Function -->
   <actor name="tNIiq1" type="NI">
    <port name="p0" type="in" rate="1"/>
    <port name="p1" type="out" rate="1"/>
    <port name="p2" type="in" rate="1"/>
    <port name="p3" type="out" rate="1"/>
<port name="p4" type="in" rate="1"/>
    <port name="p5" type="out" rate="1"/>
  </actor>
  <actor name="Siq1" type="S">
    <port name="p0" type="in" rate="1"/>
    <port name="p1" type="out" rate="1"/>
    <port name="p2" type="in" rate="1"/>
    <port name="p3" type="out" rate="1"/>
<port name="p4" type="in" rate="1"/>
    <port name="p5" type="out" rate="1"/>
  </actor>
  <actor name="iNIiq1" type="NI">
    <port name="p0" type="in" rate="1"/>
    <port name="p1" type="out" rate="1"/>
    <port name="p2" type="in" rate="1"/>
    <port name="p3" type="out" rate="1"/>
<port name="p4" type="in" rate="1"/>
    <port name="p5" type="out" rate="1"/>
    <port name="p6" type="in" rate="16"/>
    <port name="p7" type="out" rate="16"/>
  </actor>
  <actor name="tNIiq2" type="NI">
    <port name="p0" type="in" rate="1"/>
    <port name="p1" type="out" rate="1"/>
    <port name="p2" type="in" rate="1"/>
    <port name="p3" type="out" rate="1"/>
<port name="p4" type="in" rate="1"/>
    <port name="p5" type="out" rate="1"/>
  </actor>
  <actor name="Siq2" type="S">
    <port name="p0" type="in" rate="1"/>
    <port name="p1" type="out" rate="1"/>
    <port name="p2" type="in" rate="1"/>
    <port name="p3" type="out" rate="1"/>
<port name="p4" type="in" rate="1"/>
    <port name="p5" type="out" rate="1"/>
  </actor>
  <actor name="iNIiq2" type="NI">
    <port name="p0" type="in" rate="1"/>
    <port name="p1" type="out" rate="1"/>
    <port name="p2" type="in" rate="1"/>
    <port name="p3" type="out" rate="1"/>
<port name="p4" type="in" rate="1"/>
    <port name="p5" type="out" rate="1"/>
<port name="p6" type="in" rate="1"/>
    <port name="p7" type="out" rate="1"/>
  </actor>

  <!-- MC Function -->
  <actor name="tNImc3" type="NI">
    <port name="p0" type="in" rate="1"/>
    <port name="p1" type="out" rate="1"/>
    <port name="p2" type="in" rate="1"/>
    <port name="p3" type="out" rate="1"/>
```

```
    <port name="p4" type="in" rate="1"/>
        <port name="p5" type="out" rate="1"/>
      </actor>
      <actor name="Smc3" type="S">
        <port name="p0" type="in" rate="1"/>
        <port name="p1" type="out" rate="1"/>
        <port name="p2" type="in" rate="1"/>
        <port name="p3" type="out" rate="1"/>
    <port name="p4" type="in" rate="1"/>
        <port name="p5" type="out" rate="1"/>
      </actor>
      <actor name="iNImc3" type="NI">
        <port name="p0" type="in" rate="1"/>
        <port name="p1" type="out" rate="1"/>
        <port name="p2" type="in" rate="1"/>
        <port name="p3" type="out" rate="1"/>
    <port name="p4" type="in" rate="1"/>
        <port name="p5" type="out" rate="1"/>
        <port name="p6" type="in" rate="1"/>
        <port name="p7" type="out" rate="1"/>
      </actor>
      <actor name="tNImc4" type="NI">
        <port name="p0" type="in" rate="1"/>
        <port name="p1" type="out" rate="1"/>
        <port name="p2" type="in" rate="1"/>
        <port name="p3" type="out" rate="1"/>
    <port name="p4" type="in" rate="1"/>
        <port name="p5" type="out" rate="1"/>
      </actor>
      <actor name="Smc4" type="S">
        <port name="p0" type="in" rate="1"/>
        <port name="p1" type="out" rate="1"/>
        <port name="p2" type="in" rate="1"/>
        <port name="p3" type="out" rate="1"/>
    <port name="p4" type="in" rate="1"/>
        <port name="p5" type="out" rate="1"/>
      </actor>
      <actor name="iNImc4" type="NI">
        <port name="p0" type="in" rate="1"/>
        <port name="p1" type="out" rate="1"/>
        <port name="p2" type="in" rate="1"/>
        <port name="p3" type="out" rate="1"/>
    <port name="p4" type="in" rate="1"/>
        <port name="p5" type="out" rate="1"/>
    <port name="p6" type="in" rate="96"/>
        <port name="p7" type="out" rate="96"/>
      </actor>


<!-- Memory Controller Actors -->
      <!-- VLD Function -->
      <actor name="Mvld1" type="M">
        <port name="p0" type="in" rate="1"/>
        <port name="p1" type="out" rate="1"/>
        <port name="p3" type="out" rate="9504"/>
      </actor>
      <actor name="Mvld2" type="M">
        <port name="p0" type="in" rate="1"/>
        <port name="p1" type="out" rate="1"/>
        <port name="p2" type="in" rate="1"/>
      </actor>
      <actor name="Mvld3" type="M">
        <port name="p1" type="out" rate="1"/>
```

```
      <port name="p2" type="in" rate="1"/>
      <port name="p3" type="out" rate="1"/>
    </actor>
    <actor name="Mvld4" type="M">
      <port name="p0" type="in" rate="96"/>
      <port name="p1" type="out" rate="1"/>
      <port name="p2" type="in" rate="1"/>
    </actor>
    <!-- IQ Function -->
    <actor name="Miq1" type="M">
      <port name="p0" type="in" rate="1"/>
      <port name="p1" type="out" rate="1"/>
      <port name="p3" type="out" rate="16"/>
    </actor>
    <actor name="Miq2" type="M">
      <port name="p0" type="in" rate="1"/>
      <port name="p1" type="out" rate="1"/>
      <port name="p2" type="in" rate="1"/>
    </actor>
    <!-- MC Function -->
    <actor name="Mmc3" type="M">
      <port name="p1" type="out" rate="1"/>
      <port name="p2" type="in" rate="1"/>
      <port name="p3" type="out" rate="1"/>
    </actor>
    <actor name="Mmc4" type="M">
      <port name="p0" type="in" rate="96"/>
      <port name="p1" type="out" rate="1"/>
      <port name="p2" type="in" rate="1"/>
    </actor>

<!-- Channels between Application Actors -->
<!-- VLD Function -->
<channel name="vld12vld2" srcActor="vld1" srcPort="p3" dstActor="vld2" dstPort="p0"/>
<channel name="vld22vldexe" srcActor="vld2" srcPort="p1" dstActor="vldexe" dstPort="p0"/>
<channel name="vldexe2vld1" srcActor="vldexe" srcPort="p1" dstActor="vld1" dstPort="p2" initialTokens='9504'/>
<channel name="vldexe2vldexe" srcActor="vldexe" srcPort="p3" dstActor="vldexe" dstPort="p2" initialTokens='1'/>
<channel name="vld32vld4" srcActor="vld3" srcPort="p1" dstActor="vld4" dstPort="p0" />
<channel name="vld42vld3" srcActor="vld4" srcPort="p1" dstActor="vld3" dstPort="p2" initialTokens='1'/>
<channel name="vldexe2vld3" srcActor="vldexe" srcPort="p5" dstActor="vld3" dstPort="p4"/>
<channel name="vld42vldexe" srcActor="vld4" srcPort="p5" dstActor="vldexe" dstPort="p4" initialTokens='1'/>
<!-- Sequence edges between VLD and IQ Function -->
<channel name="vld42vld0" srcActor="vld4" srcPort="p7" dstActor="vld0" dstPort="p0"/>
<channel name="vld02iq1" srcActor="vld0" srcPort="p1" dstActor="iq1" dstPort="p4"/>
<!-- IQ Function -->
<channel name="iq12iq2" srcActor="iq1" srcPort="p3" dstActor="iq2" dstPort="p0"/>
<channel name="iqexe2iq1" srcActor="iqexe" srcPort="p1" dstActor="iq1" dstPort="p2" initialTokens='16'/>
<channel name="iqexe2iqexe" srcActor="iqexe" srcPort="p3" dstActor="iqexe" dstPort="p2" initialTokens='1'/>
<channel name="iq22iqexe" srcActor="iq2" srcPort="p1" dstActor="iqexe" dstPort="p0"/>
<!-- Sequence edges between IQ and IDCT Function -->
<channel name="idct22iqexe" srcActor="idct2" srcPort="p3" dstActor="iqexe" dstPort="p4" initialTokens='16'/>
<channel name="iqexe2idct1" srcActor="iqexe" srcPort="p5" dstActor="idct1" dstPort="p2"/>
<!-- IDCT Function -->
<channel name="idct12idct2" srcActor="idct1" srcPort="p1" dstActor="idct2" dstPort="p0"/>
<channel name="idct22idctexe" srcActor="idct2" srcPort="p1" dstActor="idctexe" dstPort="p0"/>
<channel name="idctexe2idct1" srcActor="idctexe" srcPort="p1" dstActor="idct1" dstPort="p0" initialTokens='16'/>
<!-- Sequence edges between IDCT and MC Function -->
<channel name="idctexe2mc1" srcActor="idctexe" srcPort="p3" dstActor="mc1" dstPort="p4"/>
<channel name="mc22idctexe" srcActor="mc2" srcPort="p3" dstActor="idctexe" dstPort="p2" initialTokens='16'/>
<!-- MC Function -->
<channel name="mc12mc2" srcActor="mc1" srcPort="p3" dstActor="mc2" dstPort="p0"/>
<channel name="mc22mcexe" srcActor="mc2" srcPort="p1" dstActor="mcexe" dstPort="p0"/>
```

```
<channel name="mcexe2mc1" srcActor="mcexe" srcPort="p1" dstActor="mc1" dstPort="p2" initialTokens='96'/>
<channel name="mcexe2mcexe" srcActor="mcexe" srcPort="p3" dstActor="mcexe" dstPort="p2" initialTokens='1'/>
<channel name="mc32mc4" srcActor="mc3" srcPort="p1" dstActor="mc4" dstPort="p0" />
<channel name="mc42mc3" srcActor="mc4" srcPort="p1" dstActor="mc3" dstPort="p2" initialTokens='1'/>
<channel name="mcexe2mc3" srcActor="mcexe" srcPort="p5" dstActor="mc3" dstPort="p4"/>
<channel name="mc42mcexe" srcActor="mc4" srcPort="p5" dstActor="mcexe" dstPort="p4" initialTokens='1'/>
<!-- Sequence edges between MC and VLD Function -->
      <channel name="mc42vld1" srcActor="mc4" srcPort="p7" dstActor="vld1" dstPort="p4" initialTokens='9504'/>

<!-- Channels between the NoC Actors-->
<!-- VLD Read -->
<channel name="vld12tNIvld1" srcActor="vld1" srcPort="p1" dstActor="tNIvld1" dstPort="p0"/>
<channel name="tNIvld12Svld1" srcActor="tNIvld1" srcPort="p1" dstActor="Svld1" dstPort="p0"/>
<channel name="tNIvld12tNIvld1" srcActor="tNIvld1" srcPort="p3" dstActor="tNIvld1" dstPort="p2" initialTokens='1'/>
<channel name="tNIvld12vld1" srcActor="tNIvld1" srcPort="p5" dstActor="vld1" dstPort="p0" initialTokens='1'/>
<channel name="Svld12iNIvld1" srcActor="Svld1" srcPort="p1" dstActor="iNIvld1" dstPort="p0"/>
<channel name="Svld12Svld1" srcActor="Svld1" srcPort="p3" dstActor="Svld1" dstPort="p2" initialTokens='1'/>
<channel name="Svld12tNIvld1" srcActor="Svld1" srcPort="p5" dstActor="tNIvld1" dstPort="p4" initialTokens='1'/>
<channel name="iNIvld12iNIvld1" srcActor="iNIvld1" srcPort="p3" dstActor="iNIvld1" dstPort="p2" initialTokens='1'/>
<channel name="iNIvld22iNIvld1" srcActor="iNIvld2" srcPort="p7" dstActor="iNIvld1" dstPort="p6" initialTokens='9504'/>
<channel name="iNIvld12iNIvld2" srcActor="iNIvld1" srcPort="p7" dstActor="iNIvld2" dstPort="p6"/>
<channel name="iNIvld12Svld1" srcActor="iNIvld1" srcPort="p5" dstActor="Svld1" dstPort="p4" initialTokens='1'/>
<channel name="iNIvld22iNIvld2" srcActor="iNIvld2" srcPort="p3" dstActor="iNIvld2" dstPort="p2" initialTokens='1'/>
<channel name="iNIvld22Svld2" srcActor="iNIvld2" srcPort="p5" dstActor="Svld2" dstPort="p0"/>
<channel name="Svld22Svld2" srcActor="Svld2" srcPort="p3" dstActor="Svld2" dstPort="p2" initialTokens='1'/>
<channel name="Svld22iNIvld2" srcActor="Svld2" srcPort="p5" dstActor="iNIvld2" dstPort="p4" initialTokens='1'/>
<channel name="Svld22tNIvld2" srcActor="Svld2" srcPort="p1" dstActor="tNIvld2" dstPort="p0"/>
<channel name="tNIvld22tNIvld2" srcActor="tNIvld2" srcPort="p3" dstActor="tNIvld2" dstPort="p2" initialTokens='1'/>
<channel name="tNIvld22Svld2" srcActor="tNIvld2" srcPort="p5" dstActor="Svld2" dstPort="p4" initialTokens='1'/>
<channel name="tNIvld22vld2" srcActor="tNIvld2" srcPort="p1" dstActor="vld2" dstPort="p2"/>
<channel name="vld22tNIvld2" srcActor="vld2" srcPort="p3" dstActor="tNIvld2" dstPort="p4" initialTokens='1'/>
<!-- VLD Write -->
<channel name="vld32tNIvld3" srcActor="vld3" srcPort="p3" dstActor="tNIvld3" dstPort="p0"/>
<channel name="tNIvld32Svld3" srcActor="tNIvld3" srcPort="p1" dstActor="Svld3" dstPort="p0"/>
<channel name="tNIvld32tNIvld3" srcActor="tNIvld3" srcPort="p3" dstActor="tNIvld3" dstPort="p2" initialTokens='1'/>
<channel name="tNIvld32vld3" srcActor="tNIvld3" srcPort="p5" dstActor="vld3" dstPort="p0" initialTokens='96'/>
<channel name="Svld32iNIvld3" srcActor="Svld3" srcPort="p1" dstActor="iNIvld3" dstPort="p0"/>
<channel name="Svld32Svld3" srcActor="Svld3" srcPort="p3" dstActor="Svld3" dstPort="p2" initialTokens='1'/>
<channel name="Svld32tNIvld3" srcActor="Svld3" srcPort="p5" dstActor="tNIvld3" dstPort="p4" initialTokens='1'/>
<channel name="iNIvld32iNIvld3" srcActor="iNIvld3" srcPort="p3" dstActor="iNIvld3" dstPort="p2" initialTokens='1'/>
<channel name="iNIvld42iNIvld3" srcActor="iNIvld4" srcPort="p7" dstActor="iNIvld3" dstPort="p6" initialTokens='96'/>
<channel name="iNIvld32iNIvld4" srcActor="iNIvld3" srcPort="p7" dstActor="iNIvld4" dstPort="p6" />
<channel name="iNIvld32Svld3" srcActor="iNIvld3" srcPort="p5" dstActor="Svld3" dstPort="p4" initialTokens='1'/>
<channel name="iNIvld42iNIvld4" srcActor="iNIvld4" srcPort="p3" dstActor="iNIvld4" dstPort="p2" initialTokens='1'/>
<channel name="iNIvld42Svld4" srcActor="iNIvld4" srcPort="p1" dstActor="Svld4" dstPort="p0"/>
<channel name="Svld42Svld4" srcActor="Svld4" srcPort="p3" dstActor="Svld4" dstPort="p2" initialTokens='1'/>
<channel name="Svld42iNIvld4" srcActor="Svld4" srcPort="p5" dstActor="iNIvld4" dstPort="p4" initialTokens='1'/>
<channel name="Svld42tNIvld4" srcActor="Svld4" srcPort="p1" dstActor="tNIvld4" dstPort="p0"/>
<channel name="tNIvld42tNIvld4" srcActor="tNIvld4" srcPort="p3" dstActor="tNIvld4" dstPort="p2" initialTokens='1'/>
<channel name="tNIvld42Svld4" srcActor="tNIvld4" srcPort="p5" dstActor="Svld4" dstPort="p4" initialTokens='1'/>
<channel name="tNIvld42vld4" srcActor="tNIvld4" srcPort="p1" dstActor="vld4" dstPort="p2"/>
<channel name="vld42tNIvld4" srcActor="vld4" srcPort="p3" dstActor="tNIvld4" dstPort="p4" initialTokens='1'/>

<!-- IQ Read -->
<channel name="iq12tNIiq1" srcActor="iq1" srcPort="p1" dstActor="tNIiq1" dstPort="p0"/>
<channel name="tNIiq12Siq1" srcActor="tNIiq1" srcPort="p1" dstActor="Siq1" dstPort="p0"/>
<channel name="tNIiq12tNIiq1" srcActor="tNIiq1" srcPort="p3" dstActor="tNIiq1" dstPort="p2" initialTokens='1'/>
<channel name="tNIiq12iq1" srcActor="tNIiq1" srcPort="p5" dstActor="iq1" dstPort="p0" initialTokens='1'/>
<channel name="Siq12iNIiq1" srcActor="Siq1" srcPort="p1" dstActor="iNIiq1" dstPort="p0"/>
<channel name="Siq12Siq1" srcActor="Siq1" srcPort="p3" dstActor="Siq1" dstPort="p2" initialTokens='1'/>
<channel name="Siq12tNIiq1" srcActor="Siq1" srcPort="p5" dstActor="tNIiq1" dstPort="p4" initialTokens='1'/>
<channel name="iNIiq12iNIiq1" srcActor="iNIiq1" srcPort="p3" dstActor="iNIiq1" dstPort="p2" initialTokens='1'/>
```

```
<channel name="iNIiq22iNIiq1" srcActor="iNIiq2" srcPort="p7" dstActor="iNIiq1" dstPort="p6" initialTokens='16'/>
<channel name="iNIiq12iNIiq2" srcActor="iNIiq1" srcPort="p7" dstActor="iNIiq2" dstPort="p6"/>
<channel name="iNIiq12Siq1" srcActor="iNIiq1" srcPort="p5" dstActor="Siq1" dstPort="p4" initialTokens='1'/>
<channel name="iNIiq22iNIiq2" srcActor="iNIiq2" srcPort="p3" dstActor="iNIiq2" dstPort="p2" initialTokens='1'/>
<channel name="iNIiq22Siq2" srcActor="iNIiq2" srcPort="p5" dstActor="Siq2" dstPort="p0"/>
<channel name="Siq22Siq2" srcActor="Siq2" srcPort="p3" dstActor="Siq2" dstPort="p2" initialTokens='1'/>
<channel name="Siq22iNIiq2" srcActor="Siq2" srcPort="p5" dstActor="iNIiq2" dstPort="p4" initialTokens='1'/>
<channel name="Siq22tNIiq2" srcActor="Siq2" srcPort="p1" dstActor="tNIiq2" dstPort="p0"/>
<channel name="tNIiq22tNIiq2" srcActor="tNIiq2" srcPort="p3" dstActor="tNIiq2" dstPort="p2" initialTokens='1'/>
<channel name="tNIiq22Siq2" srcActor="tNIiq2" srcPort="p5" dstActor="Siq2" dstPort="p4" initialTokens='1'/>
<channel name="tNIiq22iq2" srcActor="tNIiq2" srcPort="p1" dstActor="iq2" dstPort="p2"/>
<channel name="iq22tNIiq2" srcActor="iq2" srcPort="p3" dstActor="tNIiq2" dstPort="p4" initialTokens='1'/>
<!-- MC Write -->
<channel name="mc32tNImc3" srcActor="mc3" srcPort="p3" dstActor="tNImc3" dstPort="p0"/>
<channel name="tNImc32Smc3" srcActor="tNImc3" srcPort="p1" dstActor="Smc3" dstPort="p0"/>
<channel name="tNImc32tNImc3" srcActor="tNImc3" srcPort="p3" dstActor="tNImc3" dstPort="p2" initialTokens='1'/>
<channel name="tNImc32mc3" srcActor="tNImc3" srcPort="p5" dstActor="mc3" dstPort="p0" initialTokens='96'/>
<channel name="Smc32iNImc3" srcActor="Smc3" srcPort="p1" dstActor="iNImc3" dstPort="p0"/>
<channel name="Smc32Smc3" srcActor="Smc3" srcPort="p3" dstActor="Smc3" dstPort="p2" initialTokens='1'/>
<channel name="Smc32tNImc3" srcActor="Smc3" srcPort="p5" dstActor="tNImc3" dstPort="p4" initialTokens='1'/>
<channel name="iNImc32iNImc3" srcActor="iNImc3" srcPort="p3" dstActor="iNImc3" dstPort="p2" initialTokens='1'/>
<channel name="iNImc42iNImc3" srcActor="iNImc4" srcPort="p7" dstActor="iNImc3" dstPort="p6" initialTokens='96'/>
<channel name="iNImc32iNImc4" srcActor="iNImc3" srcPort="p7" dstActor="iNImc4" dstPort="p6" />
<channel name="iNImc32Smc3" srcActor="iNImc3" srcPort="p5" dstActor="Smc3" dstPort="p4" initialTokens='1'/>
<channel name="iNImc42iNImc4" srcActor="iNImc4" srcPort="p3" dstActor="iNImc4" dstPort="p2" initialTokens='1'/>
<channel name="iNImc42Smc4" srcActor="iNImc4" srcPort="p1" dstActor="Smc4" dstPort="p0"/>
<channel name="Smc42Smc4" srcActor="Smc4" srcPort="p3" dstActor="Smc4" dstPort="p2" initialTokens='1'/>
<channel name="Smc42iNImc4" srcActor="Smc4" srcPort="p5" dstActor="iNImc4" dstPort="p4" initialTokens='1'/>
<channel name="Smc42tNImc4" srcActor="Smc4" srcPort="p1" dstActor="tNImc4" dstPort="p0"/>
<channel name="tNImc42tNImc4" srcActor="tNImc4" srcPort="p3" dstActor="tNImc4" dstPort="p2" initialTokens='1'/>
<channel name="tNImc42Smc4" srcActor="tNImc4" srcPort="p5" dstActor="Smc4" dstPort="p4" initialTokens='1'/>
<channel name="tNImc42mc4" srcActor="tNImc4" srcPort="p1" dstActor="mc4" dstPort="p2"/>
<channel name="mc42tNImc4" srcActor="mc4" srcPort="p3" dstActor="tNImc4" dstPort="p4" initialTokens='1'/>

<!-- Channels between NoC and Memory Controller -->
<!-- VLD Read Function -->
<channel name="iNIvld12Mvld1" srcActor="iNIvld1" srcPort="p1" dstActor="Mvld1" dstPort="p0"/>
<channel name="Mvld12iNIvld1" srcActor="Mvld1" srcPort="p1" dstActor="iNIvld1" dstPort="p4" initialTokens='1'/>
<channel name="Mvld22iNIvld2" srcActor="Mvld2" srcPort="p1" dstActor="iNIvld2" dstPort="p0"/>
<channel name="iNIvld22Mvld2" srcActor="iNIvld2" srcPort="p1" dstActor="Mvld2" dstPort="p2" initialTokens='1'/>
<!-- VLD Write Function -->
<channel name="iNIvld32Mvld3" srcActor="iNIvld3" srcPort="p1" dstActor="Mvld3" dstPort="p2"/>
<channel name="Mvld32iNIvld3" srcActor="Mvld3" srcPort="p3" dstActor="iNIvld3" dstPort="p4" initialTokens='1'/>
<channel name="Mvld42iNIvld4" srcActor="Mvld4" srcPort="p1" dstActor="iNIvld4" dstPort="p0"/>
<channel name="iNIvld42Mvld4" srcActor="iNIvld4" srcPort="p5" dstActor="Mvld4" dstPort="p2" initialTokens='1'/>
<!-- IQ Read Function -->
<channel name="iNIiq12Miq1" srcActor="iNIiq1" srcPort="p1" dstActor="Miq1" dstPort="p0"/>
<channel name="Miq12iNIiq1" srcActor="Miq1" srcPort="p1" dstActor="iNIiq1" dstPort="p4" initialTokens='1'/>
<channel name="Miq22iNIiq2" srcActor="Miq2" srcPort="p1" dstActor="iNIiq2" dstPort="p0"/>
<channel name="iNIiq22Miq2" srcActor="iNIiq2" srcPort="p1" dstActor="Miq2" dstPort="p2" initialTokens='1'/>
<!-- MC Write Function -->
<channel name="iNImc32Mmc3" srcActor="iNImc3" srcPort="p1" dstActor="Mmc3" dstPort="p2"/>
<channel name="Mmc32iNImc3" srcActor="Mmc3" srcPort="p3" dstActor="iNImc3" dstPort="p4" initialTokens='1'/>
<channel name="Mmc42iNImc4" srcActor="Mmc4" srcPort="p1" dstActor="iNImc4" dstPort="p0"/>
<channel name="iNImc42Mmc4" srcActor="iNImc4" srcPort="p5" dstActor="Mmc4" dstPort="p2" initialTokens='1'/>

<!-- Channels of the Memory Controller -->
<!-- VLD Read Function -->
<channel name="Mvld12Mvld2" srcActor="Mvld1" srcPort="p3" dstActor="Mvld2" dstPort="p0"/>
<!-- VLD Write Function -->
<channel name="Mvld32Mvld4" srcActor="Mvld3" srcPort="p1" dstActor="Mvld4" dstPort="p0"/>
<!-- IQ Read Function -->
```

```
<channel name="Miq12Miq2" srcActor="Miq1" srcPort="p3" dstActor="Miq2" dstPort="p0"/>
<!-- MC Write Function -->
<channel name="Mmc32Mmc4" srcActor="Mmc3" srcPort="p1" dstActor="Mmc4" dstPort="p0"/>

    </sdf>

   <sdfProperties>
<!-- SDF Properties VLD Function -->
      <actorProperties actor="vld1">
        <processor type="arm" default="true">
          <executionTime time="0"/>
        </processor>
      </actorProperties>
      <actorProperties actor="vld2">
        <processor type="arm" default="true">
          <executionTime time="0"/>
        </processor>
      </actorProperties>
      <actorProperties actor="vldexe">
        <processor type="arm" default="true">
          <executionTime time="260180"/>
        </processor>
      </actorProperties>
      <actorProperties actor="vld3">
        <processor type="arm" default="true">
          <executionTime time="0"/>
        </processor>
      </actorProperties>
      <actorProperties actor="vld4">
        <processor type="arm" default="true">
          <executionTime time="0"/>
        </processor>
      </actorProperties>
      <actorProperties actor="vld0">
        <processor type="arm" default="true">
          <executionTime time="0"/>
        </processor>
      </actorProperties>


<!-- SDF Properties IQ Function -->
      <actorProperties actor="iq1">
        <processor type="arm" default="true">
          <executionTime time="0"/>
        </processor>
      </actorProperties>
      <actorProperties actor="iq2">
        <processor type="arm" default="true">
          <executionTime time="0"/>
        </processor>
      </actorProperties>
      <actorProperties actor="iqexe">
        <processor type="arm" default="true">
          <executionTime time="5590"/>
        </processor>
      </actorProperties>
<!-- SDF Properties IDCT Function -->
      <actorProperties actor="idct1">
        <processor type="arm" default="true">
          <executionTime time="0"/>
        </processor>
      </actorProperties>
      <actorProperties actor="idct2">
```

```
        <processor type="arm" default="true">
          <executionTime time="0"/>
        </processor>
      </actorProperties>
      <actorProperties actor="idctexe">
        <processor type="arm" default="true">
          <executionTime time="4860"/>
        </processor>
      </actorProperties>
<!-- SDF Properties MC Function -->
      <actorProperties actor="mc1">
        <processor type="arm" default="true">
          <executionTime time="0"/>
        </processor>
      </actorProperties>
      <actorProperties actor="mc2">
        <processor type="arm" default="true">
          <executionTime time="0"/>
        </processor>
      </actorProperties>
      <actorProperties actor="mcexe">
        <processor type="arm" default="true">
          <executionTime time="109580"/>
        </processor>
      </actorProperties>
      <actorProperties actor="mc3">
        <processor type="arm" default="true">
          <executionTime time="0"/>
        </processor>
      </actorProperties>
      <actorProperties actor="mc4">
        <processor type="arm" default="true">
          <executionTime time="0"/>
        </processor>
      </actorProperties>


<!-- SDF Properties NoC -->
      <!-- VLD Function -->
      <actorProperties actor="tNIvld1">
        <processor type="arm" default="true">
          <executionTime time="4"/>
        </processor>
      </actorProperties>
      <actorProperties actor="Svld1">
        <processor type="arm" default="true">
          <executionTime time="4"/>
        </processor>
      </actorProperties>
      <actorProperties actor="iNIvld1">
        <processor type="arm" default="true">
          <executionTime time="4"/>
        </processor>
      </actorProperties>
      <actorProperties actor="Mvld1">
        <processor type="arm" default="true">
          <executionTime time="39016"/>
        </processor>
      </actorProperties>
      <actorProperties actor="tNIvld2">
        <processor type="arm" default="true">
          <executionTime time="4"/>
        </processor>
```

```
</actorProperties>
<actorProperties actor="Svld2">
  <processor type="arm" default="true">
    <executionTime time="4"/>
  </processor>
</actorProperties>
<actorProperties actor="iNIvld2">
  <processor type="arm" default="true">
    <executionTime time="4"/>
  </processor>
</actorProperties>
<actorProperties actor="Mvld2">
  <processor type="arm" default="true">
    <executionTime time="0"/>
  </processor>
</actorProperties>

<actorProperties actor="tNIvld3">
  <processor type="arm" default="true">
    <executionTime time="384"/>
  </processor>
</actorProperties>
<actorProperties actor="Svld3">
  <processor type="arm" default="true">
    <executionTime time="4"/>
  </processor>
</actorProperties>
<actorProperties actor="iNIvld3">
  <processor type="arm" default="true">
    <executionTime time="4"/>
  </processor>
</actorProperties>
<actorProperties actor="Mvld3">
  <processor type="arm" default="true">
    <executionTime time="4"/>
  </processor>
</actorProperties>
<actorProperties actor="tNIvld4">
  <processor type="arm" default="true">
    <executionTime time="4"/>
  </processor>
</actorProperties>
<actorProperties actor="Svld4">
  <processor type="arm" default="true">
    <executionTime time="4"/>
  </processor>
</actorProperties>
<actorProperties actor="iNIvld4">
  <processor type="arm" default="true">
    <executionTime time="4"/>
  </processor>
</actorProperties>
<actorProperties actor="Mvld4">
  <processor type="arm" default="true">
    <executionTime time="0"/>
  </processor>
</actorProperties>

<!-- IQ Function -->
<actorProperties actor="tNIiq1">
  <processor type="arm" default="true">
    <executionTime time="4"/>
```

```
  </processor>
</actorProperties>
<actorProperties actor="Siq1">
  <processor type="arm" default="true">
    <executionTime time="4"/>
  </processor>
</actorProperties>
<actorProperties actor="iNIiq1">
  <processor type="arm" default="true">
    <executionTime time="4"/>
  </processor>
</actorProperties>
<actorProperties actor="Miq1">
  <processor type="arm" default="true">
    <executionTime time="32"/>
  </processor>
</actorProperties>
 <actorProperties actor="tNIiq2">
  <processor type="arm" default="true">
    <executionTime time="4"/>
  </processor>
</actorProperties>
<actorProperties actor="Siq2">
  <processor type="arm" default="true">
    <executionTime time="4"/>
  </processor>
</actorProperties>
<actorProperties actor="iNIiq2">
  <processor type="arm" default="true">
    <executionTime time="4"/>
  </processor>
</actorProperties>
<actorProperties actor="Miq2">
  <processor type="arm" default="true">
    <executionTime time="0"/>
  </processor>
</actorProperties>


<!-- MC Function -->
 <actorProperties actor="tNImc3">
  <processor type="arm" default="true">
    <executionTime time="384"/>
  </processor>
</actorProperties>
<actorProperties actor="Smc3">
  <processor type="arm" default="true">
    <executionTime time="4"/>
  </processor>
</actorProperties>
<actorProperties actor="iNImc3">
  <processor type="arm" default="true">
    <executionTime time="4"/>
  </processor>
</actorProperties>
<actorProperties actor="Mmc3">
  <processor type="arm" default="true">
    <executionTime time="4"/>
  </processor>
</actorProperties>
<actorProperties actor="tNImc4">
  <processor type="arm" default="true">
    <executionTime time="4"/>
```

```
        </processor>
      </actorProperties>
      <actorProperties actor="Smc4">
        <processor type="arm" default="true">
          <executionTime time="4"/>
        </processor>
      </actorProperties>
      <actorProperties actor="iNImc4">
        <processor type="arm" default="true">
          <executionTime time="4"/>
        </processor>
      </actorProperties>
      <actorProperties actor="Mmc4">
        <processor type="arm" default="true">
          <executionTime time="0"/>
        </processor>
      </actorProperties>

    </sdfProperties>
  </applicationGraph>
</sdf3>
```

Appendices

Appendix 2: XML file for improved SDFG model

```
<?xml version="1.0" encoding="UTF-8"?>
<sdf3 type="sdf" version="1.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="http://www.es.ele.tue.nl/sdf3/xsd/sdf3-sdf.xsd">
  <applicationGraph name='h263dec4unic'>
    <sdf name="h263dec4unic" type="H263dec4unic">

<!-- Application Actors -->
      <!-- VLD Function -->
      <actor name="vld1" type="A">
        <port name="p1" type="out" rate="1"/>
        <port name="p2" type="in" rate="297"/>
        <port name="p3" type="out" rate="297"/>
        <port name="p4" type="in" rate="297"/>
      </actor>
      <actor name="vld2" type="A">
        <port name="p0" type="in" rate="1"/>
        <port name="p1" type="out" rate="1"/>
        <port name="p2" type="in" rate="1"/>
        <port name="p3" type="out" rate="1"/>
      </actor>
      <actor name="vldexe" type="A">
        <port name="p0" type="in" rate="3"/>
        <port name="p1" type="out" rate="3"/>
        <port name="p2" type="in" rate="1"/>
        <port name="p3" type="out" rate="1"/>
        <port name="p4" type="in" rate="3"/>
        <port name="p5" type="out" rate="1"/>
      </actor>
      <actor name="vld3" type="A">
        <port name="p1" type="out" rate="1"/>
        <port name="p2" type="in" rate="3"/>
        <port name="p3" type="out" rate="3"/>
        <port name="p4" type="in" rate="1"/>
      </actor>
      <actor name="vld4" type="A">
        <port name="p0" type="in" rate="1"/>
        <port name="p1" type="out" rate="1"/>
        <port name="p2" type="in" rate="1"/>
        <port name="p3" type="out" rate="1"/>
        <port name="p5" type="out" rate="1"/>
        <port name="p7" type="out" rate="128"/>
      </actor>
      <actor name="vld0" type="A">
        <port name="p0" type="in" rate="38016"/>
        <port name="p1" type="out" rate="38016"/>
```

```
  </actor>

  <!-- IQ Function -->
  <actor name="iq1" type="A">
    <port name="p1" type="out" rate="1"/>
    <port name="p2" type="in" rate="1"/>
    <port name="p3" type="out" rate="1"/>
<port name="p4" type="in" rate="64"/>
  </actor>
  <actor name="iq2" type="A">
    <port name="p0" type="in" rate="1"/>
    <port name="p1" type="out" rate="1"/>
    <port name="p2" type="in" rate="1"/>
    <port name="p3" type="out" rate="1"/>
  </actor>
  <actor name="iqexe" type="A">
    <port name="p0" type="in" rate="1"/>
    <port name="p1" type="out" rate="1"/>
    <port name="p2" type="in" rate="1"/>
    <port name="p3" type="out" rate="1"/>
    <port name="p4" type="in" rate="1"/>
    <port name="p5" type="out" rate="1"/>
  </actor>

  <!-- IDCT Function -->
  <actor name="idct1" type="A">
    <port name="p2" type="in" rate="1"/>
    <port name="p3" type="out" rate="1"/>
<port name="p4" type="in" rate="1"/>
  </actor>
  <actor name="idct2" type="A">
    <port name="p0" type="in" rate="1"/>
    <port name="p1" type="out" rate="1"/>
    <port name="p3" type="out" rate="1"/>
  </actor>
  <actor name="idctexe" type="A">
    <port name="p0" type="in" rate="1"/>
    <port name="p1" type="out" rate="1"/>
<port name="p4" type="in" rate="1"/>
    <port name="p5" type="out" rate="1"/>
  </actor>

  <!-- MC Function -->
  <actor name="mc1" type="A">
    <port name="p2" type="in" rate="1"/>
    <port name="p3" type="out" rate="1"/>
<port name="p4" type="in" rate="1"/>
  </actor>
  <actor name="mc2" type="A">
    <port name="p0" type="in" rate="1"/>
    <port name="p1" type="out" rate="1"/>
    <port name="p3" type="out" rate="1"/>
  </actor>
  <actor name="mcexe" type="A">
    <port name="p0" type="in" rate="6"/>
    <port name="p1" type="out" rate="6"/>
    <port name="p2" type="in" rate="1"/>
    <port name="p3" type="out" rate="1"/>
<port name="p4" type="in" rate="3"/>
    <port name="p5" type="out" rate="1"/>
  </actor>
  <actor name="mc3" type="A">
```

```
        <port name="p1" type="out" rate="1"/>
        <port name="p2" type="in" rate="3"/>
        <port name="p3" type="out" rate="3"/>
    <port name="p4" type="in" rate="1"/>
      </actor>
      <actor name="mc4" type="A">
        <port name="p0" type="in" rate="1"/>
        <port name="p1" type="out" rate="1"/>
        <port name="p2" type="in" rate="1"/>
        <port name="p3" type="out" rate="1"/>
        <port name="p5" type="out" rate="1"/>
        <port name="p7" type="out" rate="1"/>
      </actor>

<!-- Processor NW Adapter -->
      <!-- VLD Function -->
      <actor name="chopvldR" type="AO">
        <port name="p0" type="in" rate="1"/>
        <port name="p1" type="out" rate="297"/>
    <port name="p2" type="in" rate="1"/>
        <port name="p3" type="out" rate="1"/>
      </actor>
      <actor name="chopvldW" type="AO">
        <port name="p0" type="in" rate="1"/>
        <port name="p1" type="out" rate="3"/>
    <port name="p2" type="in" rate="1"/>
        <port name="p3" type="out" rate="1"/>
      </actor>
      <!-- IQ Function -->
      <actor name="chopiqR" type="AO">
        <port name="p0" type="in" rate="1"/>
        <port name="p1" type="out" rate="1"/>
    <port name="p2" type="in" rate="1"/>
        <port name="p3" type="out" rate="1"/>
      </actor>
      <!-- MC Function -->
      <actor name="chopmcW" type="AO">
        <port name="p0" type="in" rate="1"/>
        <port name="p1" type="out" rate="3"/>
    <port name="p2" type="in" rate="1"/>
        <port name="p3" type="out" rate="1"/>
      </actor>

<!-- NoC Actors -->
      <!-- VLD Function -->
      <actor name="tNIvld1" type="NI">
        <port name="p0" type="in" rate="1"/>
        <port name="p1" type="out" rate="1"/>
        <port name="p2" type="in" rate="1"/>
        <port name="p3" type="out" rate="1"/>
    <port name="p4" type="in" rate="1"/>
      </actor>
      <actor name="Svld1" type="S">
        <port name="p0" type="in" rate="1"/>
        <port name="p1" type="out" rate="1"/>
        <port name="p2" type="in" rate="1"/>
        <port name="p3" type="out" rate="1"/>
    <port name="p4" type="in" rate="1"/>
        <port name="p5" type="out" rate="1"/>
      </actor>
      <actor name="iNIvld1" type="NI">
        <port name="p0" type="in" rate="1"/>
```

```
    <port name="p1" type="out" rate="1"/>
    <port name="p2" type="in" rate="1"/>
    <port name="p3" type="out" rate="1"/>
<port name="p4" type="in" rate="1"/>
    <port name="p5" type="out" rate="1"/>
    <port name="p6" type="in" rate="1"/>
    <port name="p7" type="out" rate="1"/>
  </actor>
  <actor name="tNIvld2" type="NI">
    <port name="p0" type="in" rate="32"/>
    <port name="p1" type="out" rate="1"/>
    <port name="p2" type="in" rate="1"/>
    <port name="p3" type="out" rate="1"/>
<port name="p4" type="in" rate="1"/>
    <port name="p5" type="out" rate="32"/>
  </actor>
  <actor name="Svld2" type="S">
    <port name="p0" type="in" rate="1"/>
    <port name="p1" type="out" rate="1"/>
    <port name="p2" type="in" rate="1"/>
    <port name="p3" type="out" rate="1"/>
<port name="p4" type="in" rate="1"/>
    <port name="p5" type="out" rate="1"/>
  </actor>
  <actor name="iNIvld2" type="NI">
    <port name="p0" type="in" rate="1"/>
    <port name="p1" type="out" rate="1"/>
    <port name="p2" type="in" rate="1"/>
    <port name="p3" type="out" rate="1"/>
<port name="p4" type="in" rate="32"/>
    <port name="p5" type="out" rate="32"/>
    <port name="p6" type="in" rate="1"/>
    <port name="p7" type="out" rate="1"/>
  </actor>


  <actor name="tNIvld3" type="NI">
    <port name="p0" type="in" rate="1"/>
    <port name="p1" type="out" rate="32"/>
    <port name="p2" type="in" rate="1"/>
    <port name="p3" type="out" rate="1"/>
<port name="p4" type="in" rate="32"/>
  </actor>
  <actor name="Svld3" type="S">
    <port name="p0" type="in" rate="1"/>
    <port name="p1" type="out" rate="1"/>
    <port name="p2" type="in" rate="1"/>
    <port name="p3" type="out" rate="1"/>
<port name="p4" type="in" rate="1"/>
    <port name="p5" type="out" rate="1"/>
  </actor>
  <actor name="iNIvld3" type="NI">
    <port name="p0" type="in" rate="32"/>
    <port name="p1" type="out" rate="1"/>
    <port name="p2" type="in" rate="1"/>
    <port name="p3" type="out" rate="1"/>
<port name="p4" type="in" rate="1"/>
    <port name="p5" type="out" rate="32"/>
    <port name="p6" type="in" rate="1"/>
    <port name="p7" type="out" rate="1"/>
  </actor>
  <actor name="tNIvld4" type="NI">
    <port name="p0" type="in" rate="1"/>
```

```
    <port name="p1" type="out" rate="1"/>
    <port name="p2" type="in" rate="1"/>
    <port name="p3" type="out" rate="1"/>
<port name="p4" type="in" rate="1"/>
    <port name="p5" type="out" rate="1"/>
  </actor>
  <actor name="Svld4" type="S">
    <port name="p0" type="in" rate="1"/>
    <port name="p1" type="out" rate="1"/>
    <port name="p2" type="in" rate="1"/>
    <port name="p3" type="out" rate="1"/>
<port name="p4" type="in" rate="1"/>
    <port name="p5" type="out" rate="1"/>
  </actor>
  <actor name="iNIvld4" type="NI">
    <port name="p0" type="in" rate="1"/>
    <port name="p1" type="out" rate="1"/>
    <port name="p2" type="in" rate="1"/>
    <port name="p3" type="out" rate="1"/>
<port name="p4" type="in" rate="1"/>
    <port name="p5" type="out" rate="1"/>
<port name="p6" type="in" rate="1"/>
    <port name="p7" type="out" rate="1"/>
  </actor>


  <!-- IQ Function -->
  <actor name="tNIiq1" type="NI">
    <port name="p0" type="in" rate="1"/>
    <port name="p1" type="out" rate="1"/>
    <port name="p2" type="in" rate="1"/>
    <port name="p3" type="out" rate="1"/>
<port name="p4" type="in" rate="1"/>
  </actor>
  <actor name="Siq1" type="S">
    <port name="p0" type="in" rate="1"/>
    <port name="p1" type="out" rate="1"/>
    <port name="p2" type="in" rate="1"/>
    <port name="p3" type="out" rate="1"/>
<port name="p4" type="in" rate="1"/>
    <port name="p5" type="out" rate="1"/>
  </actor>
  <actor name="iNIiq1" type="NI">
    <port name="p0" type="in" rate="1"/>
    <port name="p1" type="out" rate="1"/>
    <port name="p2" type="in" rate="1"/>
    <port name="p3" type="out" rate="1"/>
<port name="p4" type="in" rate="1"/>
    <port name="p5" type="out" rate="1"/>
    <port name="p6" type="in" rate="1"/>
    <port name="p7" type="out" rate="1"/>
  </actor>
  <actor name="tNIiq2" type="NI">
    <port name="p0" type="in" rate="16"/>
    <port name="p1" type="out" rate="1"/>
    <port name="p2" type="in" rate="1"/>
    <port name="p3" type="out" rate="1"/>
<port name="p4" type="in" rate="1"/>
    <port name="p5" type="out" rate="16"/>
  </actor>
  <actor name="Siq2" type="S">
    <port name="p0" type="in" rate="1"/>
    <port name="p1" type="out" rate="1"/>
```

```
    <port name="p2" type="in" rate="1"/>
    <port name="p3" type="out" rate="1"/>
<port name="p4" type="in" rate="1"/>
    <port name="p5" type="out" rate="1"/>
  </actor>
  <actor name="iNIiq2" type="NI">
    <port name="p0" type="in" rate="1"/>
    <port name="p1" type="out" rate="1"/>
    <port name="p2" type="in" rate="1"/>
    <port name="p3" type="out" rate="1"/>
<port name="p4" type="in" rate="16"/>
    <port name="p5" type="out" rate="16"/>
    <port name="p6" type="in" rate="1"/>
    <port name="p7" type="out" rate="1"/>
  </actor>


  <!-- MC Function -->
  <actor name="tNImc3" type="NI">
    <port name="p0" type="in" rate="1"/>
    <port name="p1" type="out" rate="32"/>
    <port name="p2" type="in" rate="1"/>
    <port name="p3" type="out" rate="1"/>
<port name="p4" type="in" rate="32"/>
  </actor>
  <actor name="Smc3" type="S">
    <port name="p0" type="in" rate="1"/>
    <port name="p1" type="out" rate="1"/>
    <port name="p2" type="in" rate="1"/>
    <port name="p3" type="out" rate="1"/>
<port name="p4" type="in" rate="1"/>
    <port name="p5" type="out" rate="1"/>
  </actor>
  <actor name="iNImc3" type="NI">
    <port name="p0" type="in" rate="32"/>
    <port name="p1" type="out" rate="1"/>
    <port name="p2" type="in" rate="1"/>
    <port name="p3" type="out" rate="1"/>
<port name="p4" type="in" rate="1"/>
    <port name="p5" type="out" rate="32"/>
    <port name="p6" type="in" rate="1"/>
    <port name="p7" type="out" rate="1"/>
  </actor>
  <actor name="tNImc4" type="NI">
    <port name="p0" type="in" rate="1"/>
    <port name="p1" type="out" rate="1"/>
    <port name="p2" type="in" rate="1"/>
    <port name="p3" type="out" rate="1"/>
<port name="p4" type="in" rate="1"/>
    <port name="p5" type="out" rate="1"/>
  </actor>
  <actor name="Smc4" type="S">
    <port name="p0" type="in" rate="1"/>
    <port name="p1" type="out" rate="1"/>
    <port name="p2" type="in" rate="1"/>
    <port name="p3" type="out" rate="1"/>
<port name="p4" type="in" rate="1"/>
    <port name="p5" type="out" rate="1"/>
  </actor>
  <actor name="iNImc4" type="NI">
    <port name="p0" type="in" rate="1"/>
    <port name="p1" type="out" rate="1"/>
    <port name="p2" type="in" rate="1"/>
```

```
        <port name="p3" type="out" rate="1"/>
    <port name="p4" type="in" rate="1"/>
        <port name="p5" type="out" rate="1"/>
    <port name="p6" type="in" rate="1"/>
        <port name="p7" type="out" rate="1"/>
      </actor>

<!-- Memory Controller Actors -->
      <!-- VLD Function -->
      <actor name="thetavld1" type="M">
        <port name="p0" type="in" rate="1"/>
        <port name="p1" type="out" rate="1"/>
        <port name="p3" type="out" rate="16"/>
      </actor>
      <actor name="rhovldR" type="M">
        <port name="p0" type="in" rate="1"/>
        <port name="p1" type="out" rate="1"/>
        <port name="p2" type="in" rate="1"/>
        <port name="p3" type="out" rate="1"/>
      </actor>
      <actor name="Mvld2" type="M">
        <port name="p0" type="in" rate="16"/>
        <port name="p1" type="out" rate="1"/>
        <port name="p2" type="in" rate="1"/>
      </actor>
      <actor name="thetavld3" type="M">
        <port name="p0" type="in" rate="1"/>
        <port name="p1" type="out" rate="1"/>
        <port name="p3" type="out" rate="1"/>
      </actor>
      <actor name="rhovldW" type="M">
        <port name="p0" type="in" rate="1"/>
        <port name="p1" type="out" rate="1"/>
        <port name="p2" type="in" rate="1"/>
        <port name="p3" type="out" rate="1"/>
      </actor>
      <actor name="Mvld4" type="M">
        <port name="p0" type="in" rate="1"/>
        <port name="p1" type="out" rate="1"/>
        <port name="p2" type="in" rate="1"/>
      </actor>

      <!-- IQ Function -->
      <actor name="thetaiq1" type="M">
        <port name="p0" type="in" rate="1"/>
        <port name="p1" type="out" rate="1"/>
        <port name="p3" type="out" rate="8"/>
      </actor>
      <actor name="rhoiqR" type="M">
        <port name="p0" type="in" rate="1"/>
        <port name="p1" type="out" rate="1"/>
        <port name="p2" type="in" rate="1"/>
        <port name="p3" type="out" rate="1"/>
      </actor>
      <actor name="Miq2" type="M">
        <port name="p0" type="in" rate="8"/>
        <port name="p1" type="out" rate="1"/>
        <port name="p2" type="in" rate="1"/>
      </actor>
      <!-- MC Function -->
      <actor name="thetamc3" type="M">
        <port name="p0" type="in" rate="1"/>
```

```
        <port name="p1" type="out" rate="1"/>
        <port name="p3" type="out" rate="1"/>
      </actor>
      <actor name="rhomcW" type="M">
        <port name="p0" type="in" rate="1"/>
        <port name="p1" type="out" rate="1"/>
        <port name="p2" type="in" rate="1"/>
        <port name="p3" type="out" rate="1"/>
      </actor>
      <actor name="Mmc4" type="M">
        <port name="p0" type="in" rate="1"/>
        <port name="p1" type="out" rate="1"/>
        <port name="p2" type="in" rate="1"/>
      </actor>

<!-- Channels between Application Actors -->
<!-- VLD Function -->
<channel name="vld12vld2" srcActor="vld1" srcPort="p3" dstActor="vld2" dstPort="p0"/>
<channel name="vld22vldexe" srcActor="vld2" srcPort="p1" dstActor="vldexe" dstPort="p0"/>
<channel name="vldexe2vld1" srcActor="vldexe" srcPort="p1" dstActor="vld1" dstPort="p2" initialTokens='297'/>
<channel name="vldexe2vldexe" srcActor="vldexe" srcPort="p3" dstActor="vldexe" dstPort="p2" initialTokens='1'/>
<channel name="vld32vld4" srcActor="vld3" srcPort="p3" dstActor="vld4" dstPort="p0" />
<channel name="vld42vld3" srcActor="vld4" srcPort="p1" dstActor="vld3" dstPort="p2" initialTokens='3'/>
<channel name="vldexe2vld3" srcActor="vldexe" srcPort="p5" dstActor="vld3" dstPort="p4"/>
<channel name="vld42vldexe" srcActor="vld4" srcPort="p5" dstActor="vldexe" dstPort="p4" initialTokens='3'/>
<!-- Sequence edges between VLD and IQ Function -->
<channel name="vld42vld0" srcActor="vld4" srcPort="p7" dstActor="vld0" dstPort="p0"/>
<channel name="vld02iq1" srcActor="vld0" srcPort="p1" dstActor="iq1" dstPort="p4"/>
<!-- IQ Function -->
<channel name="iq12iq2" srcActor="iq1" srcPort="p3" dstActor="iq2" dstPort="p0"/>
<channel name="iq22iqexe" srcActor="iq2" srcPort="p1" dstActor="iqexe" dstPort="p0"/>
<channel name="iqexe2iq1" srcActor="iqexe" srcPort="p1" dstActor="iq1" dstPort="p2" initialTokens='1'/>
<channel name="iqexe2iqexe" srcActor="iqexe" srcPort="p3" dstActor="iqexe" dstPort="p2" initialTokens='1'/>
<!-- Sequence edges between IQ and IDCT Function -->
<channel name="iqexe2idct1" srcActor="iqexe" srcPort="p5" dstActor="idct1" dstPort="p4"/>
<channel name="idct22iqexe" srcActor="idct2" srcPort="p3" dstActor="iqexe" dstPort="p4" initialTokens='1'/>
<!-- IDCT Function -->
<channel name="idct12idct2" srcActor="idct1" srcPort="p3" dstActor="idct2" dstPort="p0"/>
<channel name="idct22idctexe" srcActor="idct2" srcPort="p1" dstActor="idctexe" dstPort="p0"/>
<channel name="idctexe2idct1" srcActor="idctexe" srcPort="p1" dstActor="idct1" dstPort="p2" initialTokens='1'/>
<!-- Sequence edges between IDCT and MC Function -->
<channel name="idctexe2mc1" srcActor="idctexe" srcPort="p5" dstActor="mc1" dstPort="p4"/>
<channel name="mc22idctexe" srcActor="mc2" srcPort="p3" dstActor="idctexe" dstPort="p4" initialTokens='1'/>
<!-- MC Function -->
<channel name="mc12mc2" srcActor="mc1" srcPort="p3" dstActor="mc2" dstPort="p0"/>
<channel name="mc22mcexe" srcActor="mc2" srcPort="p1" dstActor="mcexe" dstPort="p0"/>
<channel name="mcexe2mc1" srcActor="mcexe" srcPort="p1" dstActor="mc1" dstPort="p2" initialTokens='6'/>
<channel name="mcexe2mcexe" srcActor="mcexe" srcPort="p3" dstActor="mcexe" dstPort="p2" initialTokens='1'/>
<channel name="mc32mc4" srcActor="mc3" srcPort="p3" dstActor="mc4" dstPort="p0" />
<channel name="mc42mc3" srcActor="mc4" srcPort="p1" dstActor="mc3" dstPort="p2" initialTokens='3'/>
<channel name="mcexe2mc3" srcActor="mcexe" srcPort="p5" dstActor="mc3" dstPort="p4"/>
<channel name="mc42mcexe" srcActor="mc4" srcPort="p5" dstActor="mcexe" dstPort="p4" initialTokens='3'/>
<!-- Sequence edges between MC and VLD Function -->
<channel name="mc42vld1" srcActor="mc4" srcPort="p7" dstActor="vld1" dstPort="p4" initialTokens='297'/>

<!-- Channels between Application and NW Adapter Actors -->
<!-- VLD Read -->
<channel name="vld12chopvldR" srcActor="vld1" srcPort="p1" dstActor="chopvldR" dstPort="p0"/>
<channel name="chopvldR2chopvldR" srcActor="chopvldR" srcPort="p3" dstActor="chopvldR" dstPort="p2" initialTokens='1'/>
<!-- VLD Write -->
<channel name="vld32chopvldW" srcActor="vld3" srcPort="p1" dstActor="chopvldW" dstPort="p0"/>
<channel name="chopvldW2chopvldW" srcActor="chopvldW" srcPort="p3" dstActor="chopvldW" dstPort="p2" initialTokens='1'/>
```

```
<!-- IQ Read -->
<channel name="iq12chopiqR" srcActor="iq1" srcPort="p1" dstActor="chopiqR" dstPort="p0"/>
<channel name="chopiqR2chopiqR" srcActor="chopiqR" srcPort="p3" dstActor="chopiqR" dstPort="p2" initialTokens='1'/>
<!-- MC Write -->
<channel name="mc32chopmcW" srcActor="mc3" srcPort="p1" dstActor="chopmcW" dstPort="p0"/>
<channel name="chopmcW2chopmcW" srcActor="chopmcW" srcPort="p3" dstActor="chopmcW" dstPort="p2" initialTokens='1'/>

<!-- Channels between Application, NW Adapter and NoC Actors -->
<!-- VLD Read  -->
<channel name="chopvldR2tNIvld1" srcActor="chopvldR" srcPort="p1" dstActor="tNIvld1" dstPort="p0"/>
<channel name="vld22tNIvld2" srcActor="vld2" srcPort="p3" dstActor="tNIvld2" dstPort="p4" initialTokens='1'/>
<!-- VLD Write -->
<channel name="chopvldW2tNIvld3" srcActor="chopvldW" srcPort="p1" dstActor="tNIvld3" dstPort="p0"/>
<channel name="vld42tNIvld4" srcActor="vld4" srcPort="p3" dstActor="tNIvld4" dstPort="p4" initialTokens='1'/>
<!-- IQ Read  -->
<channel name="chopiqR2tNIiq1" srcActor="chopiqR" srcPort="p1" dstActor="tNIiq1" dstPort="p0"/>
<channel name="iq22tNIiq2" srcActor="iq2" srcPort="p3" dstActor="tNIiq2" dstPort="p4" initialTokens='1'/>
<!-- MC Write -->
<channel name="chopmcW2tNImc3" srcActor="chopmcW" srcPort="p1" dstActor="tNImc3" dstPort="p0"/>
<channel name="mc42tNImc4" srcActor="mc4" srcPort="p3" dstActor="tNImc4" dstPort="p4" initialTokens='1'/>

<!-- Channels between the NoC Actors-->
<!-- VLD Read -->
<channel name="tNIvld12Svld1" srcActor="tNIvld1" srcPort="p1" dstActor="Svld1" dstPort="p0"/>
<channel name="tNIvld12tNIvld1" srcActor="tNIvld1" srcPort="p3" dstActor="tNIvld1" dstPort="p2" initialTokens='1'/>
<channel name="Svld12iNIvld1" srcActor="Svld1" srcPort="p1" dstActor="iNIvld1" dstPort="p0"/>
<channel name="Svld12Svld1" srcActor="Svld1" srcPort="p3" dstActor="Svld1" dstPort="p2" initialTokens='1'/>
<channel name="Svld12tNIvld1" srcActor="Svld1" srcPort="p5" dstActor="tNIvld1" dstPort="p4" initialTokens='1'/>
<channel name="iNIvld12iNIvld1" srcActor="iNIvld1" srcPort="p3" dstActor="iNIvld1" dstPort="p2" initialTokens='1'/>
<channel name="iNIvld22iNIvld1" srcActor="iNIvld2" srcPort="p7" dstActor="iNIvld1" dstPort="p6" initialTokens='1'/>
<channel name="iNIvld12iNIvld2" srcActor="iNIvld1" srcPort="p7" dstActor="iNIvld2" dstPort="p6"/>
<channel name="iNIvld12Svld1" srcActor="iNIvld1" srcPort="p5" dstActor="Svld1" dstPort="p4" initialTokens='1'/>
<channel name="iNIvld22iNIvld2" srcActor="iNIvld2" srcPort="p3" dstActor="iNIvld2" dstPort="p2" initialTokens='1'/>
<channel name="iNIvld22Svld2" srcActor="iNIvld2" srcPort="p5" dstActor="Svld2" dstPort="p0"/>
<channel name="Svld22Svld2" srcActor="Svld2" srcPort="p3" dstActor="Svld2" dstPort="p2" initialTokens='1'/>
<channel name="Svld22iNIvld2" srcActor="Svld2" srcPort="p5" dstActor="iNIvld2" dstPort="p4" initialTokens='32'/>
<channel name="Svld22tNIvld2" srcActor="Svld2" srcPort="p1" dstActor="tNIvld2" dstPort="p0"/>
<channel name="tNIvld22tNIvld2" srcActor="tNIvld2" srcPort="p3" dstActor="tNIvld2" dstPort="p2" initialTokens='1'/>
<channel name="tNIvld22Svld2" srcActor="tNIvld2" srcPort="p5" dstActor="Svld2" dstPort="p4" initialTokens='32'/>
<channel name="tNIvld22vld2" srcActor="tNIvld2" srcPort="p1" dstActor="vld2" dstPort="p2"/>
<!-- VLD Write -->
<channel name="tNIvld32Svld3" srcActor="tNIvld3" srcPort="p1" dstActor="Svld3" dstPort="p0"/>
<channel name="tNIvld32tNIvld3" srcActor="tNIvld3" srcPort="p3" dstActor="tNIvld3" dstPort="p2" initialTokens='1'/>
<channel name="Svld32iNIvld3" srcActor="Svld3" srcPort="p1" dstActor="iNIvld3" dstPort="p0"/>
<channel name="Svld32Svld3" srcActor="Svld3" srcPort="p3" dstActor="Svld3" dstPort="p2" initialTokens='1'/>
<channel name="Svld32tNIvld3" srcActor="Svld3" srcPort="p5" dstActor="tNIvld3" dstPort="p4" initialTokens='32'/>
<channel name="iNIvld32iNIvld3" srcActor="iNIvld3" srcPort="p3" dstActor="iNIvld3" dstPort="p2" initialTokens='1'/>
<channel name="iNIvld42iNIvld3" srcActor="iNIvld4" srcPort="p7" dstActor="iNIvld3" dstPort="p6" initialTokens='1'/>
<channel name="iNIvld32iNIvld4" srcActor="iNIvld3" srcPort="p7" dstActor="iNIvld4" dstPort="p6" />
<channel name="iNIvld32Svld3" srcActor="iNIvld3" srcPort="p5" dstActor="Svld3" dstPort="p4" initialTokens='32'/>
<channel name="iNIvld42iNIvld4" srcActor="iNIvld4" srcPort="p3" dstActor="iNIvld4" dstPort="p2" initialTokens='1'/>
<channel name="iNIvld42Svld4" srcActor="iNIvld4" srcPort="p5" dstActor="Svld4" dstPort="p0"/>
<channel name="Svld42Svld4" srcActor="Svld4" srcPort="p3" dstActor="Svld4" dstPort="p2" initialTokens='1'/>
<channel name="Svld42iNIvld4" srcActor="Svld4" srcPort="p5" dstActor="iNIvld4" dstPort="p4" initialTokens='1'/>
<channel name="Svld42tNIvld4" srcActor="Svld4" srcPort="p1" dstActor="tNIvld4" dstPort="p0"/>
<channel name="tNIvld42tNIvld4" srcActor="tNIvld4" srcPort="p3" dstActor="tNIvld4" dstPort="p2" initialTokens='1'/>
<channel name="tNIvld42Svld4" srcActor="tNIvld4" srcPort="p5" dstActor="Svld4" dstPort="p4" initialTokens='1'/>
<channel name="tNIvld42vld4" srcActor="tNIvld4" srcPort="p1" dstActor="vld4" dstPort="p2"/>

<!-- IQ Read -->
<channel name="tNIiq12Siq1" srcActor="tNIiq1" srcPort="p1" dstActor="Siq1" dstPort="p0"/>
<channel name="tNIiq12tNIiq1" srcActor="tNIiq1" srcPort="p3" dstActor="tNIiq1" dstPort="p2" initialTokens='1'/>
```

```
<channel name="Siq12iNIiq1" srcActor="Siq1" srcPort="p1" dstActor="iNIiq1" dstPort="p0"/>
<channel name="Siq12Siq1" srcActor="Siq1" srcPort="p3" dstActor="Siq1" dstPort="p2" initialTokens='1'/>
<channel name="Siq12tNIiq1" srcActor="Siq1" srcPort="p5" dstActor="tNIiq1" dstPort="p4" initialTokens='1'/>
<channel name="iNIiq12iNIiq1" srcActor="iNIiq1" srcPort="p3" dstActor="iNIiq1" dstPort="p2" initialTokens='1'/>
<channel name="iNIiq22iNIiq1" srcActor="iNIiq2" srcPort="p7" dstActor="iNIiq1" dstPort="p6" initialTokens='1'/>
<channel name="iNIiq12iNIiq2" srcActor="iNIiq1" srcPort="p7" dstActor="iNIiq2" dstPort="p6"/>
<channel name="iNIiq12Siq1" srcActor="iNIiq1" srcPort="p5" dstActor="Siq1" dstPort="p4" initialTokens='1'/>
<channel name="iNIiq22iNIiq2" srcActor="iNIiq2" srcPort="p3" dstActor="iNIiq2" dstPort="p2" initialTokens='1'/>
<channel name="iNIiq22Siq2" srcActor="iNIiq2" srcPort="p5" dstActor="Siq2" dstPort="p0"/>
<channel name="Siq22Siq2" srcActor="Siq2" srcPort="p3" dstActor="Siq2" dstPort="p2" initialTokens='1'/>
<channel name="Siq22iNIiq2" srcActor="Siq2" srcPort="p5" dstActor="iNIiq2" dstPort="p4" initialTokens='16'/>
<channel name="Siq22tNIiq2" srcActor="Siq2" srcPort="p1" dstActor="tNIiq2" dstPort="p0"/>
<channel name="tNIiq22tNIiq2" srcActor="tNIiq2" srcPort="p3" dstActor="tNIiq2" dstPort="p2" initialTokens='1'/>
<channel name="tNIiq22Siq2" srcActor="tNIiq2" srcPort="p5" dstActor="Siq2" dstPort="p4" initialTokens='16'/>
<channel name="tNIiq22iq2" srcActor="tNIiq2" srcPort="p1" dstActor="iq2" dstPort="p2"/>
<!-- MC Write -->
<channel name="tNImc32Smc3" srcActor="tNImc3" srcPort="p1" dstActor="Smc3" dstPort="p0"/>
<channel name="tNImc32tNImc3" srcActor="tNImc3" srcPort="p3" dstActor="tNImc3" dstPort="p2" initialTokens='1'/>
<channel name="Smc32iNImc3" srcActor="Smc3" srcPort="p1" dstActor="iNImc3" dstPort="p0"/>
<channel name="Smc32Smc3" srcActor="Smc3" srcPort="p3" dstActor="Smc3" dstPort="p2" initialTokens='1'/>
<channel name="Smc32tNImc3" srcActor="Smc3" srcPort="p5" dstActor="tNImc3" dstPort="p4" initialTokens='32'/>
<channel name="iNImc32iNImc3" srcActor="iNImc3" srcPort="p3" dstActor="iNImc3" dstPort="p2" initialTokens='1'/>
<channel name="iNImc42iNImc3" srcActor="iNImc4" srcPort="p7" dstActor="iNImc3" dstPort="p6" initialTokens='1'/>
<channel name="iNImc32iNImc4" srcActor="iNImc3" srcPort="p7" dstActor="iNImc4" dstPort="p6" />
<channel name="iNImc32Smc3" srcActor="iNImc3" srcPort="p5" dstActor="Smc3" dstPort="p4" initialTokens='32'/>
<channel name="iNImc42iNImc4" srcActor="iNImc4" srcPort="p3" dstActor="iNImc4" dstPort="p2" initialTokens='1'/>
<channel name="iNImc42Smc4" srcActor="iNImc4" srcPort="p5" dstActor="Smc4" dstPort="p0"/>
<channel name="Smc42Smc4" srcActor="Smc4" srcPort="p3" dstActor="Smc4" dstPort="p2" initialTokens='1'/>
<channel name="Smc42iNImc4" srcActor="Smc4" srcPort="p5" dstActor="iNImc4" dstPort="p4" initialTokens='1'/>
<channel name="Smc42tNImc4" srcActor="Smc4" srcPort="p1" dstActor="tNImc4" dstPort="p0"/>
<channel name="tNImc42tNImc4" srcActor="tNImc4" srcPort="p3" dstActor="tNImc4" dstPort="p2" initialTokens='1'/>
<channel name="tNImc42Smc4" srcActor="tNImc4" srcPort="p5" dstActor="Smc4" dstPort="p4" initialTokens='1'/>
<channel name="tNImc42mc4" srcActor="tNImc4" srcPort="p1" dstActor="mc4" dstPort="p2"/>

<!-- Channels between NoC and Memory Controller -->
<!-- VLD Read Function -->
<channel name="iNIvld12thetavld1" srcActor="iNIvld1" srcPort="p1" dstActor="thetavld1" dstPort="p0"/>
<channel name="thetavld12iNIvld1" srcActor="thetavld1" srcPort="p1" dstActor="iNIvld1" dstPort="p4" initialTokens='2'/>
<channel name="iNIvld22Mvld2" srcActor="iNIvld2" srcPort="p1" dstActor="Mvld2" dstPort="p2" initialTokens='1'/>
<channel name="Mvld22iNIvld2" srcActor="Mvld2" srcPort="p1" dstActor="iNIvld2" dstPort="p0"/>
<!-- VLD Write Function -->
<channel name="iNIvld32thetavld3" srcActor="iNIvld3" srcPort="p1" dstActor="thetavld3" dstPort="p0"/>
<channel name="thetavld32iNIvld3" srcActor="thetavld3" srcPort="p1" dstActor="iNIvld3" dstPort="p4" initialTokens='2'/>
<channel name="iNIvld42Mvld4" srcActor="iNIvld4" srcPort="p1" dstActor="Mvld4" dstPort="p2" initialTokens='1'/>
<channel name="Mvld42iNIvld4" srcActor="Mvld4" srcPort="p1" dstActor="iNIvld4" dstPort="p0"/>
<!-- IQ Read Function -->
<channel name="iNIiq12thetaiq1" srcActor="iNIiq1" srcPort="p1" dstActor="thetaiq1" dstPort="p0"/>
<channel name="thetaiq12iNIiq1" srcActor="thetaiq1" srcPort="p1" dstActor="iNIiq1" dstPort="p4" initialTokens='2'/>
<channel name="iNIiq22Miq2" srcActor="iNIiq2" srcPort="p1" dstActor="Miq2" dstPort="p2" initialTokens='1'/>
<channel name="Miq22iNIiq2" srcActor="Miq2" srcPort="p1" dstActor="iNIiq2" dstPort="p0"/>
<!-- MC Write Function -->
<channel name="iNImc32thetamc3" srcActor="iNImc3" srcPort="p1" dstActor="thetamc3" dstPort="p0"/>
<channel name="thetamc32iNImc3" srcActor="thetamc3" srcPort="p1" dstActor="iNImc3" dstPort="p4" initialTokens='2'/>
<channel name="iNImc42Mmc4" srcActor="iNImc4" srcPort="p1" dstActor="Mmc4" dstPort="p2" initialTokens='1'/>
<channel name="Mmc42iNImc4" srcActor="Mmc4" srcPort="p1" dstActor="iNImc4" dstPort="p0"/>

<!-- Channels of the Memory Controller -->
<!-- VLD Read Function -->
<channel name="thetavld12rhovldR" srcActor="thetavld1" srcPort="p3" dstActor="rhovldR" dstPort="p0"/>
<channel name="rhovldR2rhovldR" srcActor="rhovldR" srcPort="p3" dstActor="rhovldR" dstPort="p2" initialTokens='1'/>
<channel name="rhovldR2Mvld2" srcActor="rhovldR" srcPort="p1" dstActor="Mvld2" dstPort="p0"/>
<!-- VLD Write Function -->
```

```
<channel name="thetavld32rhovldW" srcActor="thetavld3" srcPort="p3" dstActor="rhovldW" dstPort="p0"/>
<channel name="rhovldW2rhovldW" srcActor="rhovldW" srcPort="p3" dstActor="rhovldW" dstPort="p2" initialTokens='1'/>
<channel name="rhovldW2Mvld4" srcActor="rhovldW" srcPort="p1" dstActor="Mvld4" dstPort="p0"/>
<!-- IQ Read Function -->
<channel name="thetaiq12rhoiqR" srcActor="thetaiq1" srcPort="p3" dstActor="rhoiqR" dstPort="p0"/>
<channel name="rhoiqR2rhoiqR" srcActor="rhoiqR" srcPort="p3" dstActor="rhoiqR" dstPort="p2" initialTokens='1'/>
<channel name="rhoiqR2Miq2" srcActor="rhoiqR" srcPort="p1" dstActor="Miq2" dstPort="p0"/>
<!-- MC Write Function -->
<channel name="thetamc32rhomcW" srcActor="thetamc3" srcPort="p3" dstActor="rhomcW" dstPort="p0"/>
<channel name="rhomcW2rhomcW" srcActor="rhomcW" srcPort="p3" dstActor="rhomcW" dstPort="p2" initialTokens='1'/>
<channel name="rhomcW2Mmc4" srcActor="rhomcW" srcPort="p1" dstActor="Mmc4" dstPort="p0"/>

    </sdf>

   <sdfProperties>
<!-- SDF Properties VLD Function -->
      <actorProperties actor="vld1">
        <processor type="arm" default="true">
          <executionTime time="0"/>
        </processor>
      </actorProperties>
      <actorProperties actor="vld2">
        <processor type="arm" default="true">
          <executionTime time="0"/>
        </processor>
      </actorProperties>
      <actorProperties actor="vldexe">
        <processor type="arm" default="true">
          <executionTime time="260180"/>
        </processor>
      </actorProperties>
      <actorProperties actor="vld3">
        <processor type="arm" default="true">
          <executionTime time="0"/>
        </processor>
      </actorProperties>
      <actorProperties actor="vld4">
        <processor type="arm" default="true">
          <executionTime time="0"/>
        </processor>
      </actorProperties>
      <actorProperties actor="vld0">
        <processor type="arm" default="true">
          <executionTime time="0"/>
        </processor>
      </actorProperties>


<!-- SDF Properties IQ Function -->
      <actorProperties actor="iq1">
        <processor type="arm" default="true">
          <executionTime time="0"/>
        </processor>
      </actorProperties>
      <actorProperties actor="iq2">
        <processor type="arm" default="true">
          <executionTime time="0"/>
        </processor>
      </actorProperties>
      <actorProperties actor="iqexe">
        <processor type="arm" default="true">
          <executionTime time="5590"/>
        </processor>
```

```
      </actorProperties>
<!-- SDF Properties IDCT Function -->
      <actorProperties actor="idct1">
        <processor type="arm" default="true">
          <executionTime time="0"/>
        </processor>
      </actorProperties>
      <actorProperties actor="idct2">
        <processor type="arm" default="true">
          <executionTime time="0"/>
        </processor>
      </actorProperties>
      <actorProperties actor="idctexe">
        <processor type="arm" default="true">
          <executionTime time="4860"/>
        </processor>
      </actorProperties>

<!-- SDF Properties MC Function -->
      <actorProperties actor="mc1">
        <processor type="arm" default="true">
          <executionTime time="0"/>
        </processor>
      </actorProperties>
      <actorProperties actor="mc2">
        <processor type="arm" default="true">
          <executionTime time="0"/>
        </processor>
      </actorProperties>
      <actorProperties actor="mcexe">
        <processor type="arm" default="true">
          <executionTime time="109580"/>
        </processor>
      </actorProperties>
      <actorProperties actor="mc3">
        <processor type="arm" default="true">
          <executionTime time="0"/>
        </processor>
      </actorProperties>
      <actorProperties actor="mc4">
        <processor type="arm" default="true">
          <executionTime time="0"/>
        </processor>
      </actorProperties>

<!-- SDF Properties Adapter -->
      <!-- VLD Function -->
      <actorProperties actor="chopvldR">
        <processor type="arm" default="true">
          <executionTime time="558"/>
        </processor>
      </actorProperties>
      <actorProperties actor="chopvldW">
        <processor type="arm" default="true">
          <executionTime time="6"/>
        </processor>
      </actorProperties>
      <!-- IQ Function -->
      <actorProperties actor="chopiqR">
        <processor type="arm" default="true">
          <executionTime time="558"/>
        </processor>
```

```
      </actorProperties>
      <!-- MC Function -->
      <actorProperties actor="chopmcW">
        <processor type="arm" default="true">
          <executionTime time="6"/>
        </processor>
      </actorProperties>

<!-- SDF Properties NoC -->
      <!-- VLD Function -->
      <actorProperties actor="tNIvld1">
        <processor type="arm" default="true">
          <executionTime time="4"/>
        </processor>
      </actorProperties>
      <actorProperties actor="Svld1">
        <processor type="arm" default="true">
          <executionTime time="4"/>
        </processor>
      </actorProperties>
      <actorProperties actor="iNIvld1">
        <processor type="arm" default="true">
          <executionTime time="4"/>
        </processor>
      </actorProperties>
      <actorProperties actor="tNIvld2">
        <processor type="arm" default="true">
          <executionTime time="4"/>
        </processor>
      </actorProperties>
      <actorProperties actor="Svld2">
        <processor type="arm" default="true">
          <executionTime time="4"/>
        </processor>
      </actorProperties>
      <actorProperties actor="iNIvld2">
        <processor type="arm" default="true">
          <executionTime time="128"/>
        </processor>
      </actorProperties>

      <actorProperties actor="tNIvld3">
        <processor type="arm" default="true">
          <executionTime time="128"/>
        </processor>
      </actorProperties>
      <actorProperties actor="Svld3">
        <processor type="arm" default="true">
          <executionTime time="4"/>
        </processor>
      </actorProperties>
      <actorProperties actor="iNIvld3">
        <processor type="arm" default="true">
          <executionTime time="4"/>
        </processor>
      </actorProperties>
      <actorProperties actor="tNIvld4">
        <processor type="arm" default="true">
          <executionTime time="4"/>
        </processor>
      </actorProperties>
      <actorProperties actor="Svld4">
```

```
  <processor type="arm" default="true">
    <executionTime time="4"/>
  </processor>
</actorProperties>
<actorProperties actor="iNIvld4">
  <processor type="arm" default="true">
    <executionTime time="4"/>
  </processor>
</actorProperties>

<!-- IQ Function -->
<actorProperties actor="tNIiq1">
  <processor type="arm" default="true">
    <executionTime time="64"/>
  </processor>
</actorProperties>
<actorProperties actor="Siq1">
  <processor type="arm" default="true">
    <executionTime time="4"/>
  </processor>
</actorProperties>
<actorProperties actor="iNIiq1">
  <processor type="arm" default="true">
    <executionTime time="4"/>
  </processor>
</actorProperties>
<actorProperties actor="tNIiq2">
  <processor type="arm" default="true">
    <executionTime time="4"/>
  </processor>
</actorProperties>
<actorProperties actor="Siq2">
  <processor type="arm" default="true">
    <executionTime time="4"/>
  </processor>
</actorProperties>
<actorProperties actor="iNIiq2">
  <processor type="arm" default="true">
    <executionTime time="64"/>
  </processor>
</actorProperties>

<!-- MC Function -->
<actorProperties actor="tNImc3">
  <processor type="arm" default="true">
    <executionTime time="128"/>
  </processor>
</actorProperties>
<actorProperties actor="Smc3">
  <processor type="arm" default="true">
    <executionTime time="4"/>
  </processor>
</actorProperties>
<actorProperties actor="iNImc3">
  <processor type="arm" default="true">
    <executionTime time="4"/>
  </processor>
</actorProperties>
<actorProperties actor="tNImc4">
  <processor type="arm" default="true">
    <executionTime time="4"/>
  </processor>
```

```
      </actorProperties>
      <actorProperties actor="Smc4">
        <processor type="arm" default="true">
          <executionTime time="4"/>
        </processor>
      </actorProperties>
      <actorProperties actor="iNImc4">
        <processor type="arm" default="true">
          <executionTime time="4"/>
        </processor>
      </actorProperties>

<!-- SDF Properties Memory Controller -->
      <!-- VLD Function -->
      <actorProperties actor="thetavld1">
        <processor type="arm" default="true">
          <executionTime time="64"/>
        </processor>
      </actorProperties>
      <actorProperties actor="rhovldR">
        <processor type="arm" default="true">
          <executionTime time="1.66"/>
        </processor>
      </actorProperties>
      <actorProperties actor="Mvld2">
        <processor type="arm" default="true">
          <executionTime time="0"/>
        </processor>
      </actorProperties>
      <actorProperties actor="thetavld3">
        <processor type="arm" default="true">
          <executionTime time="64"/>
        </processor>
      </actorProperties>
      <actorProperties actor="rhovldW">
        <processor type="arm" default="true">
          <executionTime time="1.66"/>
        </processor>
      </actorProperties>
      <actorProperties actor="Mvld4">
        <processor type="arm" default="true">
          <executionTime time="0"/>
        </processor>
      </actorProperties>


      <!-- IQ Function -->
      <actorProperties actor="thetaiq1">
        <processor type="arm" default="true">
          <executionTime time="32"/>
        </processor>
      </actorProperties>
      <actorProperties actor="rhoiqR">
        <processor type="arm" default="true">
          <executionTime time="1.66"/>
        </processor>
      </actorProperties>
      <actorProperties actor="Miq2">
        <processor type="arm" default="true">
          <executionTime time="0"/>
        </processor>
      </actorProperties>
```

```
      <!-- MC Function -->
      <actorProperties actor="thetamc3">
        <processor type="arm" default="true">
          <executionTime time="64"/>
        </processor>
      </actorProperties>
      <actorProperties actor="rhomcW">
        <processor type="arm" default="true">
          <executionTime time="1.66"/>
        </processor>
      </actorProperties>
      <actorProperties actor="Mmc4">
        <processor type="arm" default="true">
          <executionTime time="0"/>
        </processor>
      </actorProperties>

    </sdfProperties>
  </applicationGraph>
</sdf3>
```

Bibliography

[1] Round-robin scheduling, http://en.wikipedia.org/wiki/Round-robin.

[2] Duan-Shin Lee, A generalized non-preemptive priority queue, Proceedings of the Fourteenth Annual Joint Conference of the IEEE Computer and Communication Societies, Vol. 1, Page 354, 1995.

[3] Technical report, Video coding for low bit rate communication, ITU-T Recommendation H.263, 1996.

[4] ARM Limited, AMBA Specification Rev 2.0, 1999.

[5] Philips Semiconductors, Device transaction level (DTL) protocol specification version 2.2, 2002.

[6] Philips Semiconductors, Memory Transaction Level (MTL) protocol specification, 2002.

[7] ARM Limited, AMBA AXI Protocol Specification v1.0, 2003, 2004.

[8] JEDEC Solid State Technology Associatio, DDR SDRAM Specification, JESD79D edition, May 2005.

[9] B. Akesson, K. Goossens, and M. Ringhofer. Predator: A Predictable SDRAM Memory Controller, Int'l Conf. on Hardware/Software Codesign and System Synthesis (CODES+ISSS), October 2007.

[10] J. Boonstra. U-NIC specification, NXP Semiconsuctors, 2007.

[11] I. Cidon and K. Goossens. Network and Transport Layers in Networks on Chip, Giovanni De Micheli and Luca Benini, editors, Networks on Chips: Technology and Tools, The Morgan Kaufmann Series in Systems on Silicon, chapter 5, pages 147–202, Morgan Kaufmann, 2006.

[12] M. Coenen, S. Murali, A. Radulescu, K. Goossens, and G. D. Micheli. A buffer-sizing Algorithm for Networks on Chip using TDMA and credit-based end-to-end Flow Control, Int'l Conf. on Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2006.

[13] R. L. Cruz. A calculus for network delay. Network elements in isolation, IEEE Trans. Inform. Theory, vol. 37, pp. 114-131, Jan 1991.

[14] S. Floyd and V. Jacobson. Link-sharing and Resource management Models for Packet Networks, IEEE/ACM Transactions on Networking, Vol. 3, No. 4, August 1995.

[15] A. Ghamarian, M. Geilen, S. Stuijk, T. Basten, A. Moonen, M. Bekooij, B. Theelen, and M. Mousavi. Throughput Analysis of Synchronous Data Flow Graphs, Application of Concurrency to System Design, 6th International Conference, ACSD 2006, Proceedings, pages 25-34. Turku, Finland, 27-30 June 2006, IEEE Computer Society Press, Los Alamitos, CA, USA, 2006.

[16] A. Ghamarian, S. Stuijk, T. Basten, M. Geilen, and B. Theelen. Latency Minimization for Synchronous Data Flow Graphs, Digital System Design, 10th Euromicro Conference, DSD 07 Proceedings, pages 189-196, Lbeck, Germany. IEEE Computer Society Press, Los Alamitos, CA, USA, 2007.

[17] K. Goossens, J. Dielissen, O. P. Gangwal, S. G. Pestana, A. Radulescu, and E. Rijpkema. A design flow for application-specific networks on chip with guaranteed performance to accelerate SOC design and verification, in Proceedings of Design, Automation and Test in Europe Conference and Exposition (DATE '05), pp. 11821187, Munich, Germany, March 2005.

[18] K. Goossens, J. Dielissen, and A. Radulescu. Æthereal Network on Chip: Concepts, Architectures, and Implementations, IEEE Design and Test of Computers, Vol 22(5):414–421, Philips Research Laboratories, 2005.

[19] A. Hansson, K. Goossens, and A. Radulescu. A unified approach to mapping and routing on a network on chip for both best-effort and guaranteed service traffic, VLSI Design - Special issue on Networks-on-Chip,Hindawi Publishing Corporation, 2007.

[20] A. Hansson, K. Goossens, and A. Radulescu. Applying dataflow analysis to dimension buffers for guaranteed performance in networks on chip, Technical Report NXP-R-TN, 2008/00013, NXP Semiconductors, 2008.

[21] N. Holsti and S. Saarinen. Status of the Bound-T WCET tool, 2nd International Workshop on Worst-Case Execution Time Analysis, WCET 02, Proceedings, pages 3641, 2002.

[22] E. S. Shin, V. J. M. III, and G. F. Riley. Round-robin Arbiter Design and Generation, Georgia Institute of Technology, Atlanta, GA, Technical Report GIT-CC-02-38, 2002.

[23] S. Sriram and S. Bhattacharyya. Embedded Multiprocessors: Scheduling and Synchronization, Marcel Dekker, Inc, New York, NY, USA, 2000.

[24] Stewart, David, and M. Barr. Rate Monotonic Scheduling," Embedded Systems Programming, pp. 79-80, March 2002.

[25] D. Stiliadis and A. Varma. Latency-Rate Servers: A General Model for Analysis of Traffic Scheduling Algorithms, IEEE/ACM Transactions on Networking, October 1998.

[26] S. Stuijk. Eindhoven, The Netherland, 2007. Predictable Mapping of Streaming Applications on Multiprocessors, Thesis submitted to Faculty of Electrical Engineering, Eindhoven University of Technology.

[27] S. Stuijk, M. Geilen, and T. Basten. Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs, DAC'06, Proceedings, pages 899-904, ACM, 2006.

[28] S. Stuijk, M. Geilen, and T. Basten. SDF3: SDF For Free, 6th International Conference on Application of Concurrency to System Design, ACSD 06, Proceedings, pages 276-278, IEEE, 2006.

[29] B. Theelen, M. Geilen, T. Basten, J. Voeten, S. Gheorghita, and S. Stuijk. A scenario-aware data flow model for combined long-run average and worst-case performance analysis. In 4th International Conference on Formal Methods and Models for Co-Design, MEMOCODE 06, Proceedings, pages 185194. IEEE, 2006.

[30] M. H. Wiggers, M. J. G. Bekooij, and G. J. M. Smit. Modelling run-time arbitration by latency-rate servers in dataflow graphs, SCOPES '07: Proceedings of the 10th international workshop on Software compilers for embedded systems, pages 11.22, ACM, New York, NY, USA, 2007.

[31] H. Zimmermann. OSI Reference Model, The ISO Model of Architecture for Open Systems Interconnection, IEEE Transactions on Communications, vol. 28, no. 4, 1980.