Concepts and Implementation of the Philips Network-on-Chip

John Dielissen, Andrei Rădulescu, Kees Goossens, and Edwin Rijpkema
Philips Research Laboratories, Eindhoven, The Netherlands
Email: john.dielissen@philips.com

Abstract

SoC communication infrastructures, such as the Æthereal network on chip (NoC), will play a central role in integrating IPs with diverse communication requirements. To achieve a compositional and predictable system design, it is essential to reduce uncertainties in the interconnect, such as throughput and latency. In our NoC, these uncertainties are eliminated by providing guaranteed throughput and latency services. Our NoC consists of routers and network interfaces. The routers provide reliable data transfer. The network interfaces implement, via connections, high-level services, such as transaction ordering, throughput and latency guarantees, and end-to-end flow control. The network interfaces also implement adapters to existing on-chip protocols, such as AXI, OCP and DTL, to seamlessly connect existing IP modules to the NoC. These services are implemented in hardware to achieve high speed, and low area. Our NoC provides run-time reconfiguration. We show that in the Æthereal NoC, this is achieved by using the NoC itself, instead of an additional control network. We present an instance of a 6-port router with an area of $0.175mm^2$ *after layout, and a network interface with 4 IP ports having a synthesized area of* $0.172mm^2$*. Both the router and the network interface are implemented in* $0.13\mu m$ *technology, and run at* 500 *MHz.*

1 Introduction

As systems on chip (SoC) grow in complexity, the traditional on-chip interconnects, such as buses and switches, cannot be used anymore, due to their limited scalability. Networks on chip (NoC) scale better, and, therefore, they are a solution to large SoCs [2–5, 7, 9–11, 14].

NoCs offer well-defined interfaces [2, 8, 14, 16], decoupling computation from communication, and easing design. It has been shown that NoCs can provide interfaces to existing on-chip communication protocols, such as AXI [1], OCP [12], DTL [13], thus, enabling reuse of existing IP modules [8, 15].

A disadvantage of large interconnects in general (e.g., buses with bridges, or NoCs) is that they introduce uncertainties (e.g., due to contention). Applications also introduce uncertainties as they become more dynamic and heterogeneous. All these complicate integration, especially in hard real-time systems (e.g., video), as the user expects the resulting system to be predictable.

In the Æthereal NoC, we advocate the use of differentiated services and the use of guaranteed communication to eliminate uncertainties in the interconnect, and to ease integration [8]. We allow differentiated services by offering communication services on connections that can be configured individually for different services. Examples of properties that can be configured on a connection are throughput and latency that can be configured to have no guarantees (i.e., best effort) or guaranteed for a particular bound. By providing guarantees, our NoC offers predictable communication, which is a first step in designing a predictable system.

In the next section, we present the Æthereal NoC, which offers both guaranteed and best-effort services. The NoC consists of routers and network interfaces. Our routers, described in Section 2.1, use input queuing, wormhole routing, link-level flow control and source routing. It has two traffic classes for the GT and BE data. For GT, time slots are reserved such that no contention occurs, while for BE, we use a round-robin arbitration to solve contention. The network interfaces, described in Section 2.2, have a modular design, composed of kernel and shells. The NI kernel provides the basic functionality, including arbitration between connections, ordering, end-to-end flow control, packetization, and a link protocol with the router. Shells implement (a) additional functionality, such as multicast and narrowcast connections, and (b) adaptors to existing protocols, such as AXI or DTL. All these shells can be plugged in or left out at instantiation time according to the needs to optimize area cost.

The network connections are configurable at runtime via a memory-mapped configuration port. In Section 3, we show how the network is used to configured itself as opposed to using a separate control interconnect for network configuration.

2 Concepts of the Network

The network on chip as is exemplified by Figure 1, consists of two components: the routers and the network interfaces (NI). The routers can be randomly connected amongst themselves and to the network interfaces (i.e., there are no topology constraints). Note that in principle there can be multiple links between routers. The routers transport packets of data from one NI to another. The NIs are responsible for packetization/depacketization, for implementing the connections and services, and for offering a standard interface (e.g., AXI or OCP) to the IP modules connected to the NoC.

The Æthereal NoC provides both best-effort and guaranteed services (e.g., latency or throughput). To implement guarantees, we use *contention-free routing*, which is based on a time-division-multiplexed circuit-switching approach, where one or more circuits are set up for a connection [14]. This requires a logical notion of synchronicity, where all routers and NIs are in the same slot. Circuits are created by reserving consecutive slots in consecutive routers/NIs. This is, the circuits are pipelined, in the sense that if a circuit is set from router *R* to router *R'*, and slot s is reserved at router *R*, then slot $s + 1$ must be reserved at router *R'*. On these circuits, data received in one slot will be forwarded to the next router/NI in the next slot. By setting up circuits, we ensure that data is transported without contention. In this way throughput

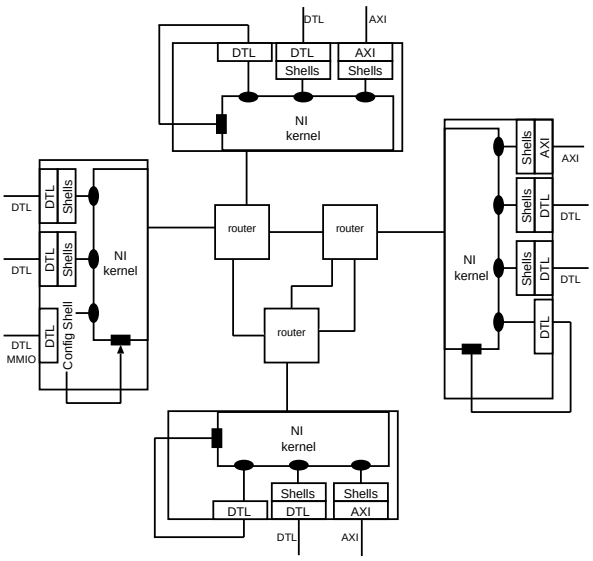

Figure 1. network example

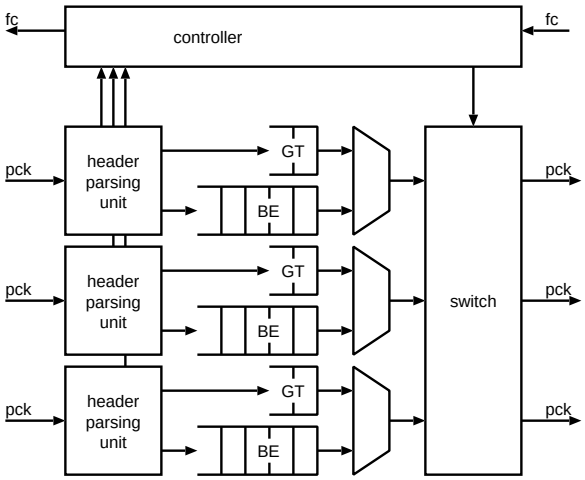

Figure 2. Router architecture

and latency are guaranteed. We call this guaranteed traffic as guaranteed throughput (GT) data, as opposed to the best-effort (BE) data, for which no throughput guarantees are given.

As mentioned above, circuits are set up by reserving slots. These slots are reserved such that no more than one GT data is scheduled at the same time on an output port of a router or NI. BE data is transferred on the slots that are not used by the GT data: either the slots are unreserved, or the slots are reserved, but not used. BE data can be delayed because of the higher priority GT data, or because of contention on the ports.

In the following sections, we describe in detail the router and NI architectures.

2.1 Router Architecture

Routers send data from one network interface to the other by means of packets. Such a packet consists of one or more flits, were a flit is the minimal transmission unit. As a transmission scheme we use wormhole routing, because of the low cost (the buffer capacity can be less than the length of a packet) and low latency (the router can start forwarding the first flit of a packet without waiting for the tail). To reduce queuing capacity of a router, and thus the area, input queuing is used, as shown in Figure 2.

We select source routing as an addressing scheme, because it allows topology independence, while at the same time has a low cost: no expensive (programmable) lookup tables are needed in the router. In source routing the path on which the packet travels is included in the header of a packet. In Æthereal, this path is a list of destination ports, from which each router on the path removes the first element for its own use.

In the Æthereal network guarantees are given by statically calculating the GT schedule. In this way conflicts at the destination ports at each router can be avoided. In fact a pipelined circuit switching network is set-up. Since the network is distributed, also the circuit switching configuration, being the time at which GT-packets arrive and to which destination port they have to go, has to be distributed In earlier versions of the Æthereal router [14], this was done in ”local” slot tables. When programming a GT-connection, all slot tables on the path are consulted to avoid conflicts in the schedule. In this way distributed programming is enabled, which is essential for large networks. However, for the next years we expect the network to be small and a centralized programming scheme is chosen, for which no ”local” slot table is needed. The area cost of the ”local” slot table is quite high because first of all, the table itself costs area (approximately 25% of the total area of a router with 6 bidirectional ports and 256 slots), and second the programming unit, and the connected additional port on the router have to be provided (an additional 25%). In this paper we present a NoC with centralized configuration. We include the switching configuration (the path) in the packet header, and, as a consequence the slot tables are removed from the routers.

Besides the path, the header also contains information for the Network interface, which is explained in section 2.2.1. As explained, a network packet is build up of one or more network flits. For the current Æthereal network, the flit size is chosen 3 to optimize the data clock frequency and control frequency. The information of the type of flit is annotated in the first element of the sideband information, being the id. The format of the flits is shown in Figure 3. The figure shows that the flit contains a header and 2 payload words. Only the first flit of a packet has a header and as a consequence, the next flits can have 3 payload words. Note that when packets consist of multiple flits, the overhead of the header is reduced. The amount of valid words in the flit is stored in the size field. The end of the packet is notified by the eop flag in the sideband information. As an example, Figure 4 shows how a packet, containing 10 payload words can be build up.

GT and BE-flits are semantically the same, but they are handled differently by the scheduler: GT-flits are always scheduled for the next cycle. The BE-flits are scheduled to the remaining destination ports according a round-robin schedule. Ones a first flit of a BE-packet is send to a certain destination port, than port remains locked until the packet is finished: the port does not schedule BE-flits from the other input ports. In this way the interleaving of BE-packets is avoided, which makes implementation simple and cheap. Note that BE-packets can still be interleaved with GT-packets. For GT-flits the interleaving amongst themselves has to be avoided in the static schedule.

The router has a controller and a data path elements. In the data path, the input messages, from either routers or network interfaces

id	credit	qid	path
size	payload 0		
eop	payload 1		

Figure 3. Network flits

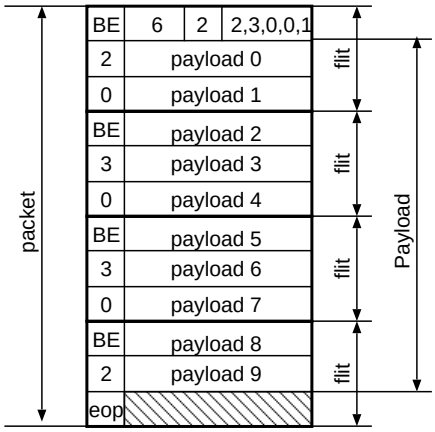

Figure 4. BE-packet example

are parsed by the header-parsing units (hpu). These units, shown in Figure 2, remove the first element for the path, send the parsed flits into GT or BE queues and notify the controller that there is a packet. The controller schedules flits for the next cycle. After scheduling the GT-flits, the remaining destination ports can serve the BE-flits. In the case of conflicts (e.g. two BE-flits address the same destination), a round-robin arbitration scheme is applied. The controller sets the switches in the right direction for the duration of the next flit cycle. Furthermore the read commands will be given to the fifo's.

To avoid overflow in the BE-input queues, a link-level flow control scheme is implemented. Each router is initialized with the amount of free space in the connected routers and network interfaces. Every time a flit is send to a next router, the free space counter corresponding to that destination port is decremented. When a router schedules a flit for the next slot, it signals its predecessor that the free space counter can be incremented. Since GT-packets follow a pipelined circuit, a GT-flit is always send to the next router in the next cycle, and therefore link-level flow control can be omitted.

2.1.1 Implementation

We synthesized and layouted in a $0.13\mu m$ technology a prototype router with 6 bidirectional ports, and BE input queues of 32-bit wide and 24-word deep each (see Figure 5). In the floorplan the area-efficient custom-made hardware fifos, that we use for the BE and GT queues, are clearly visible. The design is fully testable using the well known scan-chain test method, and power stipes are included. The total area of the router sums up to $0.175mm^2$. The router runs at a frequency of 500 MHz, and delivers a bandwidth of 16 Gbit/s per link in each direction.

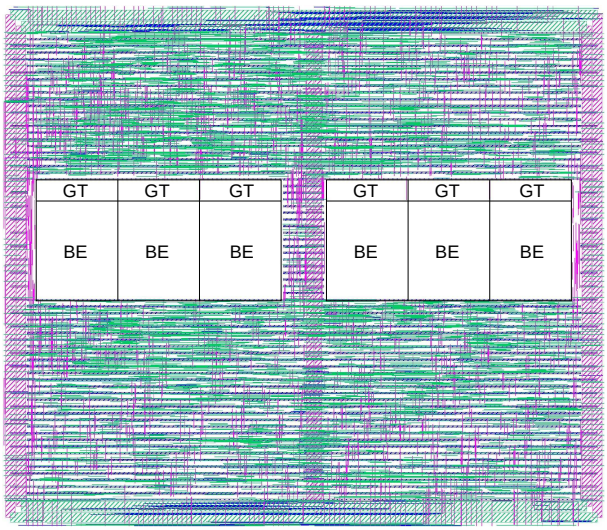

Figure 5. Layout of a router with 6 bidirectional ports

2.2 Network Interface Architecture

The network interface (NI) is the component that provides the conversion of the packet-based communication of the network to the higher-level protocol that IP modules use. We split the design of the network interface in two parts: (a) the *NI kernel*, which packetizes messages and schedules them to the routers, implements the end-to-end flow control, and the clock domain crossing, and (b) the *NI shells*, which implement the connections (e.g., narrowcast, multicast), transaction ordering, and other higher-level issues specific to the protocol offered to the IP. We describe the architectures of the NI kernel and the NI shells in the next two sections, and the results for their implementation in Section 2.2.3.

2.2.1 NI Kernel Architecture

The NI kernel (see Figure 6) receives and provides messages, which contain the data provided by the IP modules via their protocol after sequentialization. The message structure may vary depending on the protocol used by the IP module. However, the message structure is irrelevant for the NI kernel, as it just sees messages as pieces of data that must be transported over the NoC.

The NI kernel communicates with the NI shells via *ports*. At each port, peer-to-peer connections can be configured, their number being selected at NI instantiation time. A port can have multiple connection to allow differentiated traffic classes (e.g., best effort, or guaranteed throughput), in which case there are also `connid` signals to select on which connection a message is supplied or consumed.

For each connection, there are two message queues (one source queue, for messages going to the network, and one destination queue, for messages coming from the network) in the NI kernel. Their size is also selected at the NI instantiation time. Queues provide the clock domain crossing between the network and the IP modules. Each port can, therefore, have a different frequency.

Each channel is configured individually. In a first prototype of the Æthereal network interface, we can configure if a channel is

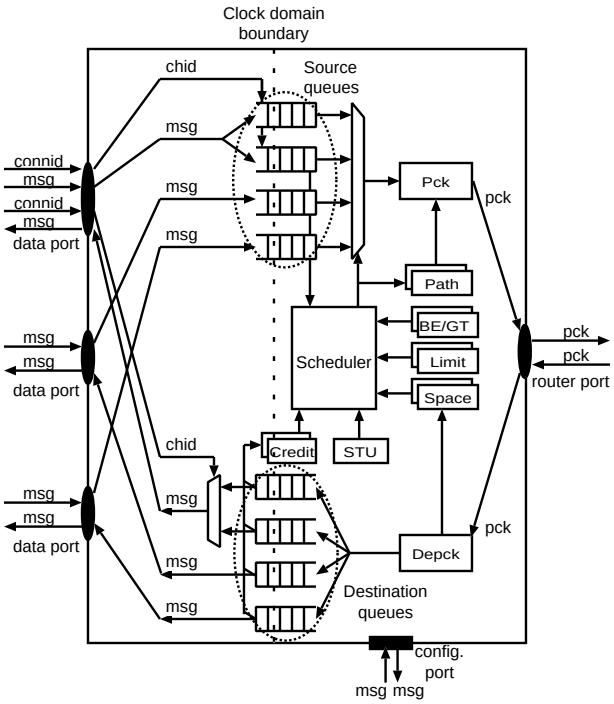

Figure 6. Network interface kernel

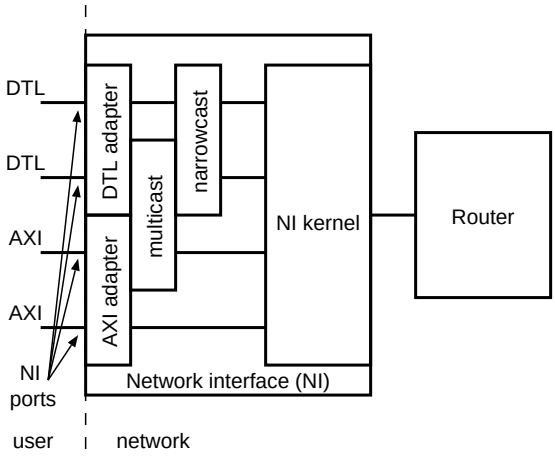

Figure 7. NI kernel and shells

best effort (BE) or providing timing guarantees (GT), reserve slots in the latter case, configure the end-to-end flow control, and the routing information.

End-to-end flow control ensures that no data is sent unless there is enough space in the destination buffer to accommodate it. This is implemented using credits [17]. For each channel, there is a counter (`Space`) tracking the empty buffer space of the remote destination queue. This counter is configured with the remote buffer size. When data is sent from the source queue, the counter is decremented. When data is consumed by the IP module at the other side, credits are produced in a counter (`Credit`) to indicate that more empty space is available. These credits are sent to the producer of data to be added to its `Space` counter. In the Æthereal prototype, we piggyback credits in the header of the packets for the data in the other direction to improve network efficiency. Note that at most `Space` data items can be transmitted. We call *sendable data*, the minimum between the data items in the queue and the value in the counter `Space`.

From the source queues, data is packetized (`Pck`) and sent to the network via a single link. A packet header consists of the routing information (NI address for destination routing, and path for source routing), remote queue id (i.e., the queue of the remote network interface in which the data will be stored), and piggybacked credits (see Figure 3).

There are multiple channels which may require data transmission, we implement a scheduler to arbitrate between them. A queue becomes eligible for scheduling when either there is sendable data (i.e., there is data to be sent, and is space in the channel's destination buffer), or when there are credits to send. In this way, when there is no sendable data, it is still possible to send credits in an empty packet.

The scheduler checks if the current slot is reserved for a GT channel. If the slot is reserved and the GT channel is eligible for scheduling, then the channel is granted data transmission. Otherwise, the scheduler selects an eligible BE channel using some arbitration scheme: e.g. round-robin, weighted round-robin, or based on the queue filling.

Once a queue is selected, a packet containing the largest possible amount of credits and data will be produced. The amount of credits is bound by implementation to the given number of bits in the packet header, and packets have a maximum length to avoid links being used exclusively by a packet/channel, leading to congestion.

On the outgoing path, packets are depacketized, credits are added to the counter `Space`, and data is stored in its corresponding queue, which is given by a queue id field in the header.

2.2.2 NI Shells: The interface to the IP

With the NI kernel described in the previous section, peer-to-peer connections (i.e., between on master and one slave) can be supported directly. These type of connections are useful in systems involving chains of modules communicating peer-to-peer with one another (e.g., video pixel processing [6]).

For more complex type of connections, such as narrowcast or multicast, and to provide conversions to other protocols, we add shells around the NI kernel. As an example, in Figure 7, we show a network interface with two DTL and two AXI ports. All ports provide peer-to-peer connections. In addition to this, the two DTL ports provide narrowcast connections, and one DTL and one AXI port provide multicast connections. Note that these shells add specific functionality, and can be plugged in or left out at design time according to the requirements. Network instantiation is simple, as we use an XML description to automatically generate the VHDL code for the NIs as well as for the network topology.

In Figures 8 and 9, we show a master and slave shells that implement a simplified version of a protocol such as AXI. The basic functionality of such a shell is to sequentialize commands and their flags, addresses, and write data in request messages, and to desequentialize messages into read data, and write responses. Examples of the message structures (i.e., after sequentialization) passing from NI shells and NI kernel are shown in Figure 10.

In full-fledged master and slave shells, more blocks would be added to implement e.g., the unbuffered writes at the master side, and read linked, write conditional at the slave side.

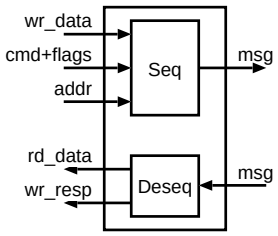

Figure 8. Master shell

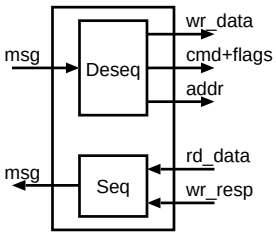

Figure 9. Slave shell

2.2.3 Implementation

We have synthesized an instance of a NI kernel with a slot table of 16 slots, and 4 ports having 1, 1, 2, and 4 channels, respectively, with all queues being 32-bit wide and 8-word deep. The queues are area-efficient custom-made hardware fifos. We use these fifos instead of RAMs, because we need simultaneous access at all NI ports (possibly running at different speeds) as well as simultaneous read and write access for incoming and outgoing packets, which cannot be offered with a single RAM. Moreover, for the small queues needed in the NI, multiple RAMs have a too large area overhead. Furthermore the hardware fifos implement the clock domain boundary allowing each NI port to run at a different frequency. The rest of the NI kernel runs at a frequency of 500 MHz, and delivers a bandwidth towards the router of 16 Gbit/s in each direction. The synthesized area for this NI-kernel instance is $0.13\ mm^2$ in a $0.13\mu m$ technology.

Next to the kernel there are also a number of shells to implement one configuration port, two master ports, and one slave port. These shells add to the are another $0.04\ mm^2$, resulting in a total NI area of $0.172\ mm^2$.

3 Network Configuration

As mentioned in Section 2.1, in our prototype Æthereal network, we opt for centralized programming. This means that there is a single configuration module that configures the whole network, and that slot tables can be removed from the routers.

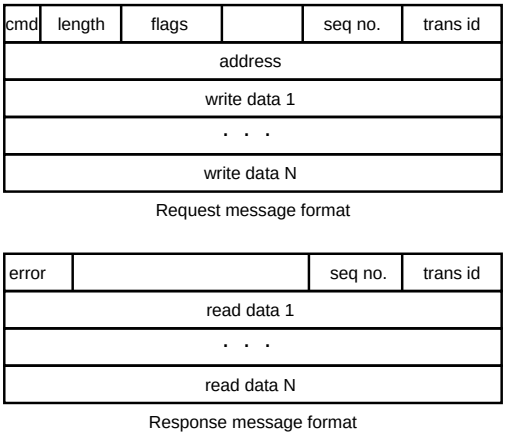

Figure 10. Message format examples

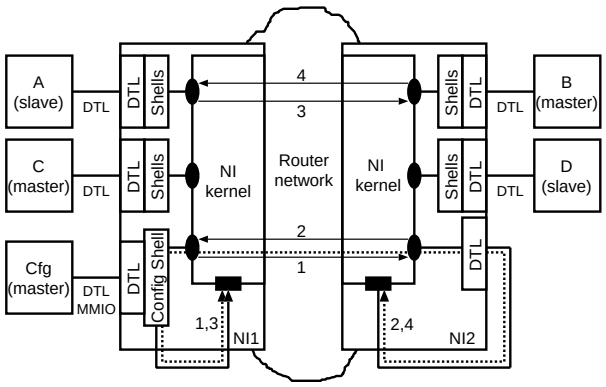

Figure 11. NI configuration

Consequently, only the NIs need to be programmed when opening/closing connections.

NIs are programmed via a configuration port (the DTL MMIO port on which the *Cfg* modules is connected). This port offers a memory-mapped view on all control registers in the NIs. This means that the registers in any NI are readable and writable using normal read and write transactions.

Configuration is performed using the network itself (i.e., there is no separate control interconnect needed for network programming). This is done by directly connecting the NI configuration ports to the network like any other slave (see NI2's configuration port in Figure 11).

At the configuration module *Cfg*'s NI, we introduce a configuration shell (*Config Shell*), which, based on the address configures the local NI (NI1), or sends configuration messages via the network to other NIs. The configuration shell optimizes away the need for an extra data port at NI1 to be connected to the NI1's configuration port.

In Figure 12, we show the necessary steps in setting up a connection between two modules (master B and slave A) from a configuration module (`Cfg`). Like for any other memory-mapped register, before sending configuration messages for programming the B to A connection, a connection to the remote NI must be set up. This involves two channels, one for the requests and and one for the responses between NI1 and the configuration port of NI2. This connection is opened in two steps. First, the channel to the remote NI configuration port is set up by writing the necessary registers in NI1 (Step 1 in Figures 11 and 12). Second, we use this channel to set up (via the network) the channel from configuration port of NI2 to the configuration port of NI1 (Step 2). The three shown messages are delivered and executed in order at NI2. The last of them also requests an acknowledgment message to confirm that the channel has been successfully set up.

After these two configuration channels have been set up, the remote NI2 can be safely programmed. We can, therefore, proceed to setting up a connection from B to A. For programming NI2 (B's NI), the previously set up configuration connection is used. For programming NI1, the NI1's configuration port is accessed directly via `Config Shell`. First, the channel from the slave module A to the master module B is configured by programming NI1 (Step 3). Second, the channel from the master module B to the slave module A is configured (Step 4) through messages to NI2.

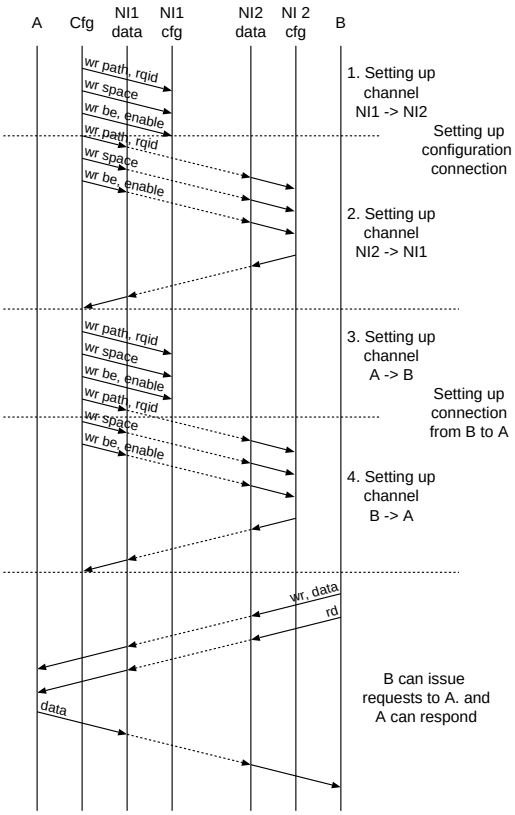

Figure 12. Connection configuration example

4 Conclusions

In this paper, we present the Æthereal network on chip, developed at the Philips Research Laboratories. This network offers, via connections, high-level services, such as transaction ordering, throughput and latency guarantees, and end-to-end flow control. The throughput/latency guarantees are implemented using pipelined time-division-multiplexed circuit-switching.

The network consists of routers and network interfaces. The routers use input queuing, wormhole routing, link-level flow control and source routing. It has two traffic classes for the GT and BE data. For GT, time slots are reserved such that no contention occurs, while for BE, we use a round-robin arbitration to solve contention. We show an instance of a router with 6 bidirectional ports, and BE input queues of 32-bit wide and 24-word deep each implemented using custom-made fifos. This router has an area of $0.175mm^2$ after layout in $0.13\mu m$ technology, and runs at 500 MHz. This has been achieved by omitting the slot tables, and making low area cost decisions at all levels.

The network interfaces have a modular design, composed of kernel and shells. The NI kernel provides the basic functionality, including arbitration between connections, ordering, end-to-end flow control, packetization, and a link protocol with the router. Shells implement (a) additional functionality, such as multicast and narrowcast connections, and (b) adaptors to existing protocols, such as AXI or DTL. All these shells can be plugged in or left out at instantiation time according to the needs to optimize area cost.

We show an instance of our network interface with a slot table of 16 slots, and 4 ports having 1, 1, 2, and 4 channels, respectively. All queues are 32-bit wide and 8-word deep, and are implemented using custom-made fifos. These fifos also implement the clock domain boundary allowing NI ports to run at a different frequency than the network. The NI kernel runs at a frequency of 500 MHz. The synthesized area for the complete network interface is $0.172\ mm^2$ in a $0.13\mu m$ technology,

The network connections are configurable at runtime via a memory-mapped configuration port. We use the network to configured itself as opposed to using a separate control interconnect for network configuration.

In conclusion, we provide efficient network offering high-level services (including guarantees), which allows runtime network programming using the network itself.

References

[1] ARM. *AMBA AXI Protocol Specification*, June 2003.

[2] L. Benini et al. Powering networks on chips. In *ISSS*, 2001.

[3] L. Benini et al. Networks on chips: A new SoC paradigm. *IEEE Computer*, 35(1):70–80, 2002.

[4] E. Bolotin et al. QNoC: QoS architecture and design process for network on chip. *Journal of Systems Architecture*, 49, Dec. 2003. Special issue on Networks on Chip.

[5] W. J. Dally et al. Route packets, not wires: On-chip interconnection networks. In *DAC*, 2001.

[6] O. P. Gangwal et al. Understanding video pixel processing applications for flexible implementations. In *Euromicro DSD*, 2003.

[7] K. Goossens et al. Networks on silicon: Combining best-effort and guaranteed services. In *DATE*, 2002.

[8] K. Goossens et al. Guaranteeing the quality of services in networks on chip. In J. Nurmi, H. Tenhunen, J. Isoaho, and A. Jantsch, editors, *Networks on Chip*, pages 61–82. Kluwer, 2003.

[9] P. Guerrier et al. A generic architecture for on-chip packet-switched interconnections. In *DATE*, 2000.

[10] F. Karim et al. An interconnect architecture for networking systems on chip. *IEEE Micro*, 22(5), 2002.

[11] S. Kumar et al. A network on chip architecture and design methodology. In *ISVLSI*, 2002.

[12] OCP International Partnership. *Open Core Protocol Specification. 2.0 Release Candidate*, 2003.

[13] Philips Semiconductors. *Device Transaction Level (DTL) Protocol Specification. Version 2.2*, July 2002.

[14] E. Rijpkema et al. Trade offs in the design of a router with both guaranteed and best-effort services for networks on chip. In *DATE*, 2003.

[15] A. Rădulescu et al. Communication services for networks on chip. In S. Bhattacharyya, E. Deprettere, and J. Teich, editors, *Domain-Specific Embedded Multiprocessors*. Marcel Dekker, 2003. to appear.

[16] M. Sgroi et al. Addressing the system-on-a-chip interconnect woes through communication-based design. In *DAC*, 2001.

[17] A. S. Tanenbaum. *Computer Networks*. Prentice Hall, 1996.