An Efficient On-Chip Network Interface Offering Guaranteed Services, Shared-Memory Abstraction, and Flexible Network Configuration

Andrei Rădulescu, John Dielissen, Kees Goossens, Edwin Rijpkema, and Paul Wielage
Philips Research Laboratories, Eindhoven, The Netherlands

Abstract

In this paper we present a network interface for an on-chip network. Our network interface decouples computation from communication by offering a shared-memory abstraction, which is independent of the network implementation. We use a transaction-based protocol to achieve backward compatibility with existing bus protocols such as AXI, OCP and DTL. Our network interface has a modular architecture, which allows flexible instantiation. It provides both guaranteed and best-effort services via connections. These are configured via network interface ports using the network itself, instead of a separate control interconnect. An example instance of this network interface with 4 *ports has an area of* $0.143mm^2$ *in a* $0.13\mu m$ *technology, and runs at* 500 *MHz.*

1 Introduction

Networks on chip (NoC) have been proposed as a solution to the interconnect problem for highly complex chips [2, 3, 5, 9, 12, 14, 15, 17, 21, 27]. NoCs help designing chips in several ways: they (a) structure and manage wires in deep submicron technologies [2, 3, 9, 12, 21], (b) allow good wire utilization through sharing [5, 9, 12, 21], (c) scale better than buses [14, 21], (d) can be energy efficient and reliable [2, 5], and (e) decouple computation from communication through well-defined interfaces, enabling IP modules and interconnect to be designed in isolation, and to be integrated more easily [2, 13, 21, 24]

Networks are composed of *routers*, which transport the data from one place to another, and *network interfaces* (NI), which implement the interface to the IP modules. In a previous article [21], we have shown the trade-offs in designing a cost-effective router combining guaranteed with best-effort traffic. In this paper, we focus on the other network component, the network interface.

Network interface design has received considerable attention for parallel computers [8,25], and computer networks [6,7]. These designs are optimized for performance (high throughput, low latency), and often consist of a dedicated processor, and large amount of buffering. As a consequence, their cost is too large to be applicable on chip.

On-chip network interfaces must provide a low-area overhead, because the size of IP modules attached to the NoC is relatively small. Designs of network interfaces with a low area have been proposed [4, 28]. However, they do not provide throughput or latency guarantees, which are essential for a compositional construction of complex SoCs.

Our NI is intended for systems on chip (SoC), hence, it must have a low area. To enable the reuse of existing IP modules, we must provide a smooth transition from buses to NoCs. A shared-memory abstraction via transactions (e.g., read, write) ensures this. Further, we also have to provide a simple and flexible configuration, preferably using the NoC itself to avoid the need for a separate scalable interconnect.

We achieve a low-cost implementation of the NI by implementing the protocol stack in hardware, and by exploiting on-chip characteristics (such as the absence of transmission errors, relatively static configuration, tight synchronization) to implement only the relevant parts of a complete OSI stack. A hardware implementation of the protocol stack provides a much lower latency overhead compared to a software implementation. Further, a hardware implementation allows both hardware and software cores to be reused without change [4].

Our NI provides services at the transport layer in the ISO-OSI reference model [22], because this is the first layer where offered services are independent of the network implementation. This is a key ingredient in achieving the *decoupling between computation and communication* [16, 24], which allows IP modules and interconnect to be designed independently from each other. We provide transport-layer services by defining connections (e.g., point-to-point or multicast) configured for specific properties (e.g., throughput, ordering).

We offer *guaranteed services* as they are essential for a compositional construction (design and programming) of SoC. The reasons are that they limit the possible interactions of IPs with the communication environment [12, 13], separate the IP requirements and their implementation, and make application quality of service independent of the IP and NoC implementations. Examples of such guarantees are lower bounds on throughput, and upper bounds on latency.

Our NoC, called Æthereal, offers a *shared-memory abstraction* to the IP modules. Communication is performed using a transaction-based protocol, where master IP modules issue *request messages* (e.g., read and write commands at an address, possibly carrying data) that are executed by the addressed slave modules, which may respond with a *response message* (i.e., status of the command execution, and possibly data) [23]. We adopt this protocol to provide backward compatibility to existing on-chip communication protocols (e.g., AXI [1], OCP [18], DTL [19]), and also to allow future protocols better suited to NoCs.

We provide a modular NI, which can be configured at design time. This is, the number of ports and their type (i.e., configuration port, master port, or slave port), the number of connections at each port, memory allocated for the queues, the level of services per port, and the interface to the IP modules are all configurable at design (instantiation) time using an XML description [11].

The NI allows flexible NoC configuration at run time. Each connection can be configured individually, requiring configurable NoC components (i.e., router and NI). However, instead of using

a separate control interconnect to program them, the NoC is used to program itself. This is performed through configuration ports using DTL-MMIO (memory-mapped IO) transactions [19]. The NoC can be configured in a distributed fashion (i.e., via multiple configuration ports), or centralized (i.e., via a single port).

The paper is organized as follows. In the next section, the services that we implement, and the interface offered to the IP modules are described. In Section 3, we show that NoCs can be configured both in a distributed and in a centralized way, and we present the trade-offs between the two approaches. In Section 4, we present a modular network interface architecture, which is split into a kernel, providing core functionality, and a number of shells to extend functionality, e.g., wrappers to provide an interface to existing bus protocols, such as AXI or DTL. In this section, we also show how the NI allows NoC configuration using the NoC itself as opposed to via a separate control interconnect. In Section 5, we demonstrate the feasibility of our network interface design through a prototype implementation in a $0.13\mu m$ technology, and we conclude in Section 6.

2 NoC Services

As mentioned in the previous section, the communication services of the Æthereal NoC are defined to meet the following goals: (a) decouple computation (IP modules) from communication (NoC), (b) provide backward compatibility to existing bus protocols, (c) provide support for real-time communication, and (d) have a low-cost implementation.

Decoupling computation from communication is a key ingredient in managing the complexity of designing chips with billions of transistors, because it allows the IP modules and the interconnect to be designed independently [16, 24]. In NoCs, this decoupling is achieved by positioning the network services at the transport level [3, 21] or above in the ISO-OSI reference model [22]. At the transport level, the offered services are *end to end* between communicating IP modules, hiding, thus, the network internals, such as topology, routing scheme, etc.

Backward compatibility with existing protocols, such as AXI or DTL, is achieved by using a model based on *transactions* [23]. In a transaction-based model, there are two types of IP modules: masters and slaves. Masters initiate transactions by issuing requests, which can be further split in commands, and write data (corresponding to the address and write signal groups in AXI). Examples of commands are read, and write. One or more slaves receive and execute each transaction. Optionally, a transaction can also include a response issued by the slave to the master to return data or an acknowledgment of the transaction execution (corresponding to the read data and write response groups in AXI).

In the Æthereal NoC, all these signals are sequentialized in request and response *messages*, which are supplied to the NoC, where they are transported by means of *packets*. Sequentialization is performed to reduce the number of wires, increasing their utilization, and to simplify arbitration. Packetization is performed by the NI, and is thus transparent to the IP modules.

The Æthereal NoC offers its services on *connections*, which can be point to point (one master, one slave), multicast (one master, multiple slaves, all slaves executing each transaction), and narrowcast (one master, multiple slaves, a transaction is executed by only one slave) [23]. Connections are composed of unidirectional point-to-point *channels* (between a single master and a single slave). To each channel, *properties* are attached, such as guaranteed message delivery or not, in order or un-ordered message delivery, and with or without timing guarantees. As a result, different properties can be attached to the request and response parts of a connection, or for different slaves within the same connection. Connections can be opened and closed at any time. Opening and closing of connections takes time, and is intended to be performed at a granularity larger than individual transactions.

Support for real-time communication is achieved by providing throughput, latency and jitter *guarantees*. In Æthereal, this is implemented by configuring connections as pipelined time-division-multiplexed circuits over the network. Time multiplexing is only possible when the network routers have a notion of synchronicity which allows slots to be reserved consecutively in a sequence of routers [13, 21]. This scheme [21] has smaller packet buffers, and, hence, has lower implementation cost compared to alternatives, such as rate-based packet switching [29], or deadline-based packet switching [20].

Throughput guarantees are given by the number of slots reserved for a connection. Slots correspond to a given bandwidth: B_i, and, therefore, reserving N slots for a connection results in a total bandwidth of $N \times B_i$. The latency bound is given by the waiting time until the reserved slot arrives and the number of routers data passes to reach its destination. Jitter is given by the maximum distance between two slot reservations.

Protocol stacks that are used in networks to implement communication services, require additional cost compared to buses. Protocol stacks are necessary in networks to manage the complexity of networks, and to offer differentiated services. The pressure to keep the protocol stack small is higher on-chip than off-chip, because the size of the IP modules attached to the NoC is relatively small. However, for NoCs, the protocol stacks can be reduced by exploiting the on-chip characteristics (e.g., no transfer errors, short wires) [23]. In the Æthereal NoC, we optimize the performance and minimize the cost of the protocol stack by implementing it in hardware, rather than in software. We support this claim in Section 5.

3 Network Configuration

Before the Æthereal NoC can be used by an application, it must be configured. NoC (re)configuration means opening and closing connections in the system. Connections are set up depending on the application or the mode the system is running. Therefore, we must be able to open and close connections while the system is running. (Re)configuration can be partial or total (some or all connections are opened/closed, respectively).

Opening a connection involves setting several registers, and allocating shared resources (for more details see Section 4). In the case of the current prototype of the Æthereal NoC, for each pair of one master and one slave of a connection, there are 5 and 3 registers written at the master and slave network interfaces, respectively. The shared resources consist of the slots allocated to the connections. These slots can be configured using either a distributed or a centralized model.

In the distributed case, a connection can be opened/closed from multiple network interface ports. Multiple configuration operations can be performed simultaneously, however, potential conflicts must also be solved (e.g., connection configurations initiated at two configuration ports may try to reserve the same slot in a router). Information about the slots is maintained in the routers, which also accept or reject a tentative slot allocation.

In a centralized system, there is only one place that performs NoC configuration. In such a case, the slot information can be

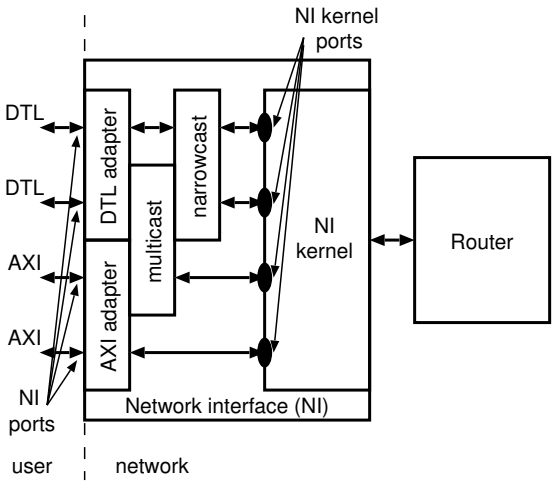

Figure 1. NI kernel and shells

stored in the configuration module instead of the routers, which simplifies the design, and, in the case of small NoCs, may even speed up configuration. For large NoCs, however, centralized configuration can introduce a bottleneck.

In the initial prototype of the Æthereal NoC, we opt for centralized configuration, because it is able to satisfy the needs of a small NoC (around 10 routers), and has a simpler design and lower cost. We use transactions to program the NoC, both for connection registers in the NIs, and for the slot information. We present details of how NoC configuration is performed in Section 4.

4 Network Interface Architecture

The network interface (NI) is the component that provides the conversion of the packet-based communication of the NoC to the higher-level protocol that IP modules use. We split the design of the network interface in two parts (see Figure 1): (a) the *NI kernel*, which implements the channels, 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 for connections, and other higher-level issues specific to the protocol offered to the IP.

4.1 NI Kernel Architecture

The NI kernel (see Figure 2) 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 to be transported over the NoC.

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

Int he NI kernel, there are two message queues for each point-to-point connection (one source queue, for messages going to the NoC, and one destination queue, for messages coming from the NoC). Their size is also selected at the NI instantiation time. In our NI, queues are implemented using custom-made hardware fifos,

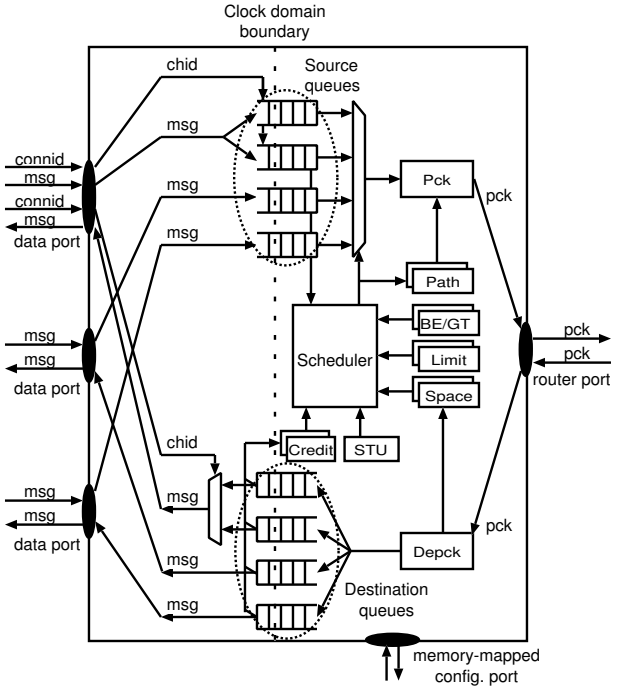

Figure 2. Network interface kernel

and are also used to provide the clock domain crossing between the network and the IP modules. Each port can, therefore, have a different clock frequency.

Each channel is configured individually. In a first prototype of the Æthereal NI, we can configure if a channel provides time guarantees (GT) or not (we call this best effort, BE), reserve slots for GT connections, 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 [26]. For each channel, there is a counter (`Space`) tracking the empty buffer space of the remote destination queue. This counter is initialized 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 NoC efficiency. Note that at most `Space` data items can be transmitted before credits are received. We call the minimum between the data items in the queue and the value in the counter `Space`, the *sendable data*.

From the source queues, data is packetized (`Pck`) and sent to the NoC 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 NI in which the data will be stored), and piggybacked credits.

There are multiple channels which may require data transmission, we implement a scheduler to arbitrate between them. The scheduler checks if the current slot is reserved for a GT channel. If the slot is reserved, is the GT channel has data which can be transmitted, and if there is space in the channel's destination buffer, then the channel is granted data transmission. Otherwise,

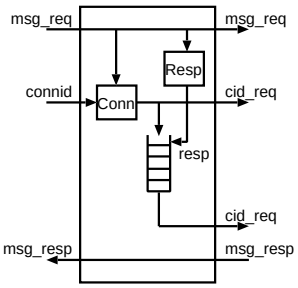

Figure 3. Narrowcast shell

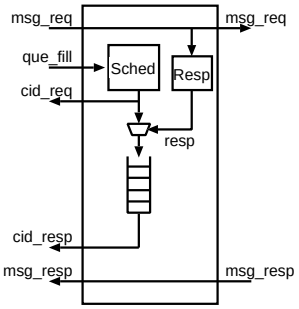

Figure 4. Multi-connection shell

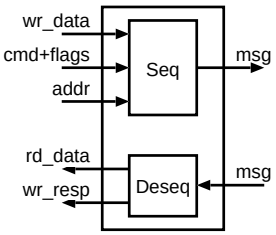

Figure 5. Master shell

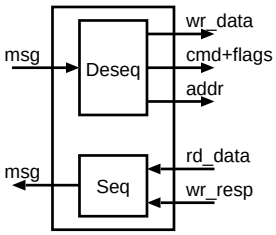

Figure 6. Slave shell

the scheduler selects a BE channel with data and remote space using some arbitration scheme: e.g. round-robin, weighted round-robin, or based on the queue filling.

To optimize the NoC utilization, it is preferable to send longer packets. To achieve this, we implemented a configurable threshold mechanism, which skips a channel as long as the sendable data is below the threshold. This is applicable for both BE and GT channels. To prevent starvation at user/application level (e.g., due to write data being buffered indefinitely on which the IP module waits for an acknowledge), we also provide a *flush* signal for each channel (and a bit in the message header) to temporarily override the threshold. When the flush signal is high for a cycle, a snapshot of its source queue filling is taken, and as long as all the words in the queue at the time of flushing have not been sent, the threshold for that queue is bypassed.

A similar threshold is set for credit transmission. The reason is that, when there is no data on which the credits can be piggybacked, the credits are sent as empty packets, thus, consuming extra bandwidth. To minimize the bandwidth consumed by credits, a credit threshold is set, which allows credits to be transmitted only when their sum is above the threshold. Similarly to the data case, to prevent possible starvation, we provide a flush signal to force credits to be sent even when they are below their threshold.

As credits are piggybacked on packets, a queue becomes eligible for scheduling when either the amount of sendable data are above a first threshold, or when the amount of credits is above a second threshold. However, once a queue is selected, a packet containing the largest possible amount of credits and data will be produced. Note the amount of credits is bound by implementation to the given number of bits in the packet header, and packet have a maximum length to avoid links being used exclusively by a packet/channel, which would cause 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.

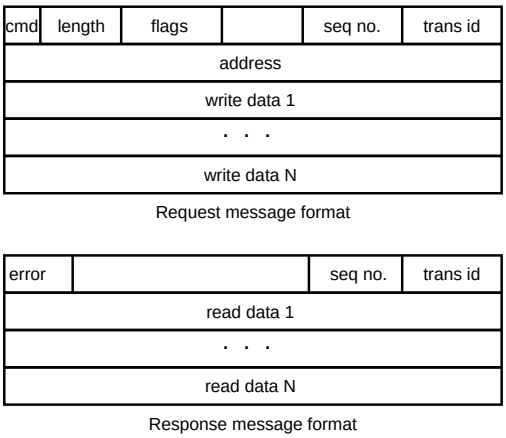

Figure 7. Message format examples

4.2 NI Shells Architectures

With the NI kernel described in the previous section, point-to-point 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 point to point with one another (e.g., video pixel processing [10]).

For more complex types 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 1, we show a NI with two DTL and two AXI ports. All ports provide point-to-point 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. NoC instantiation is simple, as we use an XML description to automatically generate the VHDL code for the NIs as well as for the NoC topology.

In Figure 3, we show an example of a narrowcast shell. Narrowcast connections are connections between one master and several slaves, where each transaction is executed by a single slave selected based on the address provided in the transaction [23]. Narrowcast connections provide a simple, low-cost solution for a single shared address space mapped on multiple memories. It implements the splitting/merging of data going to/coming from these memories.

We implement the narrowcast connection as a collection of point-to-point connections, one for each master-slave pair. Within a narrowcast connection, the slave for which the transaction is destined is selected based on the address (`Conn` block). The address range assigned to a slave is configurable in the narrowcast module. To provide in-order response delivery, the narrowcast must also keep a history of connection identifiers of the transactions including responses (e.g., reads, and acknowledged writes), and the length of these responses. In-order delivery per slave of request messages is already provided by the point-to-point connections.

When a slave using a connectionless protocol (e.g., DTL) is connected to a NI port supporting multiple connections, a multi-connection shell must be included to arbitrate between the connections. A multi-connection shell (see Figure 4) includes a scheduler to select connections from which messages are consumed, based e.g., on their filling. As for the narrowcast, the multi-connection shell has a connection id history for scheduling the responses.

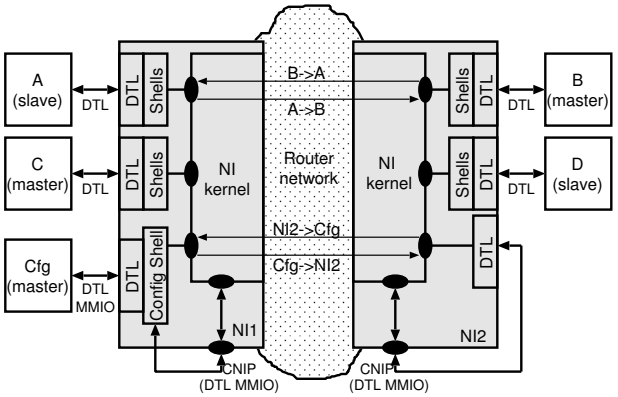

Figure 8. NI configuration

In Figures 5 and 6, 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 7. 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.

4.3 NI Configuration

As mentioned in Section 3, in our prototype Æthereal NoC, we opt for centralized configuration. This means that there is a single configuration module that configures the whole NoC, and that slot tables can be removed from the routers. Consequently, only the NIs need to be configured when opening/closing connections.

NIs are configured via a configuration port (CNIP), which offers a memory-mapped view on all control registers in the NIs. This means that the registers in the NI are readable and writable by any master using normal read and write transactions.

Configuration is performed using the NoC itself (i.e., there is no separate control interconnect needed for NoC configuration). Consequently, the CNIPs are connected to the NoC like any other slave (see CNIP at NI2 in Figure 8). 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 NoC to other NIs. The configuration shell optimizes away the need for an extra data port at NI1 to be connected to the NI1's CNIP.

In Figure 9, 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 configuring the B to A connection at NI2, a connection from Cfg to NI2's CNIP must be set up. This connection is opened in two steps corresponding to the request and response channels. First, the request channel to the NI2's CNIP is set up by writing the necessary registers in NI1 (Step 1 in Figure 9). Second, we use this channel to set up (via the NoC) the response channel from NI2's CNIP to Cfg (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 the configuration connection has been set up, the remote NI2 can be configured to set up a channel from B to A. For configuring NI2 (B's NI), the previously set up configuration connection is used. For configuring 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 at NI1 (Step 3). Second, the channel from the master module B to the slave module A is configured (Step 4) through messages to NI2.

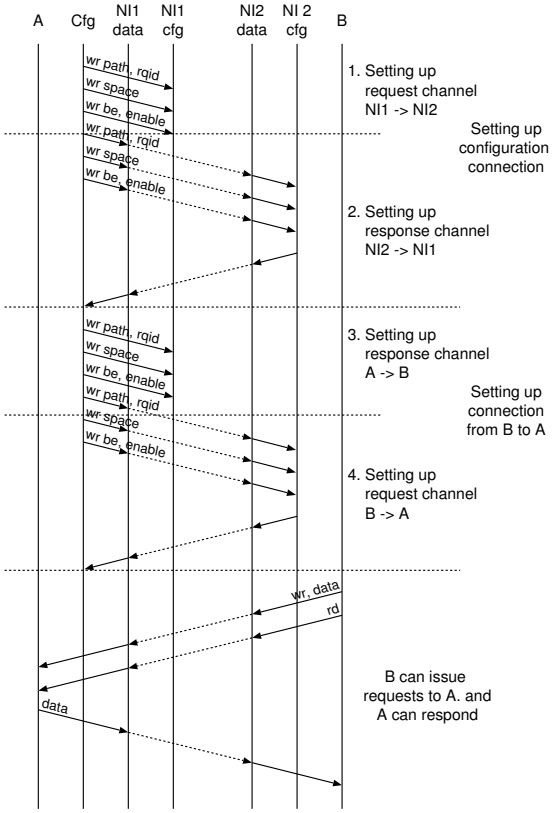

Figure 9. Connection configuration example

5 Implementation

In the previous section, we describe a prototype of a configurable NI architecture. In this section, we discuss the synthesized area and speed figures for the network interface components: NI kernel, narrowcast, multichannel and configuration shells, and master and slave shells for a simplified version of DTL.

We have synthesized an instance of a NI kernel with a STU of 8 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. Finally, for the small queues needed in the NI, multiple RAMs have a too large area overhead. Moreover, the hardware fifos implement the clock domain boundary allowing each NI port to run at a different clock frequency. The router side of the NI kernel runs at a frequency of 500 MHz, which matches our prototype router frequency [21], and delivers a bandwidth toward the router of 16 Gbit/s in each direction. The synthesized area for this NI-kernel instance is $0.11\ mm^2$ in a $0.13\mu m$ technology.

Narrowcast and multi-connection shells have an area of $0.004\ mm^2$ and $0.007\ mm^2$, corresponding to 4% and 6% of the NI kernel area. The DTL shells are very small, $0.005\ mm^2$ and $0.002\ mm^2$ for the master and slave ports, corresponding to 5% and 2% of the NI kernel area, respectively. (This is also due to the fact that not all of the DTL functionality has been implemented). The configuration shell, which provides a simplified DTL-MMIO interface to configure the NoC, has an area of $0.01\ mm^2$.

Summing up, for an example NI with 4 ports, one for configuration (one channel to which the configuration shell is attached), two masters (one offering narrowcast), and one slave (multichannel), the total area is $0.11 + 0.01 + 2 \times 0.005 + 0.004 + 0.002 + 0.007 = 0.143\ mm^2$.

The latency introduced by our current NI is 2 cycles in the DTL master shell (due to sequentialization, as part of packetization), 0 to 2 in the narrowcast and multicast shells (depending on the NI instance), and between 1 and 3 cycles in the NI kernels (as data needs to be aligned to a 3 word flit boundary), and 2 clock cycles for clock domain crossing. Additional delay is caused by the arbitration, but we do not include this in the NI latency overhead, as it needs to be performed anyway (also in the case of a bus, arbitration is performed).

The resulting latency overhead introduced by our NI is between 4 and 10 cycles, which is pipelined to maximize throughput. The latency overhead of a software implementation of the protocol is much larger (e.g., 47 instructions for packetization only [4]). A hardware implementation allows both legacy software and hardware task implementations to be used without change.

6 Conclusions

In this paper, we describe a network interface architecture which offers high-level services at a low cost. Our network interface provides a shared-memory abstraction, where communication is performed using read/write transactions. We offer, via connections, high-level services, such as transaction ordering, throughput and latency guarantees, and end-to-end flow control. These connections are configurable at runtime via a memory-mapped configuration port. We use the network to configure itself as opposed to using a separate control interconnect for network configuration.

Our network interface has a modular design, composed of kernel and shells. The NI kernel provides the basic functionality, including arbitration between channels, transaction 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) adapters to existing protocols, such as AXI or DTL. All these shells can be plugged in or left out at design time according to the needs. This is done using an XML description of the network, which is used to automatically generate the VHDL code for the network interfaces, as well as for the network topology.

We show an instance of our network interface, which shows that the cost of implementing our protocol stack in hardware is small ($0.143mm^2$ in a $0.13\mu m$ technology, running at 500 MHz). Our hardware protocol stack implementation provides a very low protocol overhead of 4 to 6 cycles, which is much lower than a software stack implementation.

In conclusion, we provide an efficient network interface offering a shared-memory abstraction, high-level services (including guarantees), which allows runtime network configuration using the network itself.

References

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

[2] L. Benini and G. De Micheli. Powering networks on chips. In *Proc. ISSS*, 2001.

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

[4] P. Bhojwani and R. Mahapatra. Interfacing cores with on-chip packet-switched networks. In *Proc. VLSI Design*, 2003.

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

[6] T. Callahan and S. C. Goldstein. NIFDY: A low overhead, high throughput network interface. In *Proc. ISCA*, 1995.

[7] A. Chien et al. Design challenges for high-performance network interfaces. *IEEE Computer*, 31(11):42–44, 1998.

[8] D. J. Culler et al. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, 1999.

[9] W. J. Dally and B. Towles. Route packets, not wires: On-chip interconnection networks. In *Proc. DAC*, 2001.

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

[11] S. Gonzalez Pestana et al. Cost-performance trade-offs in networks on chip: A simulation based approach. In *Proc. DATE*, 2004.

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

[13] 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.

[14] P. Guerrier and A. Greiner. A generic architecture for on-chip packet-switched interconnections. In *Proc. DATE*, 2000.

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

[16] K. Keutzer et al. System-level design: Orthogonalization of concerns and platform-based design. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 19(12):1523–1543, 2000.

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

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

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

[20] J. Rexford. *Tailoring Router Architectures to Performance Requirements in Cut-Through Networks*. PhD thesis, Univ. Michigan, 1999.

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

[22] M. T. Rose. *The Open Book: A Practical Perspective on OSI*. Prentice Hall, 1990.

[23] A. Rădulescu and K. Goossens. Communication services for networks on chip. In S. Bhattacharyya, E. Deprettere, and J. Teich, editors, *Domain-Specific Embedded Multiprocessors*. Dekker, 2003.

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

[25] P. Steenkiste. A high-speed network interface for distributed-memory systems: Architecture and applications. *ACM Trans. on Computer Systems*, 15(1):75–109, 1997.

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

[27] D. Wiklund and D. Liu. Socbus: switched network on chip for hard real time embedded systems. In *Proc. IPDPS*, 2003.

[28] C. A. Zeferino, M. E. Kreutz, L. Carro, and A. A. Susin. A study on communication issues for systems-on-chip. In *Proc. SBCCI*, 2002.

[29] H. Zhang. Service disciplines for guaranteed performance service in packet-switching networks. *Proc. of the IEEE*, 83(10):1374–1396, 1995.