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

Andrei Rădulescu, John Dielissen, Santiago González Pestana, Om Prakash Gangwal, *Member, IEEE*, Edwin Rijpkema, Paul Wielage, and Kees Goossens

***Abstract*—In this paper, we present a network interface (NI) for an on-chip network. Our NI 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 NI has a modular architecture, which allows flexible instantiation. It provides both guaranteed and best-effort services via connections. These are configured via NI ports using the network itself, instead of a separate control interconnect. An example instance of this NI with four ports has an area of 0.25 mm$^2$ after layout in 0.13-$\mu$m technology, and runs at 500 MHz.**

***Index Terms*—Best-effort communication, communication protocols, network interfaces, networks on chip, packet switching, performance guarantees.**

## I. Introduction

NETWORKS-ON-CHIP (NoC) have been proposed as a solution to the interconnect problem for highly complex chips [1], [3]–[5], [9], [14], [20], [21], [23], [26], [28]–[30], [35], [42]. NoCs help designing chips in several ways. They

- structure and manage wires in deep submicron technologies [4], [5], [14], [20], [29], [35].
- allow good wire utilization through sharing [9], [14], [20], [35].
- scale better than buses [1], [21], [28], [35].
- can be energy efficient and reliable [4], [9].
- decouple computation from communication through well-defined interfaces, enabling IP modules and interconnect to be designed in isolation, and to be integrated and reused more easily [4], [18], [28], [29], [35],.[39].

Networks are composed of *routers*, which transport the data from one place to another, and network interfaces (NIs), which implement the interface to the IP modules. In a previous article [35], we showed the tradeoffs in designing a cost-effective router combining guaranteed with best-effort traffic. In this paper, we focus on the other NoC component, the NI.

On-chip NIs must provide a low area overhead, because the size of IP modules attached to the NoC is relatively small. 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 control 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 ISO–OSI stack [36]. 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 [7].

Our NI provides services at the transport layer in the ISO-OSI reference model [36], 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* [24], [39], which allows IP modules and interconnect to be designed independently from each other. We provide transport-layer services by defining connections (e.g., peer-to-peer or multicast) configured for specific properties (e.g., throughput, ordering).

We offer *guaranteed services* (e.g., lower bounds on throughput, and upper bounds on latency) 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 [18], [20], separate the IP requirements and their implementation, and make application quality of service independent of the IP and NoC implementations.

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) [38]. We adopt this protocol to provide backward compatibility to existing on-chip communication protocols (e.g., AXI [2], OCP [32], DTL [33]), and also to allow efficient implementation of future protocols, which are better suited to NoCs.

We provide a modular NI, which can be configured at design time. That is, the number of ports and their type (i.e., master port, or slave port), the number of connections at each port,

Manuscript received March 1, 2004; revised June 9, 2004. This paper was recommended by Guest Editor J. Figueras.

The authors are with the Philips Research Laboratories, 5656 AA Eindhoven, The Netherlands (e-mail: andrei.radulescu@philips.com).

Digital Object Identifier 10.1109/TCAD.2004.839493

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.

The NI allows flexible NoC configuration at runtime. Each connection can be configured individually, requiring configurable NoC components (i.e., router and NI). However, instead of using a separate control interconnect, which must be scalable too, for NoC program, the NoC is used to program itself. This is performed through configuration ports using memory-mapped input/output (DTL-MMIO) transactions [33]. The NoC an 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, we briefly cover the related work on NIs. In Section III, we describe NoC services that we implement, and the interface offered to the IP modules. In Section IV, we present a modular NI 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 Section V, we show that NoCs can be configured both in a distributed and in a centralized way, and we present the tradeoff between the two approaches. We then show how the NI allows NoC configuration using the NoC itself as opposed to via a separate control interconnect. In Section VI, we demonstrate the feasibility of our NI design through a prototype implementation in a 0.13-$\mu$m technology, and we conclude in Section VII.

## II. Related Work

NI design has received considerable attention for parallel computers [10], [13], [27], [31], [40], and computer networks [6], [8], [11], [12]. 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 high to be applicable on chip.

On-chip interconnect interfaces have been already used for decoupling computation from communication, allowing interconnect and processing cores to be designed independently and to be reused [15], [19], [25], [43] For NoC interconnects, first, NIs are just being designed.

Dally and Towles describe a tile-based NoC which provides a low-level datagram protocol [14]. The interface has an input port and an output port, each with a separate data part (256 bits) and a control part (38 and 22 bits for input and output ports, respectively). The control data includes data types, packet size, virtual channel, and, for the input port, routing information. Higher-level services can be layered on top of this interface.

Liang *et al.* present a NoC architecture in which the router and the NI are integrated in a single block, called communication interface [28]. Communication patterns are determined at compile time and programmed in the communication interface. The switch in the communication interface is programmed in every cycle based on this programming. The communication interface is estimated to have an area of $30 \times 10^6\ \lambda^2$ and to run at 350 MHz in 0.3-$\mu$m technology. This results in approximately 0.50 mm$^2$ in 0.13-$\mu$m technology.

Bolotin *et al.* present an irregular mesh NoC, called QNoC [9]. The QNoC NI offers a bus-like protocol with conventional read and write semantics. This NI offers four service levels, where signaling has the highest priority, followed by real time, read/write and block transfer.

Liu *et al.* propose the NoC to be viewed as an interconnect intellectual property module [29]. The NI of this NoC offers services at the transport layer, being responsible for message to/from packet conversion. An NI example offering a unidirectional 64-bit data only interface to a CPU is presented.

Bhojwani and Mahapatra study the impact of the packetization implementation on area, latency, complexity, and flexibility [7]. The paper shows three packetization schemes: 1) in software and 2) in hardware on core, and 3) in a wrapper. Software implementation results in high latencies and an increase of code size. On-core hardware packetization has a 13 K gates area overhead and a moderate latency: 10.8 ns. A hardware wrapper implementation has the lowest area overhead: 4 K gates, and the lowest latency: 3.02 ns.

Other NoC proposals exist, where an NI is envisaged to provide communication services to the IP modules connected to the NoC [21], [22], [26], [30], [42]. However, no design details are offered yet for these NIs.

The existing NI designs address one or more of the aspects of our NI: low cost, high-level services offering an abstraction of the NoC, modular design, and differentiated services. However, to the best of our knowledge, our NI is the first to address all these aspects. In addition to these, our NI also provides time guarantees (i.e., bandwidth and latency bounds), and offers a solution to NoC and IP modules configuration at runtime, resulting in a complete NI solution.

## III. NoC Services

As mentioned in the previous section, the communication services of the Æthereal NoC are defined to meet the following goals:

- decouple computation (IP modules) from communication (NoC);
- provide backward compatibility to existing bus protocols;
- provide support for real-time communication;
- 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 [24], [39]. In NoCs, this decoupling is achieved by positioning the network services at the transport level [5], [35] or above in the ISO–OSI reference model [36]. 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* [38]. In a transaction-based model, there are two types of IP modules: masters and slaves (see Fig. 1). 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.

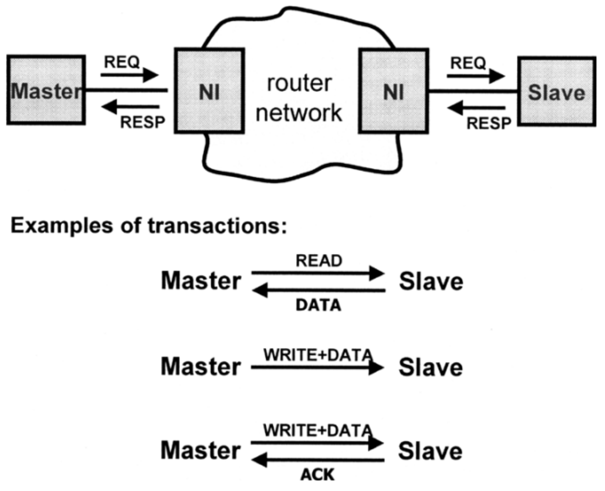


Fig. 1. Transaction model.

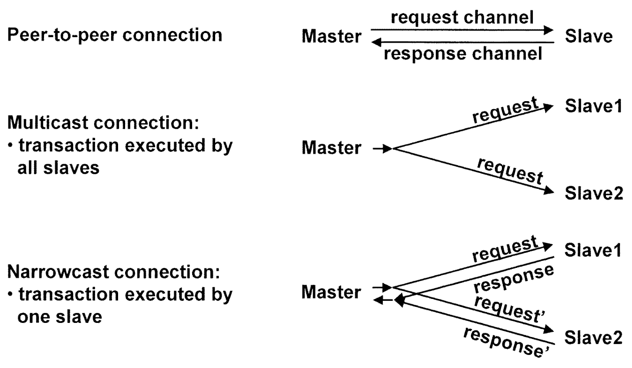


Fig. 2. Types of connections.

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 reduce the area for the switching elements. Packetization is performed by the NI and is, thus, transparent to the IP modules.

The Æthereal NoC offers its services on *connections*. The reason we use connections is to allow differentiated communication services and guarantees offered to the IP modules. Connections can be peer to peer (one master, one slave), multicast (one master, multiple slaves, all slaves executing each transaction, and, currently, no responses allowed to avoid merging of messages), and narrowcast (one master, multiple slaves, a transaction is executed by only one slave), as shown in Fig. 2 [38].

Connections are composed of unidirectional peer-to-peer *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 unordered 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.

In Æthereal, message delivery is guaranteed by not allowing network buffers to overflow [35], [37]. This is ensured using credit-based flow control at the link level to avoid router buffer overflow, and at the channel level (i.e., peer-to-peer between two NIs) to avoid NI buffer overflow.

Message ordering is offered natively by the channel implementation. Within a channel, messages are packetized and sent by the NI in order of their receipt from the IP. The packets in a channel are forced on the same path in the NoC, where they are kept in order by the routers and NIs. This choice limits the flexibility in routing the packets (e.g., no dynamic routing is possible), however, simplifies significantly NI design and reduces its cost, because there is no need to reorder within the NI.

Ordering guarantees are provided only within a channel. Different channels are treated as separate entities in the NI scheduler, and they may have different routes. As a result, message reordering is possible across channels.

Support for real-time communication is achieved by providing throughput and latency *guarantees*. These are essential for complex real-time streaming application (e.g., high-end TV chips), because they considerably reduce the integration time, and allow IP reuse.

In Æthereal, throughput and latency guarantees are 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 [18], [35]. This scheme has smaller packet buffers, and, hence, has lower implementation cost compared to alternatives, such as rate-based packet switching [44], or deadline-based packet switching [34].

Throughput guarantees are given by the number of slots reserved for a connection. A slot corresponds 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.

Protocol stacks that are used in networks to implement different 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 hip, 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 transmission errors,[1] short wires) [38]. In the Æthereal NoC, we optimize the performance and minimize the cost of

[1]In current CMOS technologies (e.g., 0.13 $\mu$m), the mean time between failure is very large. Moreover, when NoCs are used, e.g., in mesh, wires are structured, and their properties can be controlled such that the probability of transmission errors is nearly 0. Considering these, it is safe to assume there are no transmission errors in an NoC. In future technologies, the error rate is going to increase, and, consequently, additional support for error handling, such as error detection, error correction, and retransmission after error detection, might become necessary. This is a possible extension to NoCs, typically done at the data link layer, which has little impact on the higher layers, i.e., network and transport.

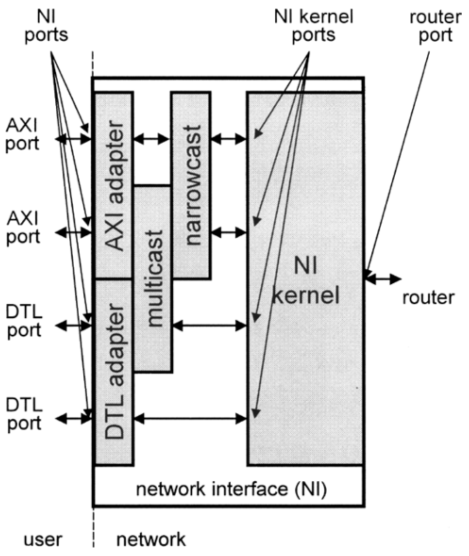


Fig. 3. NI kernel and shells.

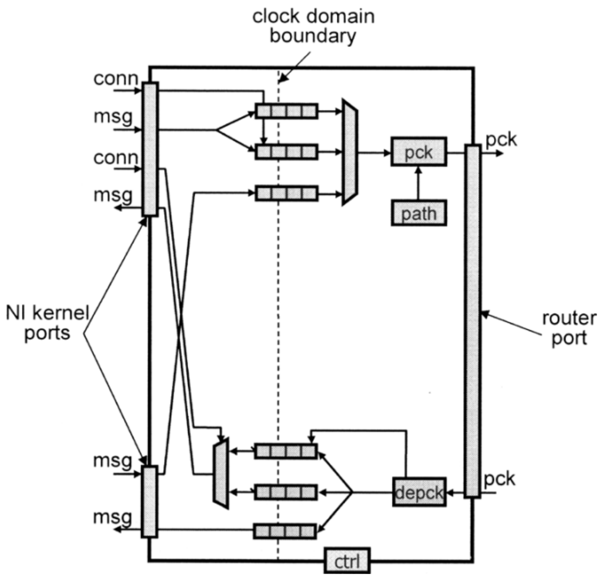


Fig. 4. NI kernel ports.

the protocol stack by implementing it in hardware, rather than in software. We support this claim in Section VI.

More on the services offered by the Æthereal NoC can be found in [38].

## IV. NI Architecture

The NI is the NoC component that implements the services described in the previous sections, and 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 NI in two parts (see Fig. 3):

1) the *NI kernel*, which implements the channels, packetizes messages and schedules them to the routers, implements the end-to-end flow control and clock domain crossings;
2) 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.

The reasons for this split in the design are

- the reuse of the NI kernel across various NI flavors.
- easy migration to various communication protocols by implementing a simple shell.
- cost optimizations by omitting the shells that are not needed.
- the option to automatically generate NI instances by assembling the kernel and various existing shells.

### A. NI Kernel Architecture

The NI kernel (see Fig. 4) receives and delivers messages from the IP modules. These messages contain the sequentialized data provided by the IP modules via their protocol. 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.

*NI Ports:* The NI kernel communicates with the NI shells via *ports*. Typically, there are multiple ports per NI, as an IP module does not transfer enough data to fill up the NoC link capacity (16 Gb/s in our current prototype).

At each port, peer-to-peer 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 `conn` signals to select on which connection a message is supplied or consumed. In Fig. 4, we show an example NI with two ports, in which the first port can have up to two connections, and the second port can only have one connection.

*Connection Implementation:* A peer-to-peer connection consists of two channels: one request channel and one response channel (see Fig. 5). The reason for this choice is to come close to existing on-chip communication protocols (AXI, OCP, or DTL), and, consequently, have low-cost shell implementations. Alternatively, unidirectional connections (i.e., consisting of a single channel) are also possible, but we decided not to use them, because the communication model they offer (message passing) is not common practice for on-chip designs.

Each channel uses two queues for storing messages, one in each of the two NI kernels. As there are two channels per connection, in each NI kernel, there are two message queues for each 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 first-in first-outs (FIFOs), and are also used to provide the clock-domain crossing between the network and the IP modules. Each NI-kernel port can, therefore, have a different clock frequency.

Each channel is configured individually. In a first prototype of the Æthereal NI (see Fig. 8), we can configure if a channel provides time guarantees (GT) or not (we call this best effort, BE), reserve slots for GT connections (in the Slot Table Unit `STU`), configure the end-to-end flow control (in the `space` counter), and the routing information (in the `conn` table). For details on how these channel properties are configured, see Section V.

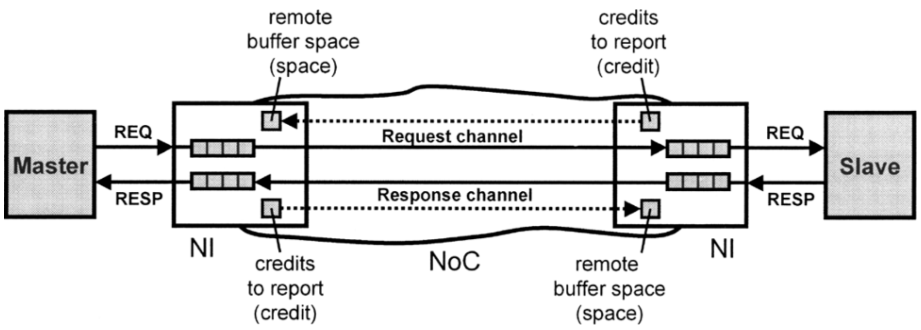

remote buffer space (space)
credits to report (credit)
Master
REQ
RESP
Request channel
Response channel
REQ
RESP
Slave
NI
credits to report (credit)
NoC
remote buffer space (space)
NI

Fig. 5. Connection implementation.

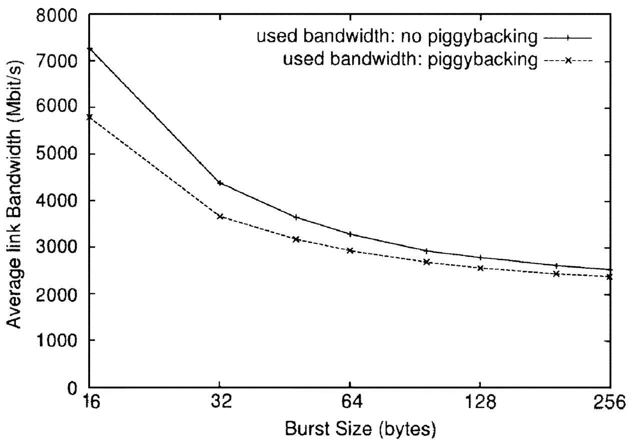


Fig. 6. Average link bandwidth.

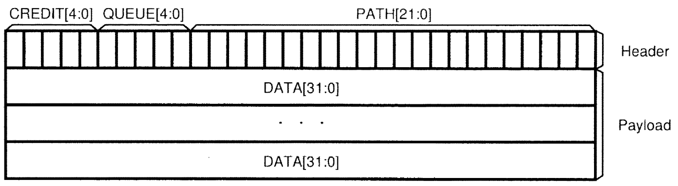


Fig. 7. Æthereal packet format.

End-to-end flow control ensures that no data is sent unless there is enough space in the destination buffer to accommodate it. The alternative of not using end-to-end flow control would lead to either (1) dropping data at the destination NI in case the buffer is full, which would add considerable complexity and cost for data retransmission, or (2) letting the data chain in the network starting at the NI's full buffer, which can lead to NoC congestion, and deadlock.

We implement end-to-end flow control using credits [41]. As shown in Fig. 5, 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 in the remote NI (`credit`) to indicate that more empty space is available. These credits are sent to the producer of data (dashed line in Fig. 5) 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. As shown in Fig. 6, which plots the average link bandwidth resulted when implementing a mpeg2 video encoder/decoder[2] using our NoC interconnect, overhead bandwidth can be reduced with up to 20% by piggybacking credits on the data packets. As the burst size grows, the curves converge (the relative overhead introduced by credit packets decreases). The reason is that the larger the burst size is, the largest amount of credits are reported in a credit packet. Consequently, the number of credit packets (i.e., overhead introduced by credits) decreases. The drawback of increasing burst sizes is that buffers need to increase to accomodate the larger bursts.

As shown in Fig. 7, besides the routing information (path and queue identifier), we use 5 bits (27 to 31) in the packet header for piggybacking the credits. Thus, the maximum amount of credits that can be sent at a time is currently $2^5 = 32$. Note that at most `space` data items can be transmitted before credits must be received. We call the minimum between the data items in the queue and the value in the counter `space`, the *sendable data*. Whenever a queue contains sendable data, the `request generator` issues a signal specifying that that queue can be scheduled.

From the source queues, data is packetized, and sent to the NoC via the router port. A packet header consists of the routing information (in the current prototype, this is the path from the source NI to the destination NI as a sequence of router ports[3]), remote queue id (i.e., the queue of the remote NI in which the data will be stored), and piggybacked credits. The packetization is controlled by the `flit ctrl` module, which indicates if the produced word is a header. The path and the remote queue id are taken from the `conn` table, and the credits to be reported are taken from `credit` table).

*NI Scheduler:* We implement a scheduler to arbitrate between the channels that have data to be transmitted (see Fig. 8). The scheduler is split in two.

1) The `GT scheduler` checks if the current slot is reserved for a GT channel. If the current slot is reserved, and there is sendable data in the queue for which the reservation has been made, that queue is scheduled.

[2]We modeled an mpeg2 encoder/decoder using main profile (4:2:0 chroma sampling) at main level (720 × 480 resolution with 15 Mb/s), supporting interlaced video up to 30 frames per second. This application consists of 15 processing cores and an external SDRAM, and has 21 streams (with an aggregated bandwidth of 3 GB/s), all configured to have guaranteed throughput.

[3]We opt for source routing because it does not require routing tables in the routers, and, thus, routing table configuration. Source routing also allows router design to be independent of the NoC topology.

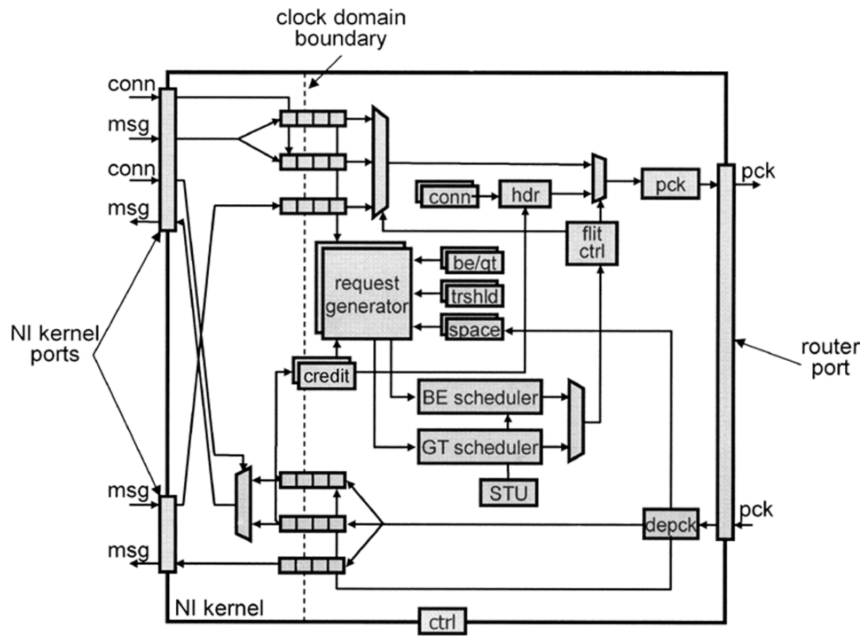


Fig. 8. NI kernel architecture.

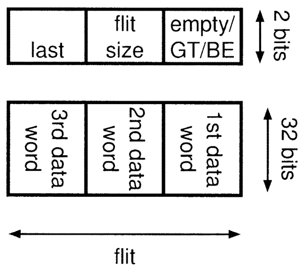


Fig. 9. Control data associated with a flit.

2) The `BE scheduler` uses round-robin arbitration to select between the BE queues which have sendable data, but only when there is no queue scheduled in the GT scheduler.

In this way, our NoC provides time guarantees (i.e., bandwidth and latency), while at the same time allows the rest of the remaining NoC capacity to be used by best-effort traffic.

*NI Kernel Optimizations:* To increase the NoC efficiency, it is preferable to send longer packets. To achieve this, we use the following two optimizations. First, the decision on when the packet is finished is taken as late as possible (by one of the `scheduler` modules). This allows newly arrived data to be attached to an ongoing packet, increasing its length, and optimizing NoC utilization. The packet end is marked using the already existing two control bits used by the routers to manage flits. In the current implementation, the link width is 32, and flits consist of three words (see Fig. 9) [35]. Consequently, there are three control values sent consecutively in a flit. The first specifies the flit type (i.e., $0 = \text{empty}$,[4] $1 = \text{GT}$, $2 = \text{BE}$), the second gives the number of words other than a packet header in the flit, and the third uses only one bit to specify if the current flit is the last of a packet (1), or not (0).

[4]When an empty flit is sent, only the two control lines are set to 0. Data lines do not toggle, and, therefore, minimum power is consumed.

Second, we implement a configurable threshold mechanism, which skips a channel as long as the sendable data is below the threshold (thresholds are stored in the `trshld` table, see Fig. 8). This is applicable to 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 the words in the queue at the time of flushing have not been sent, the threshold for that queue is bypassed.

The flush signal is controlled by the IP modules. This is a standard technique in modern communication protocols, such as DTL (which has a similar flush signal) or AXI (which forces transmission of potentially buffered write data with an unbuffered write command). For read commands, no such flush-equivalent exists, and, therefore, the NI shells always set the flush high for read commands and read data.

As shown in Fig. 10, using thresholds increases network utilization (e.g., 20% for a threshold of 16 and a burst size of 16), especially for small burst sizes, because it forces longer packets, and, hence, a lower number of packet headers. On the other hand, waiting for data to accumulate to create longer packets increases the latency (see Fig. 11), and longer packets require larger buffers (see Fig. 12). Consequently, using data thresholds involves a tradeoff between network utilization on one hand, and latency/cost on the other hand, and should be used only when necessary.

A similar threshold is set for credit transmission (also in the `trshld` in Fig. 8). 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

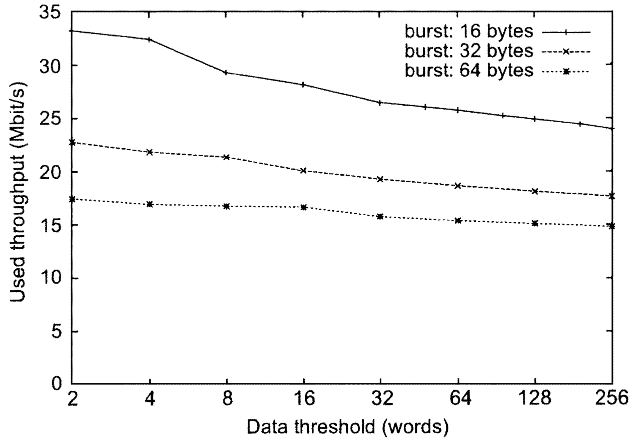


Fig. 10. Average link utilization relative to data threshold.

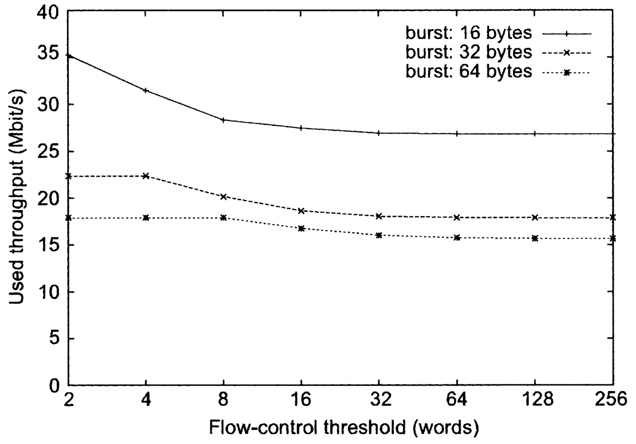


Fig. 13. Average link utilization relative to flow-control threshold.

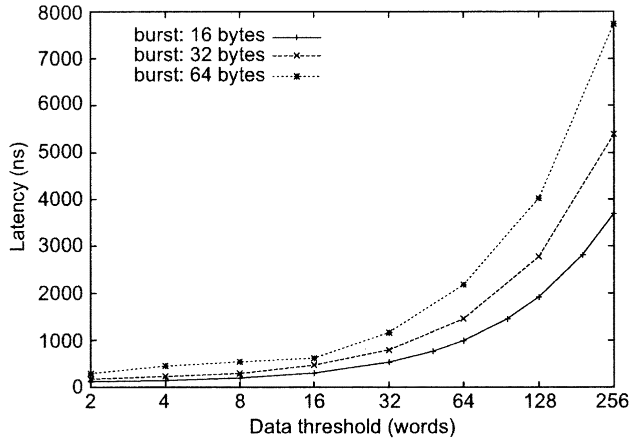


Fig. 11. Average latency relative to data threshold.

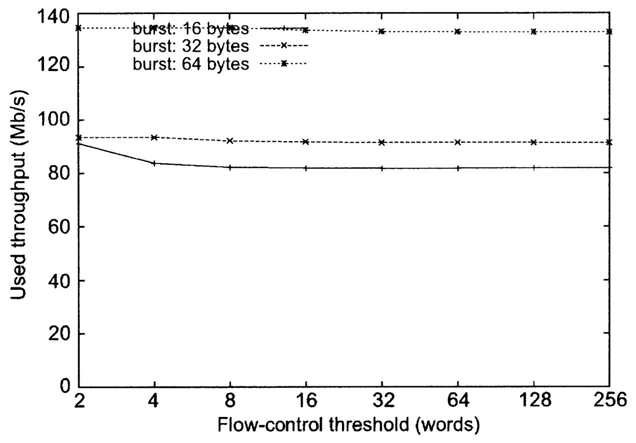


Fig. 14. Average latency relative to flow-control threshold.

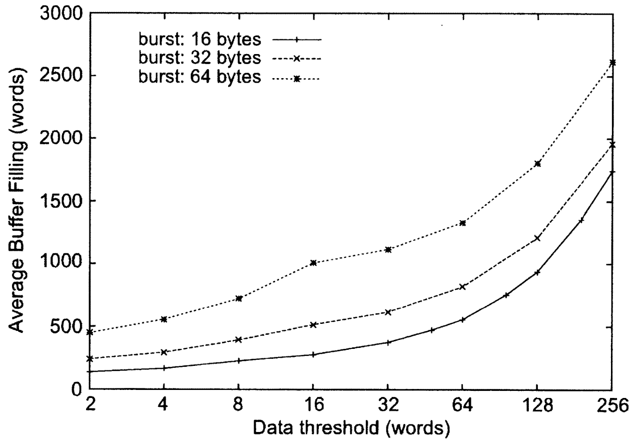


Fig. 12. Average NI buffer utilization relative to data threshold.

above the threshold. Similar to the data case, credits can also be flushed to prevent possible starvation.

As for the data thresholds, credit thresholds increase NoC utilization, especially for small burst sizes, because by forcing credit accumulation, less empty packets carrying only credits are generated (see Fig. 13, where data thresholds are set to 0). However, as opposed to the data thresholds, setting credit thresholds has little impact on the latency and buffer requirements (see Figs. 14 and 15). The reason is that the application has a periodic bursty behavior, and the time needed to report credits back (in one or multiple packets) is lower than the time between bursts. Consequently, credits are always reported in time, thus preventing data being buffered, and not affecting the latency.

As credits are piggybacked on packets, a queue becomes eligible for scheduling (i.e., `request generator` issues a signal for that queue to `GT scheduler` and `BE scheduler`) when either the amount of sendable data is above its data threshold, or when the amount of credits is above its credit 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 limited by the implementation to the given number of bits in the packet header, and BE packets have a maximum length to avoid links being used exclusively by a single packet/channel, which could cause NoC congestion and/or starvation.

For the incoming packets, the NI inspects the header, adds the credits to the counter `space`, and stores the data (without the header) in the queue specified by the queue id field in the packet header. The data is then ready for consumption by the shells at the NI-kernel ports.

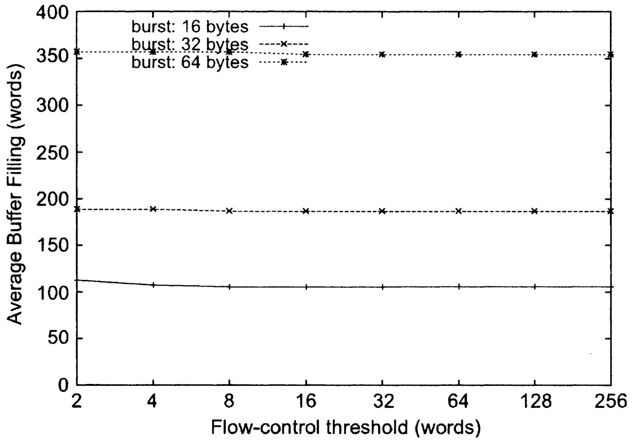


Fig. 15. Average NI buffer utilization relative to flow-control threshold.

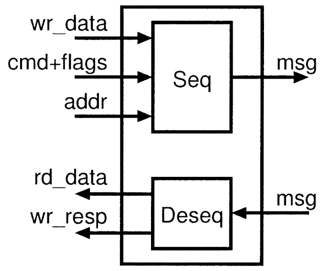


Fig. 16. Master shell.

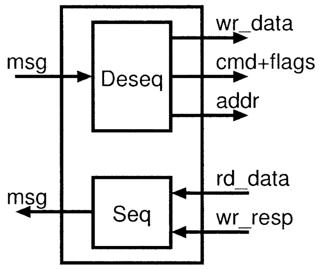


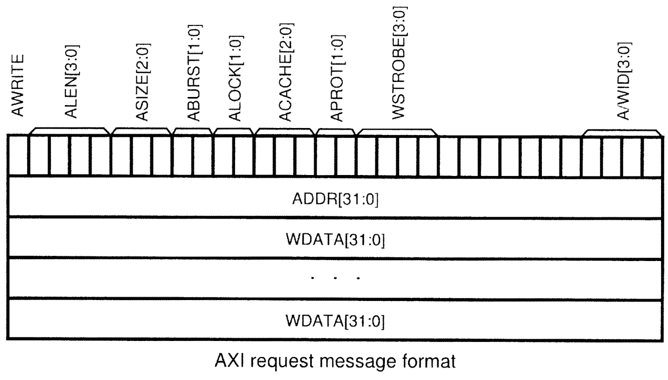
Fig. 17. Slave shell.

## B. NI Shell Architectures

With the NI kernel described in the previous section, peer-to-peer connections (i.e., between one master and one slave) can be supported directly. These types of connections are useful in systems involving chains of modules communicating peer to peer with one another (e.g., video pixel processing [16], [19]).

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 Fig. 3, we show a NI 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 instantiation time, according to the requirements. NoC instantiation is simple, as we use an XML description to automatically generate the VHDL code for the complete NoC, including the NI, router and the NoC topology [17].

*Master/Slave Shells Architecture:* In Figs. 16 and 17, we show a master and slave shells that implement a simplified version of a protocol such as DTL or AXI. Support for different protocols is possible due to the fact that NI kernel is protocol agnostic. The basic functionality of such shells is to (de)sequentialize commands and their flags, addresses, and write data in request messages, and to (de)sequentialize messages into read data, and write responses. Master shells sequentialize request messages and desequentialize response messages, while slave shells sequentialize response messages and desequentialize request messages.



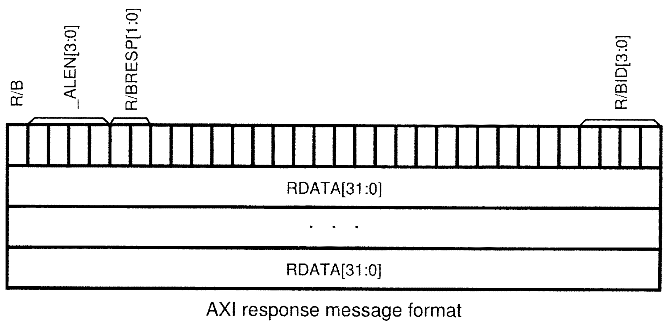


Fig. 18. AXI message format examples.

In Fig. 18, we show an example of 32-bit message structures for the AXI protocol resulting after sequentialization. These messages are passed from NI shells to the NI kernel. For the request message, the command (AWRITE) and all its flags are included in the first word of the message, the address (ADDR) is set in the second word, and the write data (WDATA), in the case of a write command, are appended at the end. There is one limitation in this encoding compared to the original AXI protocol: the strobe is identical for all transferred write words in a burst, while in AXI each write word have an individual strobe. If individual strobes for each word are required, this can be implemented either by extending the link width from e.g., 32 to 36 bits to also accommodate the strobes, or by adding extra strobe words in the message formats. This functionality leads, however, to increased cost and/or overhead.

For the response message, there is a bit (R/B) specifying if the message corresponds to a read data or to a write response. R/BRESP[1:0] indicates if the transaction is successful or not. In the case of a read data response, _ALEN[3:0] is also copied in the response message by the slave AXI NI shell to reduce the NI buffering cost. Additionally, if multiple connections are implemented at the NI port, a connection id can be generated based on the transaction's address or thread id.

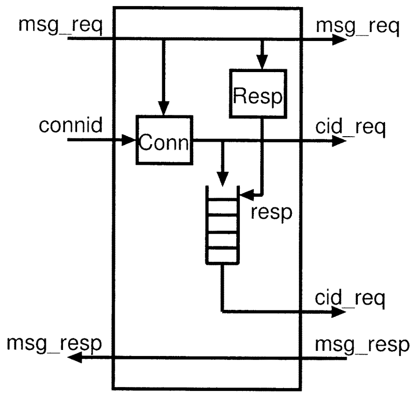


Fig. 19. Narrowcast shell.

*Narrowcast Shell Architecture:* In Fig. 19, 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 [38]. 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 peer-to-peer 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 each slave is run-time configurable in the narrowcast module. In-order delivery per slave of request messages is provided natively by the Æthereal NoC for the peer-to-peer connections. To provide in-order response delivery, the narrowcast shell keeps a history of connection identifiers of the transactions with responses (e.g., reads, and acknowledged writes), and the length of these responses. That allows the responses being received in order from each of the slaves to be interleaved correctly before being delivered to the master.

*Multichannel Shell Architecture:* When a slave using a connectionless protocol (e.g., DTL) is connected to an NI port supporting multiple connections, a multiconnection shell must be included to arbitrate between the connections. A multiconnection shell (see Fig. 20) includes a scheduler to select connections from which request messages are consumed, based for example, on their filling. As for the narrowcast, the multiconnection shell has a connection id history for mapping back the responses from the slave to their corresponding connections.

## V. 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/closing connections implies allocating/deallocating resources for communication. For example, a connection requires buffering resources, it is associated an identifier, it is configured a memory map, and it may possibly be allocated a priority and/or a portion of the bandwidth. In our NoC, these resources are allocated in the network components by writing registers via a control port using a standard protocol, such as AXI or DTL-MMIO.

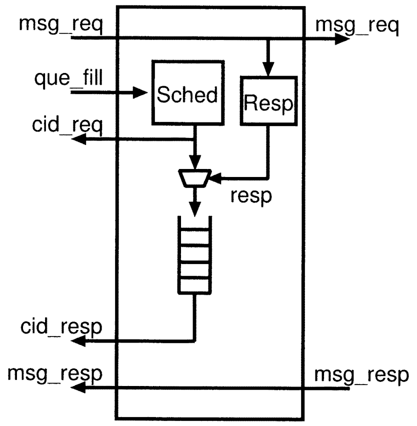


Fig. 20. Multiconnection shell.

Resource allocation requires arbitration for the case in which two connections request the same resources. This arbitration can be performed either centralized or distributed. In the centralized case, there is only one resource manager, which opens and closes all connections. In this case, the cost of managing resources is lower, and can produce better results. However, a centralized resource manager does not scale and cannot be used for large NoCs. In the distributed case, there are multiple resource managers distributed in the NoC, which arbitrate locally for their resources. This is a scalable resource management solution. However, the associated cost is higher.

In the current prototype of the Æthereal NoC, we opt for centralized configuration, because it is able to satisfy the needs of a small NoC (around ten routers), and has a simpler design and lower cost. More specifically, when a centralized configuration scheme is used, our routers do not require any configuration (they are stateless and identical). The reasons are as follows.

1) We use source routing, and, thus, the routing information is present in the packet headers.
2) In our slot allocation scheme, it is enough to reserve slots in the source NI for a given path to guarantee the bandwidth for that path.

In this way, router design is simplified, leading to an approximately 30% lower-cost router, at the price of introducing headers for the guaranteed throughput traffic too.

In general, opening a connection between two NIs requires setting up two channels (one request channel and one response channel). For each channel, only the source NI needs to be configured. Consequently, opening a connection between two NIs results in configuring these two NIs.

Setting up one channel consists of two parts:

1) finding the free slots and selecting those to be allocated to the channel;
2) writing the registers of the channel in the source NI.

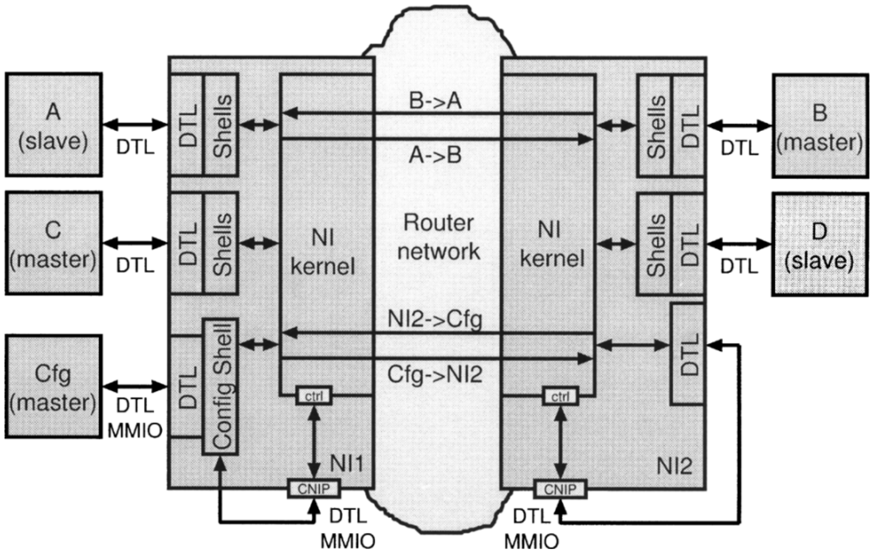


Fig. 21. NI configuration.

Finding and selecting slots is performed traversing the slot tables of each link along the path from the source NI to the destination NI, and selecting the slots required to accommodate the required bandwidth. In the case not enough free slots are available, an error is returned to the programmer.

Slot finding and selection is proportional to the length of the path and the slot table size. We implemented a software prototype centralized implementation for an ARM, which takes $NoHops \times \lceil NoSlots/32 \rceil \times 15$ ns on an ARM running at 200 MHz, where $NoHops$ is the channel path length, and $NoSlots$ is the slot table size. For example, computing the slot allocation for a channel with a path of length 4 and a 128 slot table takes 240 ns.

Writing the registers of a channel implies executing write transactions over the NoC. This is again proportional to the NoC speed, NoC size (i.e., the distance between the NI where the configuration is performed and the NI to be configured), and the number of registers to be written (three for normal channel configuration, two in case the channel is attached to a narrowcast shell, and $NoSlots$ for slot reservation). If individual write transactions are used for writing registers, the time take to configure a channel is approximately $FlitSize \times (NoRegisters + 1 + NoHops)/NocFreq$ ns, where $NoRegisters$ is the number of registers to be written, $flitSize$ is the number of words in a flit (three in Æthereal case), $NoHops$ is the channel path length, and $NocFreq$ is the frequency at which the NoC runs (500 MHz for Æthereal). For a path of length 4, part of a narrowcast connection, for which six consecutive slots have been reserved (resulting in a reserved bandwidth of 90 MB/s), the configuration time is 90 ns.

### A. NI Configuration

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 standard 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 Fig. 21). 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.

Connections are set up by writing the proper values in NI registers. To configure a connection between two modules (e.g., from master `B` and slave `A` in Fig. 21), the NIs to which the modules are attached must be configured. These NIs are configured either directly (e.g., NI1 via CNIP), or by using a configuration connection (e.g., NI2 via the connection Cfg→NI2).

As NoC configuration is quite elaborate, and is susceptible to introducing programming errors, we have also implemented a high-level API for NoC configuration. This API provides primitives for transparently opening and closing NoC connections. Besides simplifying the NoC programming, this API allows the configuration code to be independent of the NoC implementation.

## VI. Implementation

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

### A. NI Area

The Æthereal NI is modular and can be configured at design time. That is, the number of ports and their type (i.e., configuration port, master port, or slave port), the number of channels 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.

```
<NetworkInterface id="NI00" slots="8">
  <SlaveNIP id="CONFIG" protocol="DTL" channels="1" iqdpth="8" oqdpth="8" />
  <MasterNIP id="IP1" protocol="DTL" channels="1" iqdpth="8" oqdpth="8" />
  <MasterNIP id="IP2" protocol="DTL" channels="2" iqdpth="8 8" oqdpth="8 8" />
  <SlaveNIP id="IP3" protocol="DTL" channels="4" iqdpth="8 8 8 8" oqdpth="8 8 8 8" />
</NetworkInterface>
```

Fig. 22. Example of topology description.

We use an XML description to generate the RTL-VHDL code for instances of our NoC, where the configuration of the NIs and routers, as well as their interconnect is specified. The resulting VHDL code is compliant with the Philips internal design flow. In Fig. 22, we show part of such an XML description, which specifies one NI instance. This instance has a slot table with eight slots, and four DTL ports, two master ports, and two slave ports (listed as MasterNIP and SlaveNIP entries in the XML description). For each of these ports, the number of channels that can be configured is specified in the channels attribute: 1, 1, 2, and 4, respectively. For each of these channels, the input and output queue depths are specified by the "iqdpth" and "oqdpth" attributes, respectively.

We have synthesized a NI kernel with this description. 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. Finally, 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 [35], and delivers a bandwidth toward the router of 16 Gb/s in each direction. The synthesized area for this NI-kernel instance is 0.136 mm$^2$ in a 0.13-$\mu$m technology.

Narrowcast and multiconnection shells have an area of 0.004 and 0.007 mm$^2$, corresponding to 3% and 5% of the NI kernel area. The DTL shells are very small, 0.005 and 0.002 mm$^2$ for the master and slave ports, corresponding to 3.5% and 1.5% 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 interface to configure the NoC, has an area of 0.01 mm$^2$.

Summing up, the total area of an NI is

$$\begin{aligned} NIArea = {} & KernelArea + SlaveCfgShellArea \\ & + NumMasters \times MasterShellArea \\ & + NumSlaves \times SlaveShellArea \\ & + NumNCMasters \times NCShellArea \\ & + NumMCSlaves \times MCShellArea \end{aligned}$$

where $KernelArea$ is the kernel area, $SlaveCfgShellArea$ is the area of the slave shell used to configure the NI, $NumMasters$, $NumSlaves$, $NumNCMasters$, $NumMCSlaves$ are the number of masters, the number of slaves, the number of masters with multiple connections and the number of slaves with multiple connections, respectively, and $MasterShellArea$, $SlaveShellArea$, $NCShellArea$, $MCShellArea$ are the master shell, slave shell, narrowcast shell, and multichannel shell areas, respectively.

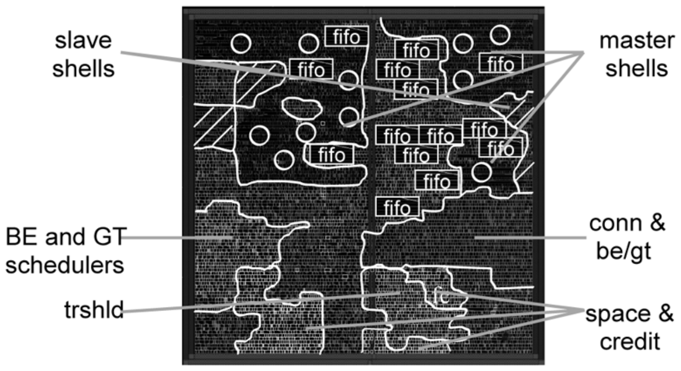


Fig. 23. Six-port NI layout.

For the above example, NI with four ports, one for configuration (one channel to which the configuration shell is attached), two masters (one offering narrowcast with two channels), and one slave (multichannel with four channels), the total area is $0.136+0.01+2\times0.005+0.004+0.002+0.007 = 0.169$ mm$^2$.

The layout of the NI instance described above, which has an area of 0.25 in a 0.13-$\mu$m technology. The increase in area compared to the synthesized area is due to the utilization (70%), and the power ring (not normally needed when the NI is part of a larger design).

In Figs. 23 and 24 we show a detailed view of the area consumption of different parts of the NI. The figures show a NI with three single-connection master ports, four single-connection slave ports of which one is used for NI configuration, where all queues are eight words deep. One can note that for this NI instance, a large part of the area is consumed by the FIFOs (27%). Master and slave shells take 26% of the NI area. The tables for channel configuration (i.e., conn, be/gt, trshld, space and credit) occupy 19% of the NI area. The BE and GT schedulers occupy 6.5% of the NI area. The rest of 21.5% is consumed by other logic, such as packetization/depacketization, multiplexing of data, or credit management.

If the NI has only single-connection ports, its area is approximately proportional to the number of ports. The nonproportional part consists of the master and slave shells, which have different cost, and the area in misc. In Fig. 25, we show the area of the NI, when we vary the number of master and slave single-connection ports.

## B. NI Latency

The latency introduced by our current NI is two cycles in the DTL master shell (due to sequentialization, as part of packetization), 0 to 2 in the narrowcast and multicast shells (de-

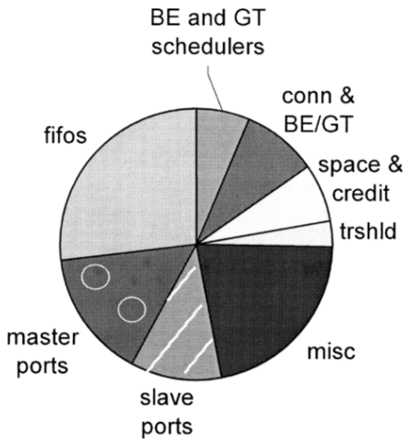


Fig. 24. Six-port NI area details.

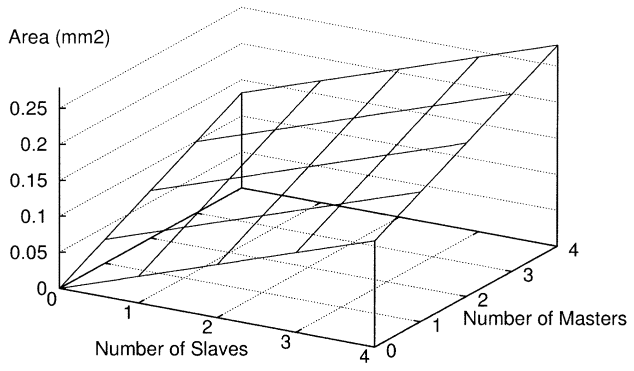


Fig. 25. NI area.

pending on the NI instance), and between 1 and three cycles in the NI kernels (as data needs to be aligned to a three word flit boundary), and two clock cycles for clock domain crossing. Additional delay is caused by the arbitration, but we do not include this in the NI latency, 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 four and ten 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 [7]).

A second advantage of a hardware implementation is that it allows both legacy software and hardware task implementations to be used without change. For example, a legacy DTL or AXI module can be connected directly to a network as it has been connected to a bus. However, if parts of the protocol stack are implemented in software (e.g., ordering or packetization), a programmable module (e.g., CPU, FPGA) would need to be attached to the NI to execute the software protocol parts. If no such CPU exists, the legacy modules could not be used, or would need to be changed to implement the missing parts (e.g., produce data in a packetized format directly if packetization is done in software).

## VII. Conclusion

In this paper, we describe an NI architecture which offers high-level services at a low cost. Our NI 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 NI has a modular design, composed of a kernel and several 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: 1) additional functionality, such as multicast and narrowcast connections and 2) 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 NoC, which is used to automatically generate the RTL–VHDL code for the NoC, including the NIs, routers and the network topology.

We show an instance of our NI, which shows that the cost of implementing our protocol stack in hardware is small (0.25 mm$^2$ after layout 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 NI offering a shared-memory abstraction, high-level services (including guarantees), which allows runtime network configuration using the network itself.

## References

[1] A. Adriahantenaina, H. Charlery, A. Greiner, L. Mortiez, and C. A. Zeferino, "SPIN: A scalable, packet switched, on-chip micro-network," in *Proc. Design Automation Test Eur.*, 2003.
[2] ARM, AMBA AXI Protocol Specification, Mar. 2004.
[3] N. Banerjee, P. Vellanki, and K. S. Chatha, "A power and performance model for networks-on-chip architectures," in *Proc. Design Automation Test Eur.*, Feb. 2004.
[4] L. Benini and G. De Micheli, "Powering networks on chips," in *Proc. ISSS*, 2001.
[5] ——, "Networks on chips: a new SoC paradigm," *IEEE Comput.*, vol. 35, no. 1, pp. 70–80, Jan. 2002.
[6] R. A. F. Bhoedjang, T. Rühl, and H. Bal, "User-level network interface protocols," *IEEE Comput.*, vol. 31, no. 11, pp. 53–59, Nov. 1998.
[7] P. Bhojwani and R. Mahapatra, "Interfacing cores with on-chip packet-switched networks," in *Proc. VLSI Design*, 2003.
[8] G. Blair, A. Campbell, G. Coulson, F. Garcia, D. Hutchison, A. Scott, and D. Shepherd, "A network interface unit to support continuous media," *IEEE J. Select. Areas Commun.*, vol. 11, no. 2, 1993.
[9] E. Bolotin, I. Cidon, R. Ginosar, and A. Kolodny, "QNoC: QoS architecture and design process for network on chip," *J. Syst. Architec.*, vol. 50, pp. 105–128, Jan. 2004.
[10] G. Buzzard, D. Jacobson, S. Marovich, and J. Wilkes, "Hamlyn: A high-performance network interface with sender-based memory management," in *Proc. Hot Interconnects*, 1995.
[11] T. Callahan and S. C. Goldstein, "NIFDY: a low overhead, high throughput network interface," in *Proc. ISCA*, 1995.
[12] A. Chien, M. Hill, and S. Mukherjee, "Design challenges for high-performance network interfaces," *IEEE Comput.*, vol. 31, no. 11, pp. 42–44, Nov. 1998.
[13] D. J. Culler, J. P. Singh, and A. Gupta, *Parallel Computer Architecture: A Hardware/Software Approach.* San Francisco, CA: Morgan Kaufmann, 1999.

[14] W. J. Dally and B. Towles, "Route packets, not wires: on-chip interconnection networks," in *Proc. Design Automation Conf.*, 2001.

[15] S. Dutta, R. Jensen, and A. Rieckmann, "Viper: a multiprocessor SOC for advanced set-top box and digital TV systems," *IEEE Des. Test Comput.*, vol. 18, no. 5, pp. 21–31, <<MONTH?>> 2001.

[16] O. P. Gangwal, J. Janssen, S. Rathnam, E. Bellers, and M. Duranton, "Understanding video pixel processing applications for flexible implementations," in *Proc. Euromicro DSD*, 2003.

[17] S. G. Pestana, E. Rijpkema, A. Rădulescu, K. Goossens, and O. P. Gangwal, "Cost-performance trade-offs in networks on chip: a simulation based approach," in *Proc. Design Automation Test Eur.*, Feb. 2004.

[18] K. Goossens, J. Dielissen, J. van Meerbergen, P. Poplavko, A. Rădulescu, E. Rijpkema, E. Waterlander, and P. Wielage, "Guaranteeing the quality of services in networks on chip," in *Networks on Chip*, J. Nurmi, H. Tenhunen, J. Isoaho, and A. Jantsch, Eds. Norwell, MA: Kluwer, 2003, ch. 4, pp. 61–82.

[19] K. Goossens, O. P. Gangwal, J. Röver, and A. Niranjan, "Interconnect and memory organization in SOCs for advanced set-top boxes and TV—evolution, analysis, and trends," in *Design for Advanced SoC and NoC*, J. Nurmi, H. Tenhunen, J. Isoaho, and A. Jantsch, Eds. Norwell, MA: Kluwer, 2004, ch. 15, pp. 399–423.

[20] K. Goossens, J. van Meerbergen, A. Peeters, and P. Wielage, "Networks on silicon: combining best-effort and guaranteed services," in *Proc. Design Automation Test Eur.*, 2002.

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

[22] A. Jalabert, S. Murali, L. Benini, and G. De Micheli, "XpipesCompiler: a tool for instantiating application specific networks on chip," in *Proc. Design Automation Test Eur.*, 2004.

[23] F. Karim, A. Nguyen, and S. Dey, "An interconnect architecture for networking systems on chip," *IEEE Micro*, vol. 22, no. 5, 2002.

[24] K. Keutzer, S. Malik, A. R. Newton, J. M. Rabaey, and A. Sangiovanni-Vincentelli, "System-level design: orthogonalization of concerns and platform-based design," *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 19, no. 12, pp. 1523–1543, Dec. 2000.

[25] P. Klapproth, "General architectural concepts for IP core re-use," in *Proc. ASP-DAC*, 2002.

[26] S. Kumar, A. Jantsch, J.-P. Soininen, M. Forsell, M. Millberg, J. Öberg, J. Tiensyrjä, and A. Hemani, "A network on chip architecture and design methodology," in *Proc. ISVLSI*, 2002.

[27] W. S. Lee, W. J. Dally, S. W. Keckler, N. P. Carter, and A. Chang, "An efficient protected message interface," *IEEE Comput.*, vol. 31, no. 11, pp. 69–74, Nov. 1998.

[28] J. Liang, S. Swaminathan, and R. Tessier, "aSOC: A scalable, single-chip communications architecture," in *Proc. PACT*, 2000.

[29] J. Liu, L.-R. Zheng, and H. Tenhunen, "Interconnect intellectual property for network-on-chip (NoC)," *J. Syst. Architec.*, vol. 50, no. 1, pp. 65–79, 2004.

[30] M. Millberg, E. Nilsson, R. Thid, and A. Jantsch, "Guaranteed bandwidth using looped containers in temporally disjoint networks within the Nostrum network on chip," in *Proc. Design Automation Test Eur.*, Feb. 2004.

[31] S. S. Mukherjee and M. D. Hill, "A Survey of User-Level Network Interfaces for System Area Networks," Univ. Wisconsin, Madison, Tech. Rep. 1340, 1997.

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

[33] Philips Semiconductors, Device Transaction Level (DTL) Protocol Specification. Version 2.2, Jul. 2002.

[34] J. Rexford, "Tailoring router architectures to performance requirements in cut-through networks," PhD dissertation, Ann Arbor, 1999.

[35] E. Rijpkema, K. Goossens, A. Rădulescu, J. Dielissen, J. van Meerbergen, P. Wielage, and E. Waterlander, "Trade offs in the design of a router with both guaranteed and best-effort services for networks on chip," in *Proc. Design Automation Test Eur.*, 2003.

[36] M. T. Rose, *The Open Book: A Practical Perspective on OSI*. Englewood Cliffs, NJ: Prentice-Hall, 1990.

[37] A. Rădulescu, J. Dielissen, K. Goossens, E. Rijpkema, and P. Wielage, "An efficient on-chip network interface offering guaranteed services, shared-memory abstraction, and flexible network programming," in *Proc. Design Automation Test Eur.*, 2004.

[38] A. Rădulescu and K. Goossens, "Communication services for networks on chip," in *Domain-Specific Processors: Systems, Architectures, Modeling, and Simulation*, S. Bhattacharyya, E. Deprettere, and J. Teich, Eds. New York: Marcel Dekker, 2004, ch. 10, pp. 193–213.

[39] M. Sgroi, M. Sheets, K. Keutzer, S. Malik, J. Rabaey, and A. Sangiovanni-Vincentelli, "Addressing the system-on-a-chip interconnect woes through communication-based design," in *Proc. Design Automation Conf.*, 2001.

[40] P. Steenkiste, "A high-speed network interface for distributed-memory systems: architecture and applications," *ACM Trans. Comput. Syst.*, vol. 15, no. 1, pp. 75–109, 1997.

[41] A. S. Tanenbaum, *Computer Networks*. Englewood Cliffs, NJ: Prentice-Hall, 1996.

[42] D. Wiklund and D. Liu, "SoCBUS: Switched network on chip for hard real time embedded systems," in *Proc. IPDPS*, 2003.

[43] D. Wingard, "Socket-based design using decoupled interconnects," in *Interconnect-Centric Design for Advanced SoC and NoC*, J. Nurmi, H. Tenhunen, J. Isoaho, and A. Jantsch, Eds. Norwell, MA: Kluwer, 2004.

[44] H. Zhang, "Service disciplines for guaranteed performance service in packet-switching networks," *Proc. IEEE*, vol. 83, no. 10, pp. 1374–1396, Oct. 1995.

**Andrei Rădulescu** received the M.Sc. degree in computer science from the Polytechnica University of Bucharest, Bucharest, Romania, in 1995 and the Ph.D. degree in computer engineering from the Delft University of Technology, Delft, The Netherlands, in 2001.

Since 2001, he has been with Philips Research, Eindhoven, The Netherlands, working on on-chip and off-chip networks, quality of service, protocols, resource mapping and scheduling, and distributed systems.

**John Dielissen** received the M.Sc. degree with honors in electrical engineering from the Eindhoven University of Technology, Eindhoven, The Netherlands, in 2000.

He is a Research Scientist with Philips Research Laboratories, Eindhoven, The Netherlands. His research interests include on-chip communication in large digital system chips, with an emphasis on networks on chip.

**Santiago González Pestana** received the MSc degree in information technology from Tampere University of Technology, Finland, in 2001 and the M.Sc. degree in telecommunication engineering from Las Palmas de Gran Canaria University, Spain, in 2002.

Since 2002, he has been granted with a Marie Curie Fellowship at Philips Research, Eindhoven, The Netherlands. His research interests include networks-on-chip, network benchmarking, and resource mapping and scheduling.

**Om Prakash Gangwal** (A'01–M'01) received the M.Tech. degree in VLSI design tools and technology from the Indian Institute of Technology, Delhi, India.

He is a Senior Scientist with Philips Research Laboratories, Eindhoven, The Netherlands. His research interests include system-level design, embedded-system architectures, video signal processing, networks on chip and communication architectures for SoCs.

**Edwin Rijpkema** received the M.Sc. degree in electrical engineering from the Delft University of Technology, Delft, The Netherlands, in 1995 and the Ph.D. degree in computer science from Leiden University.

Since 2002 he has been a Senior Scientist at Philips Research Laboratories, Eindhoven, The Netherlands. His research interests include digital signal processing and networks-on-chip.

**Paul Wielage** received the M.Sc. degree (*summa cum laude*) in electrical engineering from the Delft University of Technology (DUT), Delft, The Netherlands, in 1991.

From 1992 to 1996, he was an Associate Research Scientist with the Electrical Engineering Department, DUT. In 1996, he joined the Natlab, Philips Research Laboratories, Eindhoven, The Netherlands. His current research interests are on-chip communication, embedded memories, and system-level timing solutions.

**Kees Goossens** received the Ph.D. degree from the University of Edinburgh, Edinburgh, The Netherlands, in 1993.

He has been with Philips Research, Edinburgh, The Netherlands, since 1995, where he has led the research on networks on chip for consumer electronics, where real-time performance, predictability, and costs are major constraints.