An On-Chip Interconnect and Protocol Stack for Multiple Communication Paradigms and Programming Models

Andreas Hansson
Electronic Systems Group
Eindhoven University of Technology
Eindhoven, The Netherlands
m.a.hansson@tue.nl

Kees Goossens
Corporate Research Department
NXP Semiconductors
Eindhoven, The Netherlands
kees.goossens@nxp.com

ABSTRACT

A growing number of applications, with diverse requirements, are integrated on the same System on Chip (SoC) in the form of hardware and software Intellectual Property (IP). The diverse requirements, coupled with the IPs being developed by unrelated design teams, lead to multiple communication paradigms, programming models, and interface protocols that the on-chip interconnect must accommodate.

Traditionally, on-chip buses offer distributed shared memory communication with established memory-consistency models, but are tightly coupled to a specific interface protocol. On-chip networks, on the other hand, offer layering and interface abstraction, but are centred around point-to-point streaming communication, and do not address issues at the higher layers in the protocol stack, such as memory-consistency models and message-dependent deadlock.

In this work we introduce an on-chip interconnect and protocol stack that combines streaming and distributed shared memory communication. The proposed interconnect offers an established memory-consistency model and does not restrict any higher-level protocol dependencies. We present the protocol stack and the architectural blocks and quantify the cost, both on the block level and for a complete SoC. For a multi-processor multi-application SoC with multiple communication paradigms and programming models, our proposed interconnect occupies only 4% of the chip area.

Categories and Subject Descriptors: B.4.3 [Input/Output and Data Communications]: Interconnections – *Topology*

General Terms: Design, Performance

Keywords: System on Chip, Programming model, Network on Chip, Protocol stack

1. INTRODUCTION

Systems on Chip (SoC) grow in complexity with an increasing number of *independent applications* on a single chip [35]. The applications are realised by hardware and software Intellectual Property (IP), e.g. processors and application code. A number of trends, relating to the communication paradigms and programming models, can be seen in SoCs. First, different IP components (hardware and software) in the same system are often developed by *unrelated design teams* [16], either in-house or by independent vendors. Second, applications are often split into multiple *tasks* running concurrently, either to improve the power dissipation [32] or performance [36]. Third, SoCs are evolving in the direction of *distributed-memory* architectures, offering high throughput and low latency [22, 39], coupled with a low power consumption [23]. Fourth, address-less *streaming communication* between IPs is growing in importance to alleviate contention for shared memories and is becoming a key aspect in achieving efficient parallel processing [18, 40].

The system in Figure 1 serves to exemplify the trends. Two applications are mapped to the SoC: a video *decoder* and an audio post-processing *filter*. The applications are implemented in software, and mapped to hardware IP from different vendors, also using different interface protocols, e.g. AXI [2] for the ARM, PLB [45] for the μblaze, and DTL [31] for the IP from NXP. The decoder application is split into tasks and mapped to the ARM and VLIW. The ARM reads the encoded input from the SRAM and performs the first decoding steps. The VLIW performs the remaining steps and writes the output to the video tile. The decoder uses distributed memory (the SRAM and local memory in the VLIW tile) for inter-task communication and for private data. The filter application is mapped to the μblaze. Samples are communicated to and from the audio tile by means of streaming communication and private data is stored in the SRAM, which is shared with the decoder.

Based on the trends, we identify the following three requirements for the on-chip interconnect: 1) Due to the diversity in origin and requirements, the interconnect *must accommodate multiple communication paradigms and programming models* [19, 29]. 2) The use of distributed memories places requirements on the interconnect that *must support an established memory consistency model* [30], e.g. release consistency [10], to allow the programmer to reason about the order of memory operations. 3) To enable the use of existing IP, the interconnect *must support one or more industry-standard interfaces* and be easy to extend.

Existing on-chip interconnects are typically based either on buses or Networks on Chip (NoC). Buses offer distributed shared memory communication with established memory consistency models. However, buses have limited support for streaming communication and are typically tailored for

CODES+ISSS'09, October 11–16, 2009, Grenoble, France.
Copyright 2009 ACM 978-1-60558-628-1/09/10 ...$10.00.

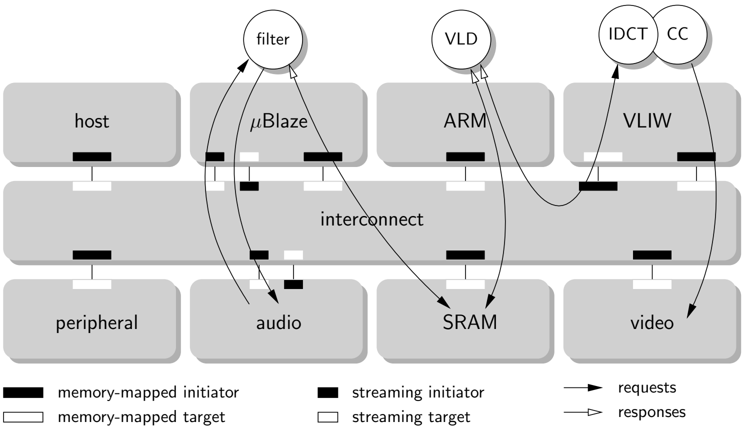


Figure 1: Example system.

one specific interface protocol, with major impacts on IP reusability. Moreover, buses are not scalable [17]. NoCs address the scalability of buses and provide layered communication [3] with more flexibility in the choice of an interface protocol. However, most NoCs are tailored for point-to-point streaming communication, and do not address higher-level protocol issues, such as ordering and dependencies between connections, that affect the memory consistency model and may introduce message-dependent deadlock [12].

For an example of the aforementioned issues, consider our example system in Figure 1, where the ARM communicates data to the VLIW via a buffer in the SRAM, and the buffer administration (used for synchronisation) is placed in the local memory of the VLIW [24]. Already in this simple example of distributed memory the interconnect must offer mechanisms to ensure that the data is written to the SRAM (and not somewhere in the interconnect) before the administration is updated. Our example system also highlights the issue of message-dependent deadlock, as neither the filter nor the decoder application adhere to a strict request-response protocol. The *message-dependency chains* [37] created by the filter and decoder contain also request-request dependencies (and thus cannot be safely mapped to e.g. [3,38]).

As the main contribution of this work, we present an on-chip interconnect and protocol stack that, in a structured way, combines local buses and a network, thus supporting multiple communication paradigms and programming models. We describe the rationale behind the proposed stack and the subdivision of the architectural building blocks and highlight their important qualitative properties. We quantify the cost and performance of the proposed interconnect (and stack) by means of synthesis results for the building blocks. We also demonstrate that the interconnect occupies only 4% of the chip area for a complete SoC example.

The rest of the paper is organised as follows. First we review related work in Section 2. Then, Section 3 gives an overview of the proposed stack and interconnect, with more details following in Sections 4 and 5, respectively. Experimental results are presented in Section 6, where after we conclude in Section 7.

2. RELATED WORK

Much work on NoCs is focused on the router network and does not address communication at the IP level. For example, networks with adaptive routing [27] typically ignore the ordering even within a point-to-point connection and it is unclear how to offer any established programming model.

NoCs that provide ordered point-to-point communication are presented in [3, 7, 26, 28, 34, 41, 44]. The NoC in [28] offers time-triggered exchange of application-layer messages. Communication must take place at a priori-determined instants, placing many constraints on the IP behaviour. These constraints are removed in [7, 26, 41] where the IPs interface with the NoC using OCP. However, OCP is used as a data-link layer protocol and the works ignore higher-level protocol issues like ordering and dependencies between connections.

Distributed and shared memory communication is addressed in [3, 17, 34, 38, 44]. However, neither of the works give any details on how to support multiple communication paradigms, i.e. combine streaming and memory-mapped communication, and also do not show how to implement a specific memory-consistency model. The issue of memory consistency and ordering is addressed in [30] by only allowing one outstanding transaction. This, however, is overly restrictive and severely impairs the interconnect performance.

Protocol stacks for NoCs are proposed in [3, 5, 6, 19, 20]. The stacks are focused on the lower layers (up to the transport layer), and do not address issues relating to synchronisation and dependencies between connections that takes place at the session layer [9]. Moreover, to the best of our knowledge, no NoC offers a complete protocol stack for both streaming and distributed shared memory communication.

With more elaborate programming models, it is necessary to address message-dependent deadlock [37]. Most NoCs rely on *strict ordering* with separate physical or logical networks [3,38], thus severely limiting the programming model, e.g. to pure request-response protocols. End-to-end flow control is proposed in [12] to avoid placing any restrictions on the dependencies between connections outside the network, thus avoiding message-dependent deadlock irrespective of the programming models used by the IPs.

Extending on [11, 34], our proposed interconnect enables multiple communication paradigms and programming models, with the mechanisms required to implement release consistency, and flexibility in the choice of IP interfaces. This is accomplished by: 1) clearly separating the network stack, the streaming stack and the memory-mapped stack, both logically and physically by *combining the network with lo-*

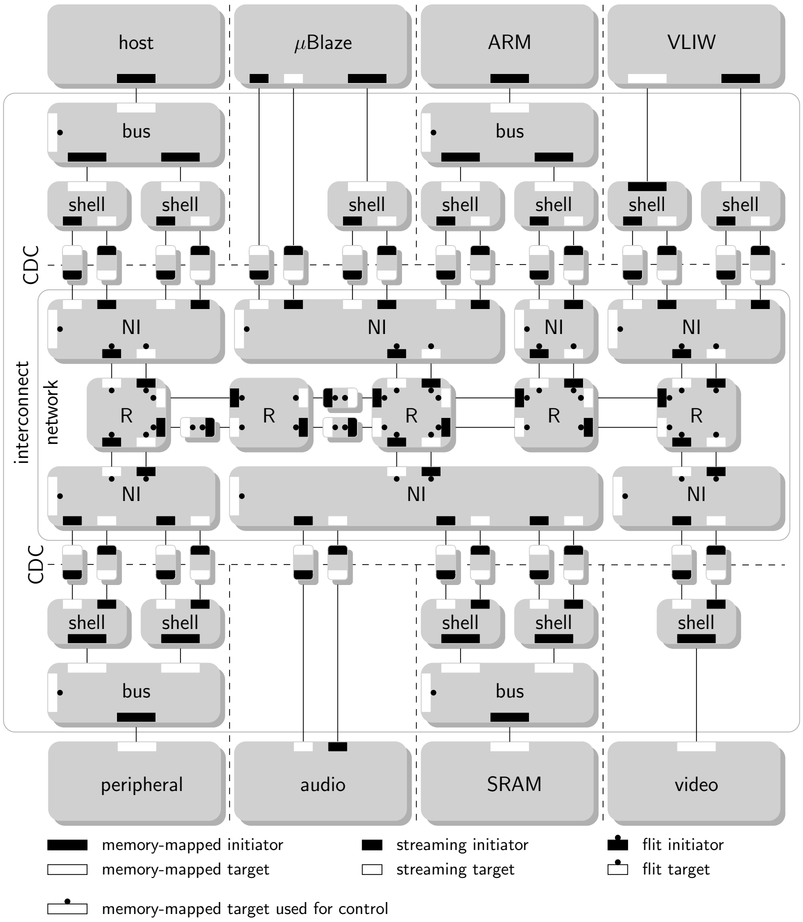


Figure 2: Interconnect architecture overview.

cal buses, 2) allowing any higher-level programming model without introducing message-dependent deadlock by *breaking all connection inter-dependencies with end-to-end flow control* inside the network.

3. OVERVIEW

In this section we introduce the interconnect building blocks and give a brief example of their functionality. Figure 2 illustrates the same system as Figure 1, but now with an expanded view of the interconnect, dimensioned for the decoder and filter applications introduced in Section 1.

To illustrate the functions of the different blocks, consider a load instruction that is executed on the ARM. The instruction causes a bus *transaction*, in this case a read transaction, to be initiated on the *memory-mapped initiator* port of the processor. Since the ARM uses distributed memory, a *target bus* forwards the read *request message* to the appropriate initiator port of the bus, *based on the address*. The *elements* that constitute the request message, i.e. the address and command flags in the case of a read, are then serialised by a *target shell* into individual words of streaming data. The streaming data is fed via a Clock Domain Crossing (CDC) into the Network Interface (NI) *input queue* of a specific *connection*. The data items reside in the queue until the NI schedules the connection. The streaming data is *packetised* and injected into the router network as *flow control digits (flits)*. The flits are forwarded *in order* through the network. Once the flits reach the destination NI, their payload, i.e. the streaming data, is put in the NI *output queue* of the connection. After another clock domain crossing, an *initiator shell* represents the ARM as a memory-mapped initiator by reassembling the request message. If the destination target port is not shared by multiple initiators, the shell is directly connected to it, e.g. the video tile in Figure 2. For a shared target, such as the SRAM, the request message is forwarded to an *initiator bus* that *arbitrates* between different initiator ports. Once granted, the request message is forwarded to the target, here the SRAM, and a response message is generated. The response message is sent back through the bus to the initiator shell. The shell serialises the response message into streaming data that is sent back through the network. On the other side of the network, the response message is reassembled by the target shell and forwarded to the target bus. The target bus enforces a transaction ordering according to the IP port protocol and also has mechanisms such as tagging [31] to enable programmers to choose a specific *memory-consistency model*. Once all ordering dependencies are resolved, the bus forwards the response to the ARM, thus completing the load.

We continue by introducing the protocol stack and discussing the rationale behind it where after which we show how the interconnect implements the different stacks.

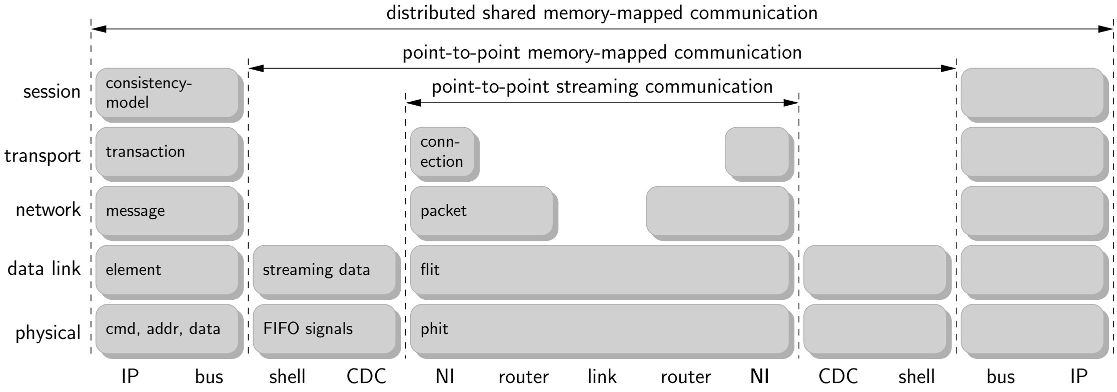


Figure 3: Interconnect protocol stack.

4. PROTOCOL STACK

The proposed stack, shown in Figure 3, is divided into five layers according to the seven-layer Open Systems Interconnection (OSI) reference model [9]. As seen in the figure, the memory-mapped, streaming and network communication each have their own stack, bridged by the shell and the NI. We discus the three stacks in turn, bottom up.

4.1 Network stack

The network stack is similar to what is proposed in [5, 6, 19, 20, 33]. The NI is on the transport layer as it maintains end-to-end (from the perspective of the network) *connections* and guarantees ordering within, but not between connections. A connection is a bidirectional point-to-point inter-connection, between two pairs of initiator and target streaming ports on the NIs. Two uni-directional *channels*, one in each direction, connect the two pairs of ports. The router is at the network layer as it performs switching of *packets*. The last element of the network, the link, is at the data-link layer and is responsible for the clock synchronisation and flow control involved in the transport of *flits*. The physical layer is governed by the *physical digit (phit)* format.

The requirements placed on the network architecture is that it: 1) offers *in-order and loss-less* communication, 2) is *free of routing deadlock*, and 3) that no *inter-connection dependencies* outside the network lead to cyclic resource dependencies inside the network. Most NoCs satisfy the first two requirements, and a few NoCs satisfy also the third requirement [12]. Using any of these NoCs, the network behaves as a collection of distributed and independent FIFOs, with data entering at a streaming target port, and later appearing at a streaming initiator port (as determined by the resource allocation). As illustrated in Figure 3, the NI bridges between the streaming stack and network stack by establishing connections between streaming ports and by embedding streaming data in network packets. Thus, the network stack is completely hidden from the IPs, and only used by means of the streaming stack, about which more presently.

4.2 Streaming stack

The streaming stack is far simpler than the network stack, and only covers the two lowest layers. The NI, clock domain crossing, shell, and IPs with streaming ports (like the μblaze in our example system or the video blocks in [40]) all make direct use of this stack. The data-link layer governs the flow control of individual words of *streaming data*. The streaming ports make use of a simple FIFO interface with a valid and accept handshake of the data. The physical layer concerns the FIFO signal interface. For robustness, the streaming interfaces use *blocking flow control* by means of back pressure. That is, writing to a streaming target port that is not ready to accept data (e.g. due to a full FIFO) or reading from a streaming target port that has no valid data (e.g. due to an empty FIFO) causes a process to stall.

The requirements placed on the streaming interfaces (of the network) are that they: 1) have *no interpretation or assumptions* on the time or value of the individual words of streaming data, and 2) operate independently without any *ordering restrictions*. Both the aforementioned properties are key in enabling multiple communication paradigms and programming models.

The most basic use of the streaming stack is exemplified in Figure 2 by the μblaze that communicates with the audio tile directly via the streaming ports of the NI, using *point-to-point streaming communication* (NI to NI). We now look at how memory-mapped communication is implemented on top of the streaming stack.

4.3 Memory-mapped stack

In contrast to the simple FIFO interface of the streaming ports, memory-mapped protocols are based on a request-response transaction model and typically have dedicated groups of wires for command, address, write data, and read data [2, 25, 31, 45]. Many protocols also support features like byte enables and burst transactions (single request multiple data elements). The block that bridges between the memory-mapped ports of the IPs and the streaming ports of the NIs is a protocol *shell*, that serialises the request and response messages. As illustrated in Figure 3, the shells enable *point-to-point memory-mapped communication* (shell to shell) by bridging between the *elements* of the bus-protocol, e.g. the address, command flags or individual words of write data, and words of streaming data by implementing the data-link protocol of both the stacks.

The requirements placed on the shells are: 1) only protocol *translation on the data-link layer* is performed in the shells (and no multiplexing, arbitration, ordering, etc), 2) the shells operate independently, thus enabling *multiple memory-mapped protocols* to co-exist by allowing different pairs of memory-mapped initiators and targets to communicate using different protocols.

Next, we show how distributed and shared memory-mapped communication is enabled by placing buses between the IPs and the shells.

4.3.1 Distributed shared memory

As demonstrated by the ARM in Figure 2, a memory-mapped initiator port often uses distributed memory [34], and accesses multiple targets, based on e.g. the address, the type of transaction or dedicated identifier signals in the interface [25]. The outgoing requests must be directed to the appropriate target, and the incoming responses *ordered* and presented to the initiator according to the protocol. In addition to the use of distributed memory at the initiator ports, memory-mapped target ports are often shared by multiple initiators, as illustrated by the SRAM in Figure 2. A shared target must be *arbitrated*, and the initiators' transactions multiplexed according to the protocol of the port.

On the session layer in the memory-mapped stack we find the *memory-consistency model*, as it governs the ordering and synchronisation between transactions (and hence connections). Note that this layer is completely left out in existing NoC stacks [3, 5, 6, 19, 20] and that it depends on the particular memory-mapped protocol. In this work we address the problem by adding buses *outside the network* (despite all the multiplexing and arbitration *inside the network*). This division of the stack (and architecture) enables us to bridge between protocols on the lower layers, something that involves far fewer challenges than doing so on the session layer, as proposed in [19, 34]. Moreover, by not adding a session layer to the network stack (but instead add a memory-mapped stack) it is possible to support multiple different memory-mapped stacks (protocols) without any modifications to the network. At the network layer in the bus stack we have *messages*, e.g. requests and responses. It is the responsibility of the bus to perform the necessary multiplexing and direct messages to the appropriate (local) destination. Each message is in turn constructed of *elements* and the data-link layer is responsible for the flow control and synchronisation of such elements. Finally, the physical layer governs the signals of the bus interface.

The requirements placed on the buses (besides adhering to the specific protocol) are that they: 1) address all ordering and synchronisation *between connections*, i.e. all functionality on the *session layer*, 2) offer the necessary mechanisms to enforce a memory-consistency model. The choice of a consistency model is thus left for the IP (and bus) developer, i.e. the interconnect provides the mechanisms, and it is up to the individual IP to implement a specific policy.

Thanks to the clear separation of protocol shells and buses it is possible to reuse available buses (library IP) and existing functional blocks for, e.g. word-width and endianness conversion, or instrumentation and debugging [42], and these blocks can be developed and verified independent of the network with established protocol-checking tools such as Cadence Specman. Furthermore, thanks to the simple interface between shell and NI, the shells belonging to one IP port, e.g. the shells of the ARM in Figure 2, can easily be distributed over multiple NIs, e.g. to provide higher throughput or lower latency.

We now continue by looking at how the proposed stack is implemented by the architectural blocks of the interconnect.

5. ARCHITECTURE

In this section we present the building blocks of the architecture bottom up, emphasising the properties that are of importance for the protocol stack. We start with a brief overview of the network in Section 5.1. We continue with a description of the clock domain crossings in Section 5.2, followed by the protocol shells in Section 5.3 and the local buses in Section 5.4.

5.1 Network

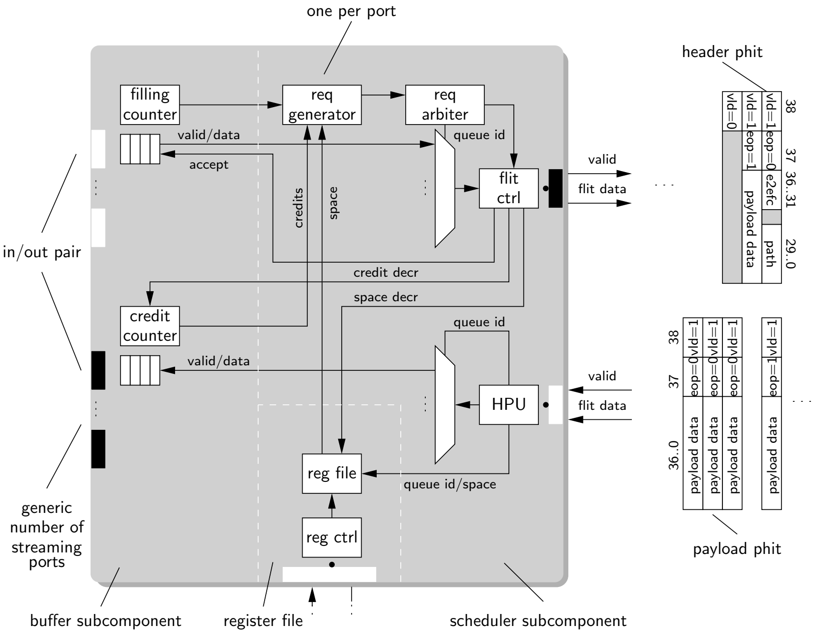
The network, consisting of NIs, routers and links, connects the streaming ports on the shells and IPs over logical connections. As already discussed in Section 4, the network is responsible for providing each connection with in-order and loss-less data transmission, free of routing deadlock. It should also not restrict any inter-connection dependencies outside the network. The details of the network are outside the scope of this work and we give a brief overview focusing on the aforementioned requirements.

5.1.1 Dependencies and ordering restrictions

Figure 4(a) illustrates the NI architecture, and highlights the subcomponents together with their control and data dependencies. Starting from the streaming ports on the left, each channel has a dedicated FIFO in both the sending and receiving NI and *credit-based end-to-end flow control* is used to ensure freedom from message-dependent deadlock [12]. Each target streaming port corresponds to an *input queue*, with data from the IP to the network. Similarly, an initiator FIFO port corresponds to an *output queue* and a *credit counter*. The latter (conservatively) tracks the number of words freed up in the FIFO that are not yet known to the sending NI. The dedicated FIFOs in combination with end-to-end flow control ensures that the network is free of message-dependent deadlock *irrespective of any inter-connection dependencies outside the network*. The drawbacks of the proposed architecture is that buffer sizes must be determined at design time (given the application requirements [15]). Moreover, the NI must maintain additional counters, and communicate credits between sending and receiving NIs. As we shall see, credits are sent as part of the packet headers.

5.1.2 In-order loss-less communication

When data or credits are present in the input and output queue, respectively, the request generator for that port informs the arbiter. The arbiter decides from which port data or credits are sent the next flit cycle. After the arbiter, the flit control unit is responsible for constructing the flits for the port decided by the arbiter. In the other direction, for flits coming from the router network, the Header Parsing Unit (HPU) decodes the flits and deliver credits to the appropriate space counter, and data to the appropriate output queue. To ensure in-order delivery, the path through the network is determined by *source routing*. That is, the path is embedded in the *packet header*, as shown in Figure 4(b). The benefit of source routing is that it is a straight forward technique to ensure in-order transmission, and the problem of *routing-deadlock avoidance is pushed to the path allocation*. Source routing does, however, require the insertion of headers, and the path encoding typically restricts the topology. Furthermore, to support multiple use-cases, the paths must be run-time reconfigurable.



(a) (b)

Figure 4: NI architecture (a) with associated flit format (b).

In our interconnect, the path field is 30 bits and holds a sequence of output ports, encoding the path through the router network and lastly the port in the destination NI. Along each hop, the router or NI looks at the lowest bits (corresponding to the 2-logarithm of its arity) of the path and then shifts those bits away. Not having fixed bit fields, as used in e.g. [4] (where a hop is always represented by three bits), *allows arbitrary topologies* with a varying number of ports per router and NI. The register file of the NI is programmable through a memory-mapped target port, enabling run-time reconfiguration of the paths and the space counters required for the end-to-end flow control.

5.1.3 Interpretation of and assumptions on contents

The flit control and HPU are not only involved in sending header phits, but also payload phits. These phits have no meaning to the NIs or routers, and are simply forwarded, *without any interpretation or assumptions* on the contents. Note that this requirement is violated in NoCs that rely on virtual circuits to avoid message-dependent deadlock [12], where the flits have control information about the messages they carry to enable the network to put them in the appropriate buffers or use specific scheduling algorithms. The benefit of having a network that is oblivious to the contents is that there is no coupling between the network and e.g. the messages of a specific bus protocol. The main drawback is that it complicates monitoring and debug of the network, e.g. the interpretation of the contents of an NI queue.

We conclude that the NIs (and routers) satisfy the requirements of the network stack and streaming stack in Section 4.

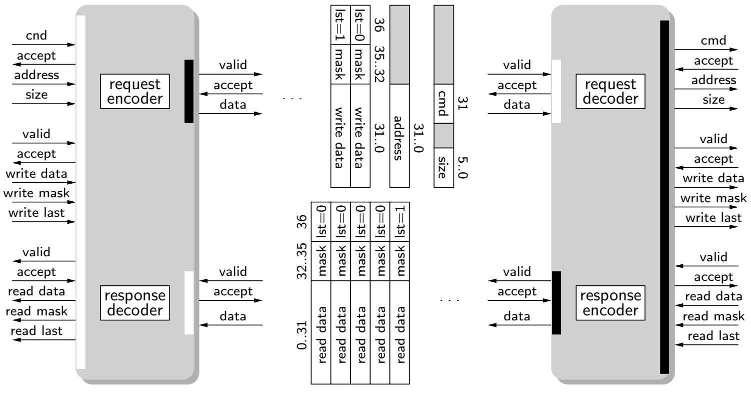
5.2 Clock domain crossings

The clock domain crossings are not important in satisfying the requirements of the protocol stacks, but illustrate the benefits of the clear interface separation. Thanks to the streaming interface between the protocol shells and the NIs, the clock domain crossings can be implemented using existing bi-synchronous FIFOs [8, 21, 43] (although additional measures must be taken in clock/reset distribution, testing, etc). For simplicity, and compatibility with both ASIC and FPGA design flows, this work uses a gray-code pointer-based FIFOs [8] to implement the clock domain crossings. The FIFOs are compatible with standard CAD tools and let the IPs (or shells) robustly interface with the network with high throughput (one transfer per clock cycle) and a small latency overhead (two clock cycles)

5.3 Protocol shells

The protocol shells bridge between memory-mapped ports and the streaming ports of the network. As seen in Figure 2, the shells are connected either directly to the IPs (on the μblaze, VLIW and video tile), to the target buses (on the bus of the ARM), or to the initiator buses (on the bus of the SRAM). For a specific memory-mapped protocol there is a target shell, an initiator shell, and their associated message formats, as shown in Figure 5. Consider, for example, the DTL target shell in Figure 5(a). The request encoder awaits a valid command (from the initiator port connected to the shell), then serialises the command, burst size, address and flags. In the case of a write, it serialises the data elements, together with the mask (byte enables) and the write-last signal. In the initiator shell, the request decoder does the opposite, and drives the command, address and burst size signals. If the transaction is a write, the initiator shell also presents the write data along with its masks and the last signal. Responses follow a similar approach in the opposite direction. The initiator shell encodes the read data, together with the mask and last signal, only later to be decoded and reassembled by the target shell.

Two properties of the shell are important for the stack.



(a) (b) (c)

Figure 5: Target shell (a) and initiator shell (c), with associated message formats (b).

First, the layer of operation that is tightly coupled to the dependencies and parallelism in the memory-mapped protocol. Second, the independent operation that enables support for multiple memory-mapped protocols. We now discus the properties in turn.

5.3.1 Layer of operation

The division into an independent request and response part, as shown in Figure 5, places the shell at the *data-link layer*, and is suitable for protocols like DTL, PLB, where requests and responses are split. Thus, for DTL (and PLB), where the command group for read and write is shared, we use one connection per memory-mapped initiator and target pair. Consequently, the initiator and target shell in Figure 5 each have one pair of streaming ports.

For protocols like AHB and OPB, with limited or no support for split transactions, there is a much a tighter coupling between the requests and responses, placing the shell at the *transport layer*. The proposed interconnect supports such protocols, but the parallelism offered by the interconnect cannot be used efficiently and offers little or no benefit.

In the other end of the spectrum are protocols that offer more parallelism than what one pair of streaming ports (i.e. one connection) offers. For protocols like OCP or AXI, with independent read and write channels, two connections are used, one for read and one for write transactions. Moreover, with support for multiple independent threads [2, 25], each thread can be given its own connection(s). Our separation of the protocol stacks allows the thread identifier [2] and connection identifier [25] to be used at other granularities than the connections. Hence, the shells *enable different amounts of parallelism for different protocols.*

5.3.2 Support for multiple protocols

Normally, the streaming protocol is narrower, i.e. uses fewer wires, than the memory-mapped protocols. Thus, as exemplified by the command group and write data group in Figure 5(b), the signal groups of the memory-mapped interfaces are (de)serialised. The proposed message format is tailored for DTL, but is suitable also for similar protocols like PLB. Different shells may use different message formats, *allowing multiple memory-mapped protocols to co-exist.*

We conclude that the shells satisfy the requirements of the streaming stack and memory-mapped stack in Section 4.

5.4 Local buses

The final building blocks of the interconnect are the local buses. Distributed memory communication is implemented by the *target bus*, as described in Section 5.4.1. The target bus is complemented by the *initiator bus* that implements shared memory communication, as elaborated on in Section 5.4.2. Next, we describe the buses in more detail.

5.4.1 Target bus

A target bus, as shown in Figure 6(a), connects *a single memory-mapped initiator to multiple targets.* The target bus is multiplexer based and very similar to an AHB-Lite layer [1]. The primary responsibility of the target bus is to direct requests to the appropriate target, based on the address of the request. To reduce the negative impact of latency, the target bus allows multiple outstanding transactions, *even to different targets.* The target bus also enforces response ordering according to the protocol specification. That is, responses are returned in the order the requests where issued (within a thread [2, 25], if applicable). As seen in Figure 6(a), the ordering of responses is enforced by storing a target identifier for every issued request. These identifiers are then used to control the demultiplexing of responses. The ordering guarantees of the target bus, together with mechanisms like tagging [31] and acknowledged writes [2], are leveraged by the IP, e.g. through barrier instructions in the ARMv5 instruction set, to implement a certain memory-consistency model.

As exemplified by the ARM in Figure 2, a target bus is directly connected to all initiator ports that use distributed memory communication. Each target bus is individually dimensioned by determining the number of concurrent targets accessed by the initiator it is connected to. Traditional bus-based systems require the designer to determine *which targets* should be reachable and what static address map to use. We only have to determine *how many targets* should be reachable and not which ones. At run-time, the address decoder is reconfigured through the memory-mapped control port shown in Figure 6(a). Thus, the address map is determined locally per target bus, and per use-case.

5.4.2 Initiator bus

An initiator bus, as shown in Figure 6(b), connects *multiple memory-mapped initiators to a single target.* The initia-

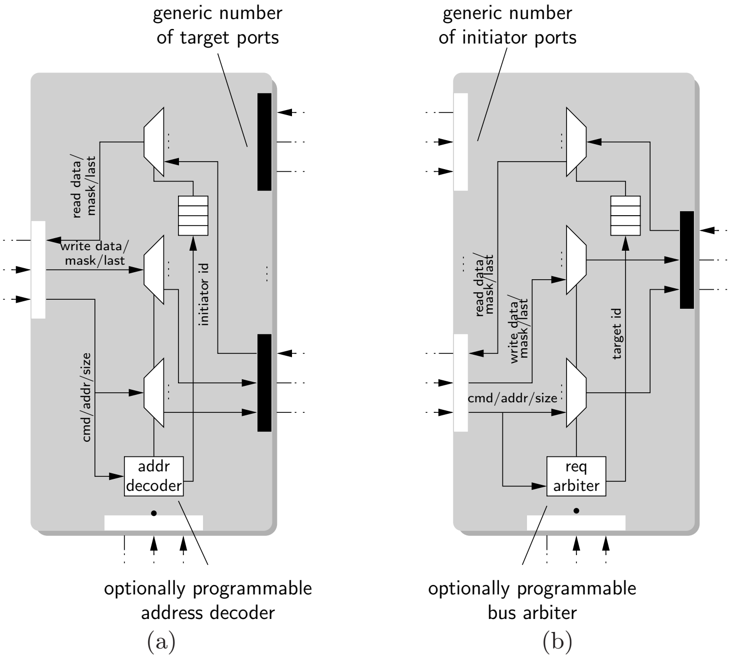


Figure 6: Local target bus (a) and initiator bus (b) architectures.

tor bus is responsible for demultiplexing and multiplexing of requests and responses, respectively. Similar to the target bus, the initiator bus implements transaction pipelining with in-order responses. It is the primary responsibility of the initiator bus to *provide sharing and arbitration* of the target port. Note that the arbitration in the initiator buses is decoupled from the arbitration in the network, and that different initiator buses can have different arbiters.

Initiator buses are placed in front of shared target ports, as exemplified by the SRAM in Figure 2. Similar to the target buses, each initiator bus is dimensioned individually based on the maximum number of concurrent initiators sharing the target port. Similar to the target buses, the initiator buses are reconfigured at run-time using memory-mapped control ports. Depending on the arbiter (round-robin, TDM, etc), the bus is run-time configured with e.g. an assignment to TDM slots, the size of a TDM wheel, or budget assignments for more elaborate arbiters.

We conclude that the buses satisfy the requirements of the memory-mapped stack in Section 4.

6. EXPERIMENTAL RESULTS

In the previous sections we have shown how the required qualitative properties are implemented by the building blocks. For our quantitative evaluation we look at the cost in terms of silicon area, and the ability to provide low latency and high throughput communication.

Synthesis results are obtained using Cadence Ambit with Philips 90 nm low-power libraries. We disable clock-gate insertion as well as scan insertion and synthesise under worst-case commercial conditions. All results reported throughout this work are *before place-and-route*, and include *cell area only*. The width of the data (and address) interfaces for all memory-mapped and streaming ports are 32 bits.

6.1 Individual blocks

We start by looking at the individual blocks, bottom up, and then look at a complete SoC instance.

6.1.1 Network

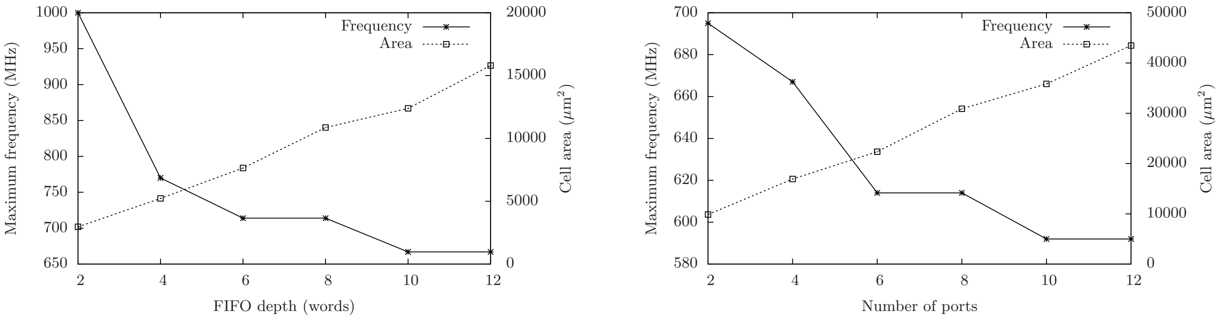
We adopt the router architecture proposed in [14], which occupies only 0.015 mm^2 in a 90 nm technology. With the router in place, we continue with the proposed NI, for which we split the synthesis into two parts and look at the buffers separately. The reason for the division is that the buffers grow independently with their depth.

All NI buffers are of a uniform width, 37 bits, corresponding to the message format of the shells, as discussed in Section 5.3. The synthesis results in Figure 7(a) shows the maximum frequency and the associated cell area for a 37-bit wide fully synchronous, pointer-based Flip-Flop FIFO of varying depth. We see that the larger FIFOs achieve maximum frequencies of around 650 MHz. The area grows linearly with the depth, as expected. The size of the FIFOs depends on the application, as discussed in [15]. As we shall see, the NI buffers have a large impact on the total area.

The synthesis results for the arbiter subcomponent and register files are shown in Figure 7(b). There is a considerable constant part that does not change with the number of ports (the HPU and flit control), but still an 8-port NI occupies only 0.1 mm^2 and runs at roughly 650 MHz.

6.1.2 Clock domain crossing

The bi-synchronous FIFO scales to high clock speeds, achieving more than 700 MHz for a 37-bit wide and 3-word deep FIFO that occupies roughly 5000μm^2. More efficient implementations of bi-synchronous FIFOs, with lower latency and area requirements, are presented in [21,43]. These FIFOs could be used as part of our interconnect to improve the performance and reduce the cost, as we shall see.



(a)

(b)

Figure 7: Area and frequency for FIFOs (a) and the remaining NI (b).

6.1.3 Protocol shells

The DTL initiator and target shell achieve a frequency of 833 MHz, with an area of 2606 and 2563 μm^2, for initiator and target shell respectively.

6.1.4 Local buses

Figure 8(a) shows the synthesis results of the target bus as the number of initiator ports is varied. Two different bus instantiations are evaluated, with a programmable and fixed address decoder, respectively. In both cases, we allow a maximum of four outstanding responses. The first thing to note about the results in Figure 8(a) is that the maximum frequency for the two architectures is the same. This is due to the fact that the address decoding is pipelined in both cases. Even with 12 ports, the target bus runs at more than 550 MHz, which is sufficient for most contemporary IPs. The area for the target bus is a minuscule 0.01 mm^2.

The synthesis results in Figure 8(b) show how an initiator bus with a non-programmable round-robin arbiter (the most complex arbiter available in the current implementation) and a maximum of four outstanding responses scales with the number of target ports.[1] Similar to the target bus, the total cell area is in the order of 0.01 mm^2, even when the operating frequency is pushed to the maximum. Moreover, even the larger instances of the bus run at more than 550 MHz, which is more than what most IPs require.

6.2 System instance

After having shown the cost and performance of the individual building blocks, we continue by constructing a proof-of-concept system instance, with a diverse set of applications, including the filter and decoder introduced in Section 1. The applications use both streaming and memory-mapped communication, and make use of distributed memory. The hardware platform is an extension of Figure 1, containing five VLIW cores with local data and instruction memories. The system instance also contains two display controllers, two memory controllers interfacing with off-chip SRAM, analog video input, an audio ADC/DAC, USB connectivity for interaction with a PC, a debug interface, and a wide range of peripherals, e.g. a touch screen, timers, push buttons and a character display. In total there are five different clock domains on the chip.

The applications together have 35 connections, with throughput and latency requirements given per connection. The interconnect is automatically dimensioned for the requirements of the specific applications [13, 15] additional connections are added for run-time reconfiguration of the interconnect and the configurable IPs , i.e. the memory-mapped target ports used for control, as shown in Figure 2.

The entire system occupies roughly 30 mm^2. The interconnect, including 4 routers (more routers can be added at a very low cost [14], e.g. to match the floorplan), 6 NIs, 35 protocol shells and 13 local buses occupies only 0.3 mm^2, excluding the NI buffers. The latter, when implemented as flip-flop gray-code FIFOs occupy 1 mm^2 (or roughly five times less with custom FIFOs). The interconnect is thus in the order of 4% (or less than 3%) of the entire chip area.

The proposed interconnect offers high throughput (several Gbps for all the components) at a very low cost. The latency, however, is only a couple of cycles for the individual components, but due to the distributed nature of the interconnect, the best-case round-trip for a read operation from the ARM to the SRAM in Figure 2 (running at 200 MHz and the network at 500 MHz), is in the order of 30 processor cycles. In other words, the logical and physical modularity comes at the price of increased latencies.

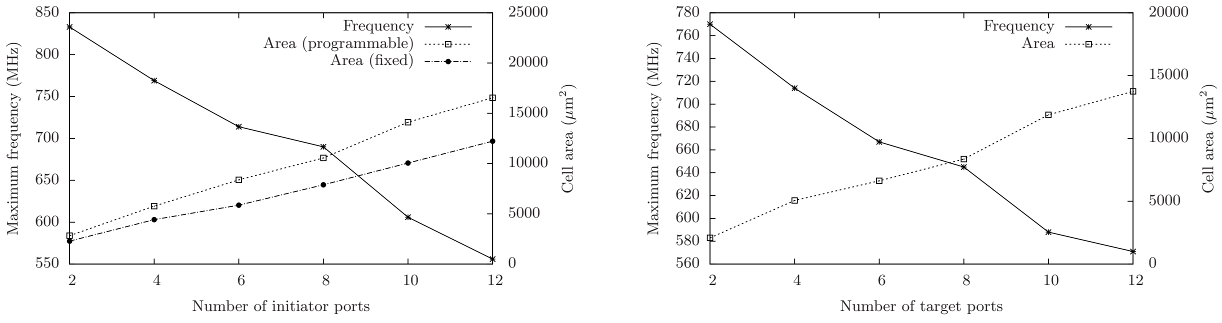
7. CONCLUSIONS

Systems on Chip (SoC) integrate applications with diverse requirements and intellectual property developed by unrelated design teams. This leads to multiple communication paradigms and programming models that the on-chip interconnect must accommodate.

In this work we propose an interconnect and protocol stack that enables multiple communication paradigms and programming models, with the mechanisms required to implement release consistency. We clearly separate the network stack, the streaming stack and the memory-mapped stack, both logically and physically by combining the network with local buses. The proposed interconnect allows any higher-level programming model without introducing message-dependent deadlock by breaking all connection inter-dependencies inside the network.

We quantify the cost and performance of the interconnect building blocks with synthesis results. For a complete SoC instance, the interconnect occupies only 4% of the chip area.

[1] The outstanding transactions are only for the shared target. There could be many more ongoing transactions in the interconnect.



(a)

(b)

Figure 8: Area and frequency for target bus (a) and initiator bus (b).

8. REFERENCES

[1] ARM Limited. *AHB-Lite Product Information*, 2001.

[2] ARM Limited. *AMBA AXI Protocol Specification*, 2003.

[3] Arteris. A comparison of network-on-chip and busses. White paper, 2005.

[4] E. Beigne *et al.* An asynchronous NOC architecture providing low latency service and its multi-level design framework. In *Proc. ASYNC*, 2005.

[5] L. Benini and G. de Micheli. Powering Networks on Chips. In *Proc. ISSS*, 2001.

[6] T. Bjerregaard and S. Mahadevan. A survey of research and practices of Network-on-Chip. *ACM Comp. Surveys*, 38(1), 2006.

[7] T. Bjerregaard *et al.* An OCP compliant network adapter for GALS-based SoC design using the MANGO network-on-chip. In *Proc. SOC*, 2005.

[8] C. E. Cummings. Simulation and synthesis techniques for asynchronous fifo design. *Synopsys Users Group*, 2002.

[9] J. Day and H. Zimmermann. The OSI reference model. *Proc. of the IEEE*, 71(12), 1983.

[10] K. Gharachorloo *et al,*. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proc. ISCA*, 1990.

[11] K. Goossens *et al.* The Æthereal network on chip: Concepts, architectures, and implementations. *IEEE Des. and Test of Comp.*, 22(5), 2005.

[12] A. Hansson *et al.* Avoiding message-dependent deadlock in network-based systems on chip. *VLSI Design*, 2007, 2007.

[13] A. Hansson *et al.* Undisrupted quality-of-service during reconfiguration of multiple applications in networks on chip. In *Proc. DATE*, 2007.

[14] A. Hansson *et al.* aelite: A flit-synchronous network on chip with composable and predictable services. In *Proc. DATE*, 2009.

[15] A. Hansson *et al.* Enabling application-level performance guarantees in network-based Systems on Chip by applying dataflow analysis. *IET Comp. and Design Techn.*, 2009.

[16] ITRS. International technology roadmap for semiconductors, 2007. Design.

[17] V. Lahtinen *et al.* Bus structures in Network-on-Chips. In *Interconnect-centric design for advanced SoC and NoC*. Springer, 2006.

[18] P. Magarshack and P. G. Paulin. System-on-Chip beyond the nanometer wall. In *Proc. DAC*, 2003.

[19] P. Martin. Design of a virtual component neutral network-on-chip transaction layer. In *Proc. DATE*, 2005.

[20] M. Millberg *et al.* The Nostrum backbone - a communication protocol stack for networks on chip. In *Proc. VLSID*, 2004.

[21] I. Miro Panades *et al.* A low cost network-on-chip with guaranteed service well suited to the gals approach. In *Proc. NANONET*, 2006.

[22] L. Nachtergaele *et al.* Optimization of memory organization and hierarchy for decreased size and power in video and image processing systems. In *Proc. MTDT*, 1995.

[23] L. Nachtergaele *et al.* System-level power optimization of video codecs on embedded cores: A systematic approach. *Jour. of VLSI Signal Processing*, 18(12), 1998.

[24] A. Nieuwland *et al.* C-HEAP: A heterogeneous multi-processor architecture template and scalable and flexible protocol for the design of embedded signal processing systems. *Design Automation for Embedded Systems*, 7(3), 2002.

[25] OCP International Partnership. *OCP Specification 2.2*, 2007.

[26] L. Ost *et al.* MAIA: a framework for networks on chip generation and verification. In *Proc. ASP-DAC*, 2005.

[27] M. Palesi *et al.* Application specific routing algorithms for networks on chip. *IEEE Trans. on Par. and Dist. Syst.*, 20(3), 2009.

[28] C. Paukovits and H. Kopetz. Concepts of switching in the time-triggered network-on-chip. In *Proc. RTCSA*, 2008.

[29] P. G. Paulin *et al.* Parallel programming models for a multi-processor SoC platform applied to high-speed traffic management. In *Proc. CODES+ISSS*, 2004.

[30] F. Pétrot and A. Greiner. Cache coherency and memory consistency in NoC based shared memory multiprocessor SoC architectures. In *Proc. DSD*, 2006.

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

[32] C. Rowen and S. Leibson. *Engineering the Complex SOC: Fast, Flexible Design with Configurable Processors*. Prentice Hall PTR, 2004.

[33] A. Rădulescu and K. Goossens. Communication services for network on silicon. In *Domain-Specific Processors: Systems, Architectures, Modeling, and Simulation*. Marcel Dekker, 2004.

[34] A. Rădulescu *et al.* An efficient on-chip network interface offering guaranteed services, shared-memory abstraction, and flexible network programming. *IEEE Trans. on CAD of Int. Circ. and Syst.*, 24(1), 2005.

[35] M. Rutten *et al.* Dynamic reconfiguration of streaming graphs on a heterogeneous multiprocessor architecture. *IS&T/SPIE Electron. Imag.*, 5683, 2005.

[36] H. Sasaki. Multimedia complex on a chip. *Proc. ISSCC*, 1996.

[37] Y. H. Song and T. M. Pinkston. On message-dependent deadlocks in multiprocessor/multicomputer systems. In *Proc. HiPC*, 2000.

[38] Sonics, Inc. *SonicsMX Datasheet*, 2005.

[39] D. Soudris *et al.* Data-reuse and parallel embedded architectures for low-power, real-time multimedia applications. In *Proc. PATMOS*, 2000.

[40] F. Steenhof *et al.* Networks on chips for high-end consumer-electronics TV system architectures. In *Proc. DATE*, 2006.

[41] S. Stergiou *et al.* ×pipes lite: A synthesis oriented design library for networks on chips. In *Proc. DATE*, 2005.

[42] B. Vermeulen *et al.* Debugging distributed-shared-memory communication at multiple granularities in networks on chip. In *Proc. NOCS*, 2008.

[43] P. Wielage *et al.* Design and DfT of a high-speed area-efficient embedded asynchronous FIFO. In *Proc. DATE*, 2007.

[44] D. Wingard. Socket-based design using decoupled interconnects. In *Interconnect-Centric design for SoC and NoC*. Kluwer, 2004.

[45] Xilinx, Inc. *Processor Local Bus (PLB) v3.4*, 2003.