Chapter 2
Proposed Solution

In this chapter we give a high-level view of the building blocks of the interconnect and discuss the rationale behind their partitioning and functionalities. We start by introducing the blocks by exemplifying their use (Section 2.1). This is followed by a discussion of how the interconnect delivers scalability at the physical and architectural level (Section 2.2). Next, we introduce the protocol stack and show how it enables diversity, both in the application programming models and in the IP interfaces (Section 2.3). We continue by explaining how we provide temporal composability when applications share resources (Section 2.4). Thereafter, we detail how our interconnect enables predictability on the application level by providing dataflow models of individual connections (Section 2.5). Next we show how we implement reconfigurability to enable applications to be independently started and stopped at run time (Section 2.6). Lastly, we describe the rationale behind the central role of automation in our proposed interconnect (Section 2.7), and end this chapter with conclusions (Section 2.8).

2.1 Architecture Overview

The blocks of the interconnect are all shown in Fig. 2.1, which illustrates the same system as Fig. 1.4, but now with an expanded view of the interconnect, dimensioned for the applications in Fig. 1.5 and use-cases in Fig. 1.6.

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 data port of the processor, i.e. the *memory-mapped initiator* port. Since the ARM uses distributed memory, a *target bus* with a *reconfigurable address decoder* forwards the read *request message* to the appropriate initiator port of the bus, *based on the address*. The *elements* that constitute the request message, e.g. the address and command flags in the case of a read request, are then serialised by a *target shell* into individual words of streaming data, as elaborated on in Section 2.3. The streaming data is fed via a *Clock Domain Crossing* (CDC) into the NI *input queue* corresponding to a specific *connection*. The data items reside in that input queue until the *reconfigurable NI arbiter* schedules

A. Hansson, K. Goossens, *On-Chip Interconnect with Aelite*, Embedded Systems,
DOI 10.1007/978-1-4419-6865-4_2, © Springer Science+Business Media, LLC 2011

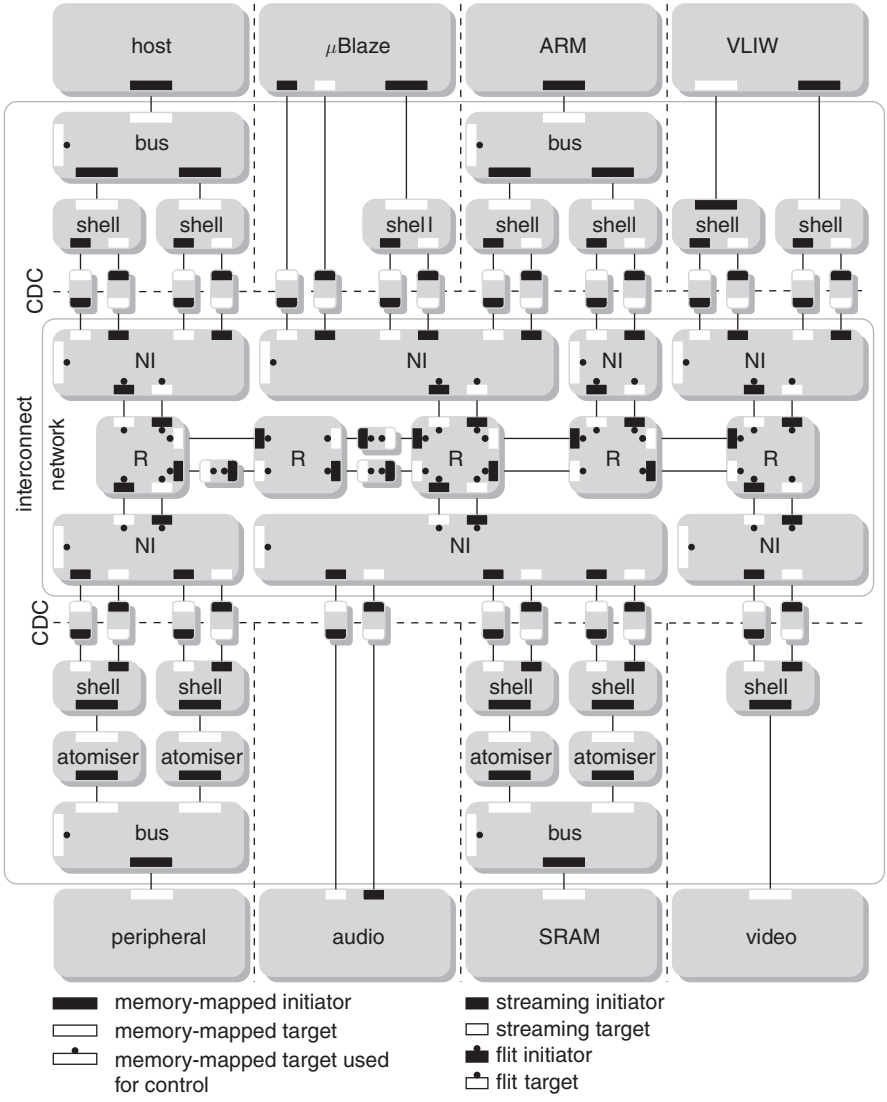

Fig. 2.1 Interconnect architecture overview

the connection. The streaming data is *packetised* and injected into the router network, as *flow-control digits (flits)*. Based on a *path* in the *packet header*, the flits are forwarded through the router network, possibly also encountering *pipeline stages* on the links, until they reach their destination NI. In the network, the flits are forwarded *without arbitration*, as discussed further in Section 2.1.1. Once the flits reach the destination NI, their payload, i.e. the streaming data, is put in the NI *output queue* of the connection and passes through a clock domain crossing into an *initiator shell*.

The 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 Fig. 2.1. For a shared target, the request message passes through an *atomiser*. By splitting the request, the atomiser ensures that transactions, from the perspective of the shared target, are of a *fixed size*. As further discussed in Section 2.4, the atomiser also provides buffering to ensure that atomised transaction can complete in their entirety *without blocking*. Each atomised request message is then forwarded to an *initiator bus* that *arbitrates* between different initiator ports. Once granted, the request message is forwarded to the target, in this case the SRAM, and a *response message* is generated. The elements that constitute the response message are sent back through the bus. Each atomised response message is buffered in the atomiser that reconstructs the entire response message. After the atomiser, the response message is presented to the initiator shell that issued the corresponding request. The shell adds a message header and serialises the response message into streaming data that is sent back through the *network*, hereafter denoting the NIs, routers and link pipeline stages. On the other side of the network, the response message is reassembled by the target shell and forwarded to the bus. The target bus implements the transaction *ordering* corresponding to the IP port protocol. Depending on the protocol, the response message may have to wait until transactions issued by the ARM before this one finish. The bus then forwards the response message to the ARM, completing the read transaction and the load instruction. In addition to response ordering, the target bus also implements mechanisms such as tagging [49] to enable programmers to choose a specific *memory-consistency model*.

In the proposed interconnect architecture, the applications share the network and possibly also the memory-mapped targets.[1] Arbitration thus takes place in the NIs and the initiator buses. Inside the network, however, *contention-free routing* [164] removes the need for any additional arbitration. We now discuss the concepts of contention-free routing in more detail.

2.1.1 Contention-Free Routing

Arbitration in the network is done at the level of flits. The injection of flits is regulated by TDM *slot tables* in the NIs such that no two flits ever arrive at the same link at the same time. Network *contention and congestion* is thus avoided. This is illustrated in Fig. 2.2, where a small part of the network from our example system is shown. In the figure, we see three *channels*, denoted c_0, c_1 and c_2, respectively. These three channels correspond to the connections from the streaming port on the μBlaze to the DAC, the request channel from the memory-mapped port on the μBlaze to the SRAM, and the request channel from the ARM to the SRAM.

[1] Memory-mapped initiator ports and target buses are not shared in the current instance, and is something we consider future work.

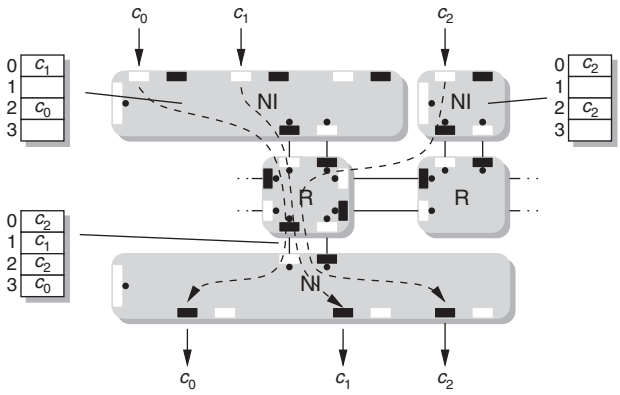

Fig. 2.2 Contention-free routing

Channels c_0 and c_1 have the same source NI and have slots 2 and 0 reserved, respectively. Channel c_2 originates from a different NI and also has slots 0 and 2 reserved. The TDM table size is the same throughout the network, in this case 4, and every slot corresponds to a flit of fixed size, assumed to be three words throughout this work.

For every hop along the path, the reservation is shifted by one slot, also denoted a *flit cycle*. For example, in Fig. 2.2, on the last link before the destination NI, c_0 uses slot 3 (one hop) c_1 slot 1 (one hop) and c_2 slots 0 and 2 (two hops). The notion of a flit cycle is a result of the alignment between the *scheduling interval* of the NI, the flit size and the forwarding delay of a router and link pipeline stage. As we shall see in Chapter 4, this alignment is crucial for the resource allocation.

With contention-free routing, the router network behaves as a non-blocking pipelined multi-stage switch, with a global schedule implied by all the slot tables. Arbitration takes place once, at the source NI, and not at the routers. As a result each channel acts like an independent FIFO, thus offering composability. Moreover, contention-free routing enables predictability by giving per-channel bounds on latency and throughput. We return to discuss composability and predictability in Sections 2.4 and 2.5, respectively. We now discuss the consequences of the scalability requirement.

2.2 Scalability

Scalability is required both at the physical and architectural level, i.e. the interconnect must enable both large die sizes and a large number of IPs. We divide the discussion of our proposed interconnect accordingly, and start by looking at the physical level.

2.2.1 Physical Scalability

To enable the IPs to run on independent clocks, i.e. GALS at the level of IPs, the NIs interface with the shells (IPs) through clock domain crossings. We choose to implement the clock domain crossings using bi-synchronous FIFOs. This offers a clearly defined, standardised interface and a simple protocol between synchronous modules, as suggested in [103, 116]. Furthermore, a clock domain crossing based on bi-synchronous FIFOs is robust with regards to metastability, and allows each locally synchronous module's frequency and voltage to be set independently [103]. The rationale behind the placement of the clock domain crossings between the NIs and shells is that all complications involved in bridging between clock domains are confined to a single component, namely the bi-synchronous FIFO. Even though the IPs have their own clock domains (and possibly voltage islands), normal test approaches are applicable as the test equipment can access independent scan chains per synchronous block of the system [103], potentially reusing the functional interconnect as a test access mechanism [20].

In addition to individual clock domains of the IPs, the link pipeline stages enable *mesochronous clocking* inside the network [78]. The entire network thus uses the same clock (frequency), but a phase difference is allowed between neighbouring routers and NIs. Thus, in contrast to a synchronous network, restrictions on the phase differences are relaxed, easing the module placement and the clock distribution. As the constraints on the maximum phase difference are only between neighbours, the clock distribution scales with the network size, as demonstrated in [83]. The rationale behind choosing a mesochronous rather than an asynchronous interconnect implementation is that it can be conceived as globally synchronous on the outside [29]. Thereby, the system designer does not need to consider its mesochronous nature. Furthermore, besides the clock domain crossings and link pipeline stages, all other (sub-)components of the interconnect are synchronous and consequently designed, implemented and tested independently.

2.2.2 Architectural Scalability

The physical scalability of the interconnect is necessary but not sufficient. The distributed nature of the interconnect enables architectural scalability by *avoiding central bottlenecks* [19, 43]. More applications and IPs are easily added by expanding the interconnect with more links, routers, NIs, shells and buses. Furthermore, the latency and throughput that can be offered to different applications is a direct result of the amount of contention in the interconnect. In the network, the *contention can be made arbitrarily low*, and links (router ports), NIs and routers can be added up to the level where each connection has its own resources, as exemplified in Fig. 2.3. In Fig. 2.3a, we start with two connections sharing an NI. We then add two links (router ports) in Fig. 2.3b (to an already existing router), thus reducing the level of contention. Figure 2.3c continues by giving each connection a dedicated NI. Lastly,

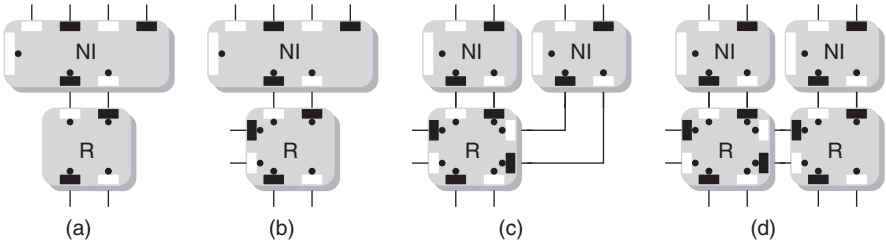

Fig. 2.3 Architecture scaling by adding links (**b**), NIs (**c**), and routers (**d**)

Fig. 2.3d distributes the connections across multiple routers, thus reducing the contention to a minimum (but increasing the cost of the interconnect). As we shall see in Section 2.4 the proposed interconnect also contributes to architectural scalability by having a router network that does not grow with the number of connections.

Outside the network, the contention for a shared target port, such as an off-chip memory controller, cannot be mitigated by the interconnect as it is inherent in the applications. This problem has to be addressed at the level of the programming model, which we discuss in depth in the following section. We return to discuss scalability in Chapter 8 where we demonstrate the *functional scalability* using two large-scale case studies, and also see concrete examples of the effects of inherent sharing.

2.3 Diversity

Diversity in interfaces and programming models is addressed by the protocol stack shown in Fig. 2.4. The stack is divided into five layers according to the seven-layer Open Systems Interconnection (OSI) reference model [45]. As seen in the figure, the memory-mapped protocol, the streaming protocol and the network protocol each have their own stack, bridged by the shell and the NI. We discuss the three stacks in turn.

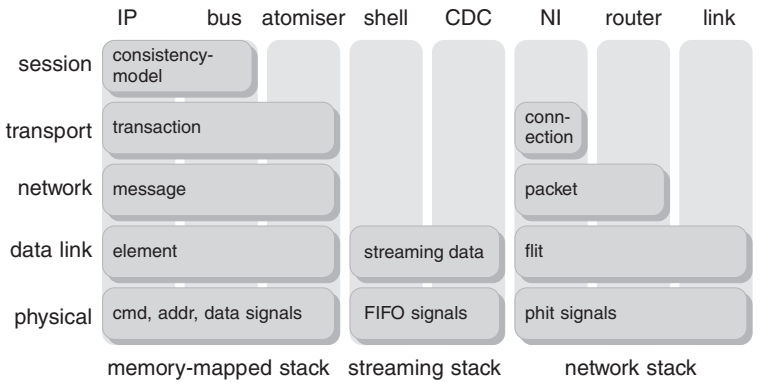

Fig. 2.4 Interconnect protocol stack

2.3.1 Network Stack

The network stack is similar to what is proposed in [18, 25, 118, 123]. The NI is on the transport level as it maintains end-to-end (from the perspective of the network stack) *connections* and guarantees ordering within, but not between connections [167]. A connection is a bi-directional point-to-point inter-connection, between two pairs of initiator and target streaming ports on the NIs. Two unidirectional *channels*, one in each direction, connect the two pairs of ports. Due to the bi-directional nature of a connection, streaming ports always appear in pairs, with one initiator and one target port. As we shall see in Section 2.4, the way in which connections are buffered and arbitrated is central to the ability to provide composable and predictable services in the network, and has a major impact on the NI and router architecture. As seen in Fig. 2.4, the router is at the network level and performs switching of *packets*. The last element of the network, the link pipeline stage, is at the data link level and is responsible for the (synchronous or mesochronous) clock synchronisation and flow control involved in the transport of *flits*. The physical layer, i.e. timing, bus width and pulse shape, is governed by the *phit* (physical digit) format.

From the perspective of the IPs, the network behaves as a collection of distributed and independent FIFOs (or virtual wires), with data entering at a streaming target port, and at a later point appearing at a corresponding streaming initiator port (determined by the allocation). Thus, the network stack is completely hidden from the IPs, and only used by means of the streaming stack.

2.3.2 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 [183]) all make direct use of this stack. The data-link level 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 level 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. We return to discuss the blocking flow control and its implications in Section 2.4.

As illustrated in Fig. 2.5, the NI bridges between the streaming stack and network stack by establishing connections between streaming ports and embedding streaming data in network packets. The connections are thus, via the streaming stack, offering bi-directional point-to-point streaming communication, *without any interpretation or assumptions* on the time or value of the individual words of streaming data. Moreover, with FIFO ordering inside a network channel, it is up to the users of the network, e.g. IPs and buses to implement a specific inter-channel ordering.

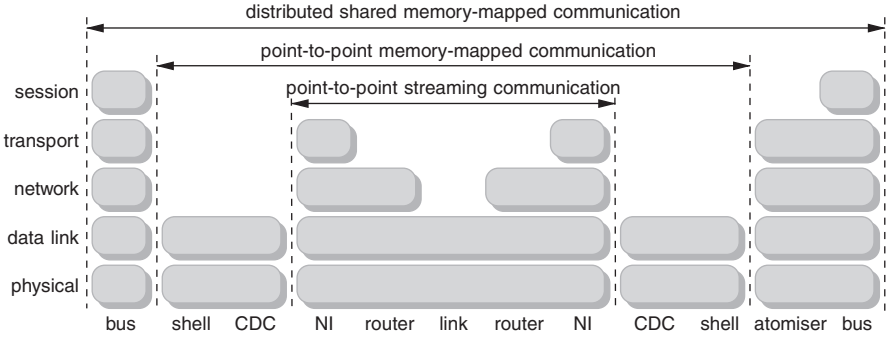

Fig. 2.5 Bridging the protocol stacks

Both the aforementioned properties are key in enabling diversity in communication paradigms and programming models, as described below.

The most basic use of the streaming stack is exemplified in Fig. 2.1 by the μBlaze that communicates with the audio directly via the streaming ports of the NI. One connection is sufficient to interconnect the two blocks, with one channel carrying data from the ADC to the μBlaze and the other channel from the μBlaze to the DAC. Only raw data, i.e. signed pulse-code-modulated samples, is communicated across both channels that are completely homogeneous [77].

2.3.3 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 interfaces with dedicated groups of wires for command, address, write data and read data [2, 4, 8, 49, 147, 160]. 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 *shell* that (independently) serialises the memory-mapped request and response messages. As illustrated in Fig. 2.5, the shells enable *point-to-point memory-mapped communication* by bridging between the *elements* of a bus-protocol message, 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 bus stack and the streaming stack. As already mentioned, the network places no assumptions on the time and value of individual words of streaming data, and also not on the size, syntax, semantics or synchronisation granularity of messages. Consequently, the shells enable streaming communication and memory-mapped communication to co-exist in the interconnect. Moreover, different pairs of memory-mapped initiators and targets may communicate using different protocols, thus enabling multiple memory-mapped protocols.

The shells work on the granularity of a single memory-mapped initiator or target port. However, a memory-mapped initiator port often uses distributed

memory [168], and accesses multiple targets, based on e.g. the address, the type of transaction (e.g. read or write) or dedicated identifier signals in the port interface [147]. The use of distributed memory is demonstrated by the host and ARM in Fig. 2.1. As the ARM accesses more than one target port, the outgoing requests have to 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 Fig. 2.1. A shared target must hence be *arbitrated*, and the initiators' transactions multiplexed according to the protocol of the target port. Next, we show how distributed and shared memory-mapped communication, as shown in Fig. 2.5, is enabled by the target buses and initiator buses, respectively.

Despite all the multiplexing and arbitration *inside the network* we choose to add buses *outside the network*. The primary reason why the network is not responsible for all multiplexing and arbitration is *the separation of bus and network stacks*. The network stack offers logical (bi-directional) point-to-point connections, without any ordering between connections. Ordering of, and hence synchronisation between, transactions destined for different memories depends on particular IP port protocols and is therefore addressed outside the network, in the target buses. The ordering and synchronisation between transactions place the buses at the session level in the stack, and adds a fifth layer to the interconnect protocol stack, as seen in Fig. 2.4. At the network level in the bus stack we have *messages* in the form of requests and responses. It is the responsibility of the bus to perform the necessary multiplexing and direct messages to the appropriate 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 different *signal groups* of the bus interface.

The division of the protocol stack clearly separates the network from any session-level issues and pushes those responsibilities to the buses. The network is thereby independent of the IP protocols and relies on target buses for distributed memory communication, and initiator buses for shared memory communication. The division enables the interconnect to support different types of bus protocols and different memory consistency models. The choice of a consistency model, e.g. *release consistency* [59], is left for the IP (and bus) developer. That is, the interconnect provides the necessary mechanisms, and it is left for the IPs to implement a specific policy. There are no restrictions on the consistency model, but to exploit the parallelism in the interconnect, it is important that the ordering is as relaxed as possible. The importance of parallelism, e.g. through transaction pipelining, is emphasised by the large (10–100 cycles best-case) latency involved in receiving response from a remotely located target.[2]

Note that the stack separation does not exclude the option to implement the protocol-related ordering and arbitration in the protocol shell, i.e. merge the bus

[2] If the ARM and SRAM in Fig. 2.1 run at 200 MHz and the network at 500 MHz, the best-case round-trip for a read operation is in the order of 30 processor cycles.

and shells as suggested in [168]. An integrated implementation may result in a more efficient design [91]. Indeed, placing both the shell and bus functionality inside such a block enables a design where the lower layers involved in the protocol translation functionality are shared by the ports, thus potentially achieving a lower hardware cost and latency. The main drawback with such a monolithic approach is the complexity involved in handling multiple concurrent transactions, while complying with both the bus protocol (on the session level) and streaming protocol. Our proposed division of the stack enables us to bridge between protocols on the lower layers, something that involves far fewer challenges than attempting to do so on the session level [118]. Furthermore, by having buses that complies with the IP protocol it is possible to reuse available library components. It is also possible to verify the buses independent of the network with established protocol-checking tools such as Specman [34], and to use any existing functional blocks for, e.g. word-width and endianness conversion, or instrumentation and debugging [192]. Lastly, thanks to the clear interface between shell and NI, the shells belonging to one IP port, e.g. the two shells of the ARM in Fig. 2.1, can easily be distributed over multiple NIs, e.g. to increase the throughput or provide lower latency.

2.4 Composability

As discussed in Chapter 1, we refer to a system where applications do not influence each other temporally as composable. Application composability is easily achieved by using dedicated resources for the different applications. This, however, is often too costly (in terms of power and area) or even impossible. Consider, for example, the case of an off-chip memory where pin constraints limit the number of memories. To fulfil the requirement on application composability, it is thus necessary to offer composable resource sharing by removing all interferences between applications. Four properties are central to the ability of providing composable sharing:

- the *flow-control scheme* used by the shared resource,
- the respective *granularities of flow control and arbitration*,
- the *size of an arbitration unit*,
- the *temporal interference* between arbitration units.

We now look at these properties in turn, using the abstract shared resource in Fig. 2.6 to illustrate the concepts. This resource corresponds to, e.g., the router network or the SRAM in Fig. 2.1.

2.4.1 Resource Flow-Control Scheme

Flow control is the process of adjusting the flow of data from a sender to a receiver to ensure that the receiver can handle all of the incoming data. The most basic flow-control mechanism is dropping. If the resource is not ready (e.g. the buffer is full or

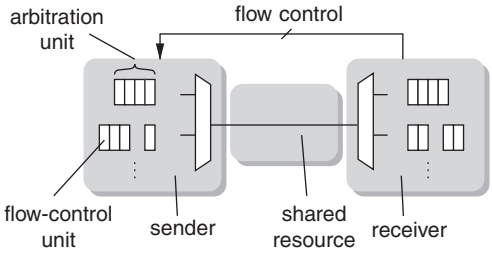

Fig. 2.6 Resource sharing

the receiver is busy), then data is simply dropped. However, dropping data and thus making the flow-control lossy complicates the use of the resource in a larger context and requires expensive recovery mechanisms to provide lossless and ordered service on higher levels [164]. Moreover, as discussed in Section 2.5, lossy flow control conflicts with predictability, as the number of retransmissions must be bounded.

Rather than dropping data, we choose to use a robust flow-control scheme where the producer waits for the availability of the resource, thus avoiding retransmissions. The most basic implementation of lossless flow-control uses (synchronous) handshake signals (e.g. valid and accept) between the sender and receiver. This solution requires no extra book keeping, but restricts the throughput (when the handshake is pipelined) or clock speed (when not pipelined) [150, 158]. An alternative flow-control mechanism uses credits to conservatively track the availability of space on the receiving side. Credit-based flow control can be pipelined efficiently and does not limit the throughput. It does, however, introduce additional counters (for storing credits) and wires (for transmitting credits). As we shall see, our interconnect uses both handshake-based and credit-based flow control at appropriate points.

With the introduction of flow control (handshake or credit based), we can transmit data between the sender and receiver in a lossless fashion. In case the receiver is busy, the sender blocks. For a resource that is not shared, the blocking is not a problem (from the perspective of composability), as only the unique application is affected. However, blocking is problematic for composable sharing as we must ensure that applications do not interfere temporally. That leads us to the next issue, namely the respective granularities of flow control and arbitration.

2.4.2 Flow Control and Arbitration Granularities

The second property that is important for composable resource sharing are the respective flow control and arbitration granularities. To illustrate the problem, consider the initiator bus where the arbiter schedules bus transactions, i.e. complete read and write transactions. Thus, arbitration decisions are taken at the level of transactions. Flow control, however, is done at the finer element level [8, 49, 160]. It is thus possible that a decision is taken and subsequently blocks while in progress, due to lack of either data or space. This is illustrated in Fig. 2.7, which shows the arbitration

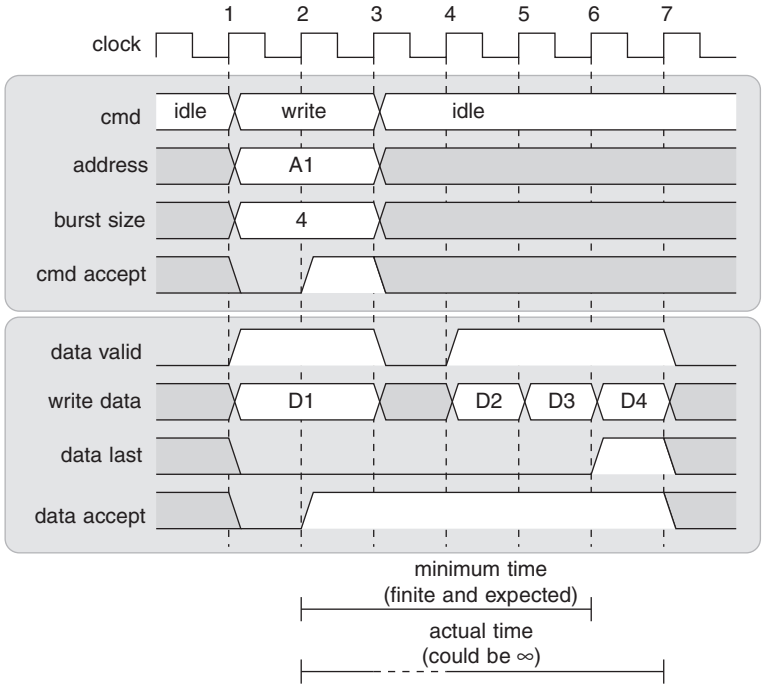

Fig. 2.7 Timing diagram of memory-mapped write transaction

unit of size 4 in Fig. 2.6 in greater detail. Using e.g. DTL [49], a memory-mapped write command (of size 4) is presented in cycle 1, and accepted in cycle 2 together with the first data element. The second data element, however, is not yet available in cycle 3, e.g. due to a slow initiator. Thus, the resource is stalled (and blocked) *for everyone*, waiting for the write data element that is only presented in cycle 4. A similar situation arises for a read transaction when the read data elements are not presented or accepted in consecutive cycles. While the handshakes on the element level ensure lossless transfer of the write data, they also lead to an unknown time before a new arbitration unit can be scheduled, because it depends on the behaviour of the users (IPs and applications) rather than the behaviour of the resource. Hence, due to the discrepancy in levels between taking and executing decisions, the user behaviour affects the resource sharing.

To avoid blocking on the level of flow-control units, we incorporate the availability of data at the source and space at the destination as preconditions for the arbiter in both the NI and the initiator bus. Thus, the arbiter must *conservatively* know how much data is available at the source and how much space is available at the destination and ensure that sufficient data and space are available prior to the scheduling of a transaction or packet. This way, we implement non-blocking flow control on *complete arbitration units*, similar to what is proposed in [132]. As a result, buffering per application is needed both before and after the shared

resource. In our case the buffering is provided by the sending (data) and receiving (space) NI for the network, and by the atomiser (data and space) for a shared target.

An alternative to raising the level of flow control to complete arbitration units, as we do in the proposed interconnect, is to change the level of arbitration, and implement *pre-emption* of transactions and packets, respectively. Doing so is nothing more than replacing the shared resource in Fig. 2.6 with one or more instances of the whole figure, as illustrated in Fig. 2.8. In other words, the problem is pushed one level down, and it becomes necessary to offer parallel states, with independent flow control and buffering, for the elements and flits (rather than transactions and packets) of all applications. This is, for example, the approach taken in multi-threaded OCP [147], and in networks based on *virtual circuits* [12, 27, 154, 162, 165] where every router is a resource as depicted in Fig. 2.8.

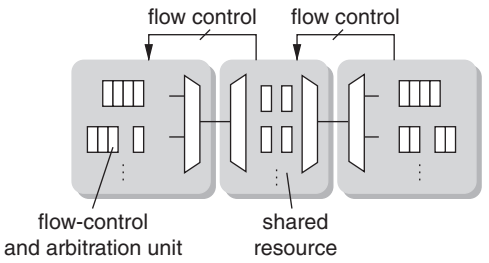

Fig. 2.8 Resource sharing by pre-emption

The reason we choose to raise the level of flow control rather than lowering the level of arbitration for shared targets is that many IPs simply do not support it. Only the OCP protocol offers such support,[3] and even for IPs using this particular protocol it is an optional profile implemented by few IP providers. For the network, the decision is based on design complexity and implementation cost. In contrast to network with virtual circuits, our routers and link pipeline stages are *stateless* [186]. As a result, they are not negatively affected by the number of resource users, in this case the number of applications (translating to a number of virtual circuits, i.e. parallel states and buffers) or the real-time requirements of the applications (the depth of the buffers). Instead, the entire router network (with pipeline stages) in our proposed interconnect can be seen as a single non-blocking shared resource, as shown in Fig. 2.6. The price we pay for the raised level of arbitration is that the NIs require end-to-end flow control, as discussed later in Chapter 3. Moreover, the NI has to know the size of the arbitration units, i.e. the packets. The arbitration unit size is indeed the third important point in enabling composable resource sharing.

[3] AXI currently does not allow multi-threading in combination with blocking flow control.

2.4.3 Arbitration Unit Size

The third point arises due to the choice of raising the flow-control granularity rather than lowering the arbitration granularity (i.e. pre-empting). As a consequence of this decision, the size of an arbitration unit must be known. Moreover, when taking a scheduling decision, the whole arbitration unit must be present in the sending buffer (to avoid placing assumptions on the incoming data rate, i.e. the module inserting data in the buffer). Thus, the maximum arbitration unit size must be known, and the sending buffer sized at least to this size. The same holds for the receiving buffer (with similar arguments as for the sending buffer). In fact, due to the response time of the shared resource and the time required to return flow control, the receiving buffer needs to hold more than one maximum sized arbitration unit to not negatively affect the throughput. We continue by looking at how the problem of variable arbitration unit size is addressed in the network and for the shared targets, respectively.

In the network, we solve the problem with variable packet sizes by letting the arbiter in the sending NI *dynamically adapt the size* of packets to fit with the number of flow-control credits that are currently available. That is, if space is only available for a packet of two flits, then that size is used for the current packet, even if more data is available in the input (sending) buffer. As a consequence, a packet is never larger than the receiving buffer (since the amount of flow-control credits never exceed its size). The buffers can hence be sized in any way (for example to satisfy a specific temporal behaviour as shown in Chapter 6), and the packet sizes adapt dynamically.

For a shared target the situation is more complicated as the size of the arbitration units is decided by the initiators that present the transactions to the initiator bus. Making worst-case assumptions about the maximum arbitration unit is costly in terms of buffering if that size is seldom (or never) used. Moreover, some memory-mapped protocols, most notably AHB [2] and OCP [147], have sequential burst modes where there is no maximum size (referred to as an un-precise burst in OCP). The atomiser, described in detail in Chapter 3, addresses those issues by chopping up transactions into fixed-size sub-transactions that are presented to the arbiter in the initiator bus. The fixed size is also important for the fourth and last item, namely the temporal interference between two arbitration units.

2.4.4 Temporal Interference

The fourth and last part of temporally composable resource sharing is time itself. Composability requires that the time at which an arbitration unit is scheduled and the time at which it finishes does not depend on the presence or absence of other applications. This can be ensured by always *enforcing the maximum temporal interference of other applications*. Note that the response time of the shared resource, i.e. the time it takes to serve a complete arbitration unit, does not matter for composability (although it is important for predictability as discussed later). For example, if we would clock gate the resource (and arbiter) in Fig. 2.6 an arbitrary period of

time, this changes the time at which arbitration units are scheduled (unpredictably), but the interference is unaffected and the resource is still composable. However, *enforcing the worst-case temporal behaviour* (per application), also including the uncertainty of the platform, is a sufficient but not necessary condition for composability.

In the network, the maximum interference between applications is enforced by time multiplexing packet injection according to the slot table and enforcing the maximum size (per packet and flit). Even when a packet does not occupy complete flits (i.e. finishes early), the size specified in the slot table is enforced from the perspective of other applications. The links accept one phit every cycle, and there is no contention in the router network (due to the contention-free routing). Consequently, the time between the scheduling of packets is known and determined by the slot table.

For a shared target port, the time between arbitration decisions depends on the transaction granularity of the atomisers (a write requires at least as many cycles as write data elements) and the behaviour of the specific IP. For the SRAM in Fig. 2.1, for example, the atomisers issue transactions of one element, and a new transaction can be issued every two cycles. Similar to the network, the initiator bus uses TDM-based arbitration to enforce the maximum interference between applications.

2.4.5 Summary

To summarise, from the perspective of the applications, our interconnect implements composable resource sharing by using pre-emptive arbitration and enforcing maximum application interference. Thus, *composability is achieved without enforcing the worst-case temporal behaviour*. The network is pre-emptive at the level of connections, but implemented using non-pre-emptive sharing at the level of packets. Similarly, a shared target is pre-emptive at the level of transactions, but implemented using non-pre-emptive sharing at the level of sub-transactions. For this purpose, a new component, namely the atomiser, is introduced. The rationale for not choosing a lower level of pre-emption is reduced cost and compliance with existing memory-mapped protocols. The rationale for not choosing a higher level of pre-emption, i.e. consider the entire interconnect as a non-blocking shared resource, as proposed in the time-triggered architectures [100, 157], is that this pushes the responsibilities of ensuring availability of data and space onto the IPs. In doing so, the interconnect is no longer suitable for general applications, which conflicts with our requirements on diversity.

2.5 Predictability

There are two important parts to predictability, namely having an architecture built from blocks that deliver bounds on their temporal behaviour [12, 26, 63, 95, 108, 122, 154, 157, 165, 193], and choosing a model in which those behaviours can

analysed together with the behaviours of the applications [24, 182, 185]. We start by presenting the architecture, followed by the rationale behind the selected analysis technique.

2.5.1 Architecture Behaviour

To provide bounds on the end-to-end temporal behaviour of an application, all the resources used by the application, e.g. the network, must be predictable, i.e. offer useful bounds on their individual temporal behaviours. If a resource is also shared between tasks of the same application, an intra-application arbiter is required to facilitate admission control, resource reservation and budget enforcement [120]. The intra-application arbiter thus prevents a misbehaving or ill-characterised task from invalidating another task's bounds. In contrast to the composable sharing discussed in Section 2.4, the work-conserving [205] intra-application arbiter does not enforce worst-case interference and can distribute residual capacity (slack) freely between the individual tasks (or connections) of the application, possibly improving performance.

In situations where an application requires bounds on its temporal behaviour and a (predictable) resource is shared also with other applications, it is possible to separate intra- and inter-application arbitration. As already mentioned, the intra-application arbiter can distribute capacity within an application and thus reduce the discretisation effects on resource requirements or use the scheduling freedom to improve average-case performance [71]. The cost, however, is in the additional level of arbitration and buffering that must be added. Therefore, in our proposed interconnect, we currently use only one level of arbitration, both within and between applications.

2.5.2 Modelling and Analysis

For application-level predictability, the entire application and platform must be captured in a specific MoC. Thus, a model is needed for every block in Fig. 2.1 that is to be used by a real-time application. Moreover, for the shared resources, i.e. the network and the initiator buses, the arbitration mechanism must be modelled. Having temporal bounds on the behaviour of every component is necessary, but not sufficient. For example, in Chapter 4, we allocate network resources to guarantee the satisfaction of *local latency and throughput bounds*, inside the network. However, as we have seen in Section 2.4, the flow control and arbitration is affected by the availability of buffer space. Consequently, it is necessary to also model the effects of buffers and flow control, between every pair of components, also considering potential variations in the granularity of arbitration units (e.g. read and write transactions).

Taking these considerations into account, we choose to capture the temporal behaviour of the interconnect using Cyclo-Static Dataflow (CSDF) as the MoC [24]. The rationale for doing so is that dataflow analysis [182], in contrast to e.g. real-time calculus, enables us to capture the effects of flow control (bounded buffers) and arbitration in a straightforward manner. As we shall see in Chapter 6, run-time arbiters from the broad class of latency-rate servers [185] can be modelled using dataflow graphs [198]. This enables us to construct a conservative model of a network channel, capturing both the interconnect architecture and resource allocations, derived in Chapter 4. Additionally, dataflow graphs cover a wide range of application behaviours, even with variable-production and consumption rates [199], enabling us to capture the platform and the mapping decisions with a good accuracy, i.e. with tight bounds [127]. Dataflow graphs also decouple the modelling technique and analysis method, thereby enabling us to analyse the same dataflow model with both fast approximation algorithms [15] and exhaustive back-tracking [44]. Using dataflow analysis it is possible to compute sufficient buffer capacities given a throughput (or latency) constraint, and to guarantee satisfaction of latency and throughput (and periodicity) constraints with given buffer sizes [10].

2.6 Reconfigurability

We have already seen examples in Fig. 1.6 of how applications are started and stopped at run time, creating many different use-cases. The different use-cases have different communication patterns (connection topology) and throughput and latency requirements (connection behaviour) that the interconnect must accommodate. Moreover, as stated in Chapter 1, starting or stopping one application should not affect the other applications that are running. We now look at the impact those requirements have on the granularity of reconfiguration and the interconnect architecture.

2.6.1 Spatial and Temporal Granularity

Figure 2.9 shows the example system, with the same progression of use-cases as already shown in Fig. 1.6b, but here illustrating the reconfiguration (resource allocation) granularity, both spatial and temporal. Given that the connection topologies and behaviours vary per use-case, a first approach is to allocate resources per use-case [138], as illustrated in Fig. 2.9a. However, applications are disrupted on use-case transitions, i.e. no composable or predictable services can be provided. Even if an application is present in multiple use-cases, e.g. the filter in the figure, resources used to provide the requested service are potentially different. As the label *global reconfiguration* in Fig. 2.9a illustrates, a use-case transition involves closing and subsequently opening all connections (of all applications) for the two use-cases. Not only does this cause a disruption in delivered services, but it leads to

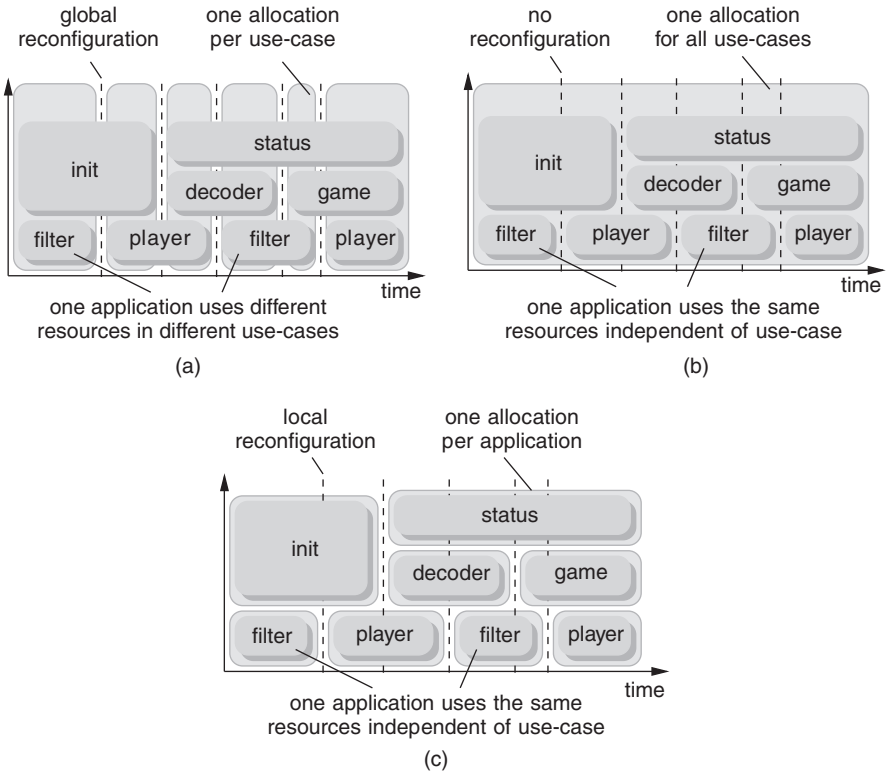

Fig. 2.9 Spatial and temporal allocation granularities

unpredictable reconfiguration times as in-flight transactions of all the running applications must be allowed to finish [102, 145], even if we only want to reconfigure one application.

Undisrupted service for the applications that keep running is achieved by extending the temporal granularity to one allocation for all use-cases [137]. As shown in Fig. 2.9b this removes the need for reconfiguration on use-case transitions. While delivering undisrupted services to the applications, the requirements of the synthetic worst-case use-case are over-specified, with a costly interconnect design as the result [138]. Moreover, if the architecture of the interconnect is given, it may not be possible to find an allocation that meets the worst-case use-case requirements, even though allocations exist for each use-case individually. The worst-case approach is also not applicable to connections that make direct use of the streaming stack. Since a streaming port is only connected to one other port at any given point in time, reconfiguration is required.

The approach we take in this work is to perform resource allocation on the granularity of applications, as later described in Chapter 4. The rationale for doing so is that it introduces spatial (in addition to temporal) granularity thus allowing

local reconfiguration, as shown in Fig. 2.9c. Similar to the worst-case approach, applications are allocated with the same resources independent of the use-case and are unaffected when other applications are started or stopped. However, reconfiguration enables us to share resources (bus ports, NI ports, links and time slots) between mutually exclusive applications (e.g. init, decoder and game in Fig. 2.9c), thus reducing the cost of the interconnect [72].

Although our proposed approach allows the user to specify both a spatial and temporal granularity of reconfiguration, it is not required to use this feature, i.e. it is possible to only distinguish the temporal axis. This is achieved by having one single worst-case application (a fully connected constraint graph), or mutually exclusive applications (no edges in the constraint graph). The methodologies in [137, 138] are thus subsumed in this more general framework.

2.6.2 Architectural Support

There are three modules in the interconnect architecture that are reconfigurable: the target bus, NI and initiator bus. The rationale behind making the target bus and the NI reconfigurable is that they are the two modules where different destinations are decided upon. In the target bus, an initiator port is selected based on the destination address. Similarly, in the (source) NI, a streaming initiator port (in the destination NI) is selected based on the path and port identifier. The NI is also one of the two locations where arbitration is performed. Thus, both the NI and initiator bus are reconfigurable, such that the service given to different connections can be modified at run time.

In Fig. 2.1 we see that the aforementioned modules have a memory-mapped control port. Thus, all reconfiguration is done using memory-mapped communication, reading and writing to the control registers of the individual blocks. The rationale for using memory-mapped communication is the diversity it enables in the implementation of the host. Any processor that has a memory-mapped interface can be used in combination with the run-time libraries, as we shall see in Chapter 5.

2.7 Automation

The proposed design flow, as shown in Fig. 1.8, extends on the flow in [62] and addresses two key problems in SoC design. First, the need for tools to quickly and efficiently generate application-specific interconnect instances for multiple applications. Second, the need for performance verification for a heterogeneous mix of firm, soft and non-real-time applications. We first discuss the rationale behind the input and output of the flow, followed by our reasons for dividing the design flow into multiple independent tools.

2.7.1 Input and Output

As we have already discussed in Chapter 1, this work takes as its starting point the specification of the physical interfaces that are used by the IPs, constraints on how the applications can be combined temporally, and the requirements that each application has on the interconnect. The reason for focusing on the requirements on the interconnect rather than application-level requirements is to allow application diversity by *not placing any assumptions on the applications*. The real-time requirements are described per application, and on a level that is understood by the application designer, e.g. through latency and throughput bounds. Furthermore, the design flow makes no distinction whether the requested throughput and latency reflect the worst-case or average-case use of a particular application. Thus, depending on the applications, the input to our design flow might stem from analytical models, measurements from simulation models, or simply guesstimates based on back-of-the-envelope calculations or previous designs.

Based on the input specification, it is the responsibility of the design flow to automatically generate a complete hardware and software interconnect architecture. There are two important properties of the design-flow output. First, it is possible to turn the output into a physical implementation in the form of an FPGA or ASIC. Second, depending on the application, it is possible to verify application-level performance using a variety of simulation-based and formal techniques. Hence, also the output of the design flow reflects the application diversity.

2.7.2 Division into Tools

The design flow is split into separate tools for several reasons. First, breaking the design flow in smaller steps simplifies steering or overriding heuristics used in each of the individual tools, enhancing user control. It is, for example, possible for the user to construct the entire architecture manually, or modify an automatically generated architecture by, e.g., inserting additional link-pipeline stages on long links. Second, splitting the flow reduces the complexity of the optimisation problem, and simpler, faster heuristics can be used. This results in a low time-complexity at the expense of optimality. As we shall see, pushing decisions to compile time is key as it enables us to guarantee, at compile time, that all application requirements are satisfied, and that all the use-cases fit on the given hardware resources. However, complexity is moved from the platform hardware and software to the design tools, leaving us with a set of challenging problems. As we shall see in Chapters 4 and 6, tools that rely on low-complexity approximation algorithms are central to the interconnect design flow. Higher-level optimisation loops involving multiple tools can then be easily added, as illustrated by the arrows on the left hand side of Fig. 1.8. Third, parts of the flow can be more easily customised, added, or replaced by the user to tailor the flow or improve its performance.

2.8 Conclusions

In this chapter, we present the rationale behind the most important design choices of our interconnect. We introduce the modular building blocks of the hardware and software architecture, and explain how they, together with the design flow, enable us to satisfy the requirements in Chapter 1. In the succeeding chapters, we return to the concepts introduced in this chapter as we explain in-depth how the dimensioning, allocation, instantiation and verification contribute to the requirements.

http://www.springer.com/978-1-4419-6496-0

On-Chip Interconnect with aelite
Composable and Predictable Systems
Hansson, A.; Goossens, K.
2011, X, 210 p., Hardcover
ISBN: 978-1-4419-6496-0