Chapter 6
Virtual-Channel Flow Control and Buffering

In all cases described so far when a packet allocated a link (or an output of a router) the connection was kept until the tail of the packet traversed the link and released its usage to other packets. This behavior was imposed by the fact that the buffers at the other side of the link (or the input of the next router) kept the control information of only one packet, thus prohibiting the interleaving of flits from different packets. This flow of packets resembles a single-lane street where cars move one after the other and even if a car wants to turn to a different direction it is obliged to wait the rest cars to pass the turning point before being able to make the turn to its preferred direction (see Fig. 6.1a). Also, this serial packet movement prohibits packet flow isolation since all traffic is inevitably mixed in the one-lane streets of the network.

Allowing for flow separation and isolation needs the dedication of multiple resources either in space (more physical lanes by adding extra wires on the links) or in time (more virtual resources interleaved on the same physical resources in a well-defined manner). This chapter deals with virtual channels that represent an efficient flow control mechanism for adding lanes to a street network in an efficient and versatile manner, as illustrated in Fig. 6.1b. Adding virtual channels to the network removes the constraints that appear in single-lane streets and allow otherwise blocked packets to continue moving by just turning to an empty (less congested) lane of the same street (Dally and Aoki 1993; Dally 1992). Since the additional lanes are virtually existent their implementation involves the time multiplexing of the packets that belong to different lanes (virtual channels) on the same physical channel. Briefly, virtual channels behave similar to having multiple wormhole channels present in parallel. However, adding extra lanes (virtual channel) to each link does not add bandwidth to the physical channel. It just enables better sharing of the physical channel by different flows (Boura and Das 1997; Nachiondo et al. 2006).

Besides performance improvement, virtual channels are used for a variety of other purposes. Initially, virtual channels were introduced for deadlock avoidance (Dally and Aoki 1993). A cyclic network can be made deadlock-free by

© Springer Science+Business Media New York 2015

G. Dimitrakopoulos et al., *Microarchitecture of Network-on-Chip Routers: A Designer's Perspective*, DOI 10.1007/978-1-4614-4301-8_6

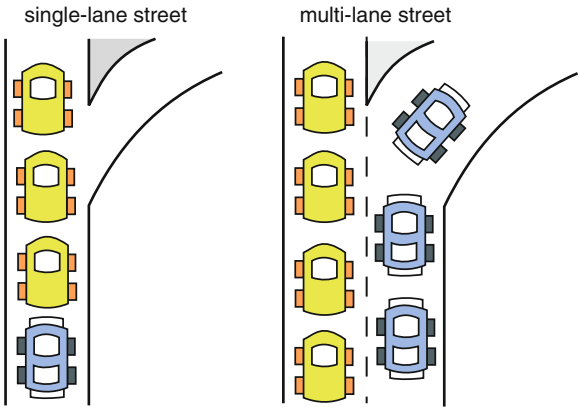

Fig. 6.1 Virtual channels is analogous to adding lanes in a street allowing cars(packets) to flow in parallel without interfering with each other. The added lanes/channels are virtualized since they do not physically exist but appear on the one physical channel in a time-multiplexed manner

restricting routing so that there are no cycles in the channel dependency graph. The channels can be thought as the resources of the network that are assigned to distinct virtual channels. The transition between these resources in the packet's routing path is being restricted in order to enforce a partial order of resource acquisition, which practically removes cyclic dependencies (Duato 1993).

In a similar manner, different types of packets, like requests and reply packets, can be assigned to disjoint sets of virtual channels (VCs) to prevent protocol-level deadlock that may appear at the terminal nodes of the network. For instance, protocol-level restrictions in Chip Multi-Processors (CMP) employing directory-based cache coherence necessitate the use of VCs. Coherence protocols require isolation between the various message classes to avoid protocol-level deadlocks (Martin et al. 2005). For example, the MOESI directory-based cache coherence protocol requires at least three virtual networks to prevent protocol-level deadlocks. A virtual network comprises of one VC (or a group of VCs) that handles a specific message class of the protocol. Virtual networks and the isolation they provide are also used for offering quality of service guarantees in terms of bandwidth allocation and packet delivery deadlines (Grot et al. 2012).

Architectures supporting the use of VCs may reduce also on-chip physical routing congestion, by trading off physical channel width with the number of VCs, thereby creating a more layout-flexible SoC architecture. Instead of connecting two nodes with many parallel links that are rarely used at the same time, one link can be used instead that supports virtual channels, which allows the interleaving in time of the initial parallel traffic, thus saving wires and increasing their utilization.

6.1 The Operation of Virtual-Channel Flow Control

To divide a physical channel into V virtual channels, the input queue at the receiver needs to be separated into as many independent queues as the number of virtual channels. These virtual channels maintain control information that is computed only

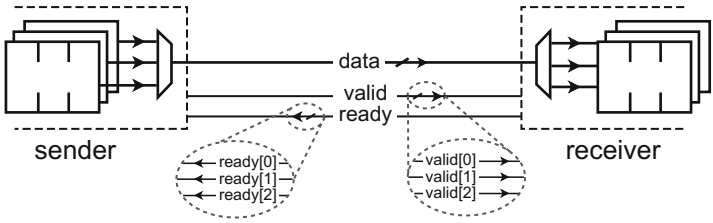

Fig. 6.2 Virtual channels require the addition of separate buffers for each VC at the receiver's side and at the same time call for enhancements to the flow control signaling to accommodate the multiple and independent flows travelling in each VC

once per packet. To support the multiple independent queues link-level flow control is also augmented and includes separate information per virtual channel.

Ready/valid handshake on each network channel cannot distinguish between different flows. This feature prevents the interleaving of packets and the isolation of traffic flows, while it complicates deadlock prevention. A channel that supports VCs consists of a set of data wires that transfer one flit per clock cycle, and as many pairs of control wires valid(i)/ready(i) as the number of VCs. Figure 6.2 shows an example of a 3-VC elastic channel. Although multiple VCs may be active at the sender, flits from only one VC can be sent per clock cycle; only one valid(i) signal is asserted per cycle. The selection of the flit that will traverse the link requires some form or arbitration that will select one VC from those that hold valid flits. At the same time, the receiver may be ready to accept flits that can potentially belong to any VC. Therefore, there is no limitation on how many ready(j) signals can be asserted per cycle.

In VC flow control, both the buffering resources and the flow-control handshake wires have been multiplied with the number of VCs. Therefore, the abstract flow control model developed for the single-lane case in Chap. 2 should be enhanced to support virtual channels. As shown in Fig. 6.3a, we use a separate slot counter for each VC that gets updated by the corresponding buffer and reflects via the ready signal the status of the VC buffer. Normally, only one VC will drain a new flit and thus the status of only one slot counter will be updated. Of course the case of multiple VCs draining flits in parallel can be supported. Keeping the rate of incoming flits equal to the rate of outgoing flits (leaving the receiver's buffer), it is safe to assume that only one update will be asserted in each clock cycle.

Moving the slot counters at the sender side, as shown in Fig. 6.3b transforms the flow control mechanism to the equivalent credit-based flow control for VCs. The ith VC is eligible to send a new flit as long as $creditCount[i] > 0$ meaning that there is at least one empty slot at the downstream buffer for the ith VC. When a new flit leaves the sender it decrements the corresponding credit counter, while the credit updates returned per cycle are indexed by the corresponding credit update wire (update[i]).

Instead of transferring V valid signals and V credit update signals in the forward and in the backward direction, it is preferable to encode the id of the valid VC and

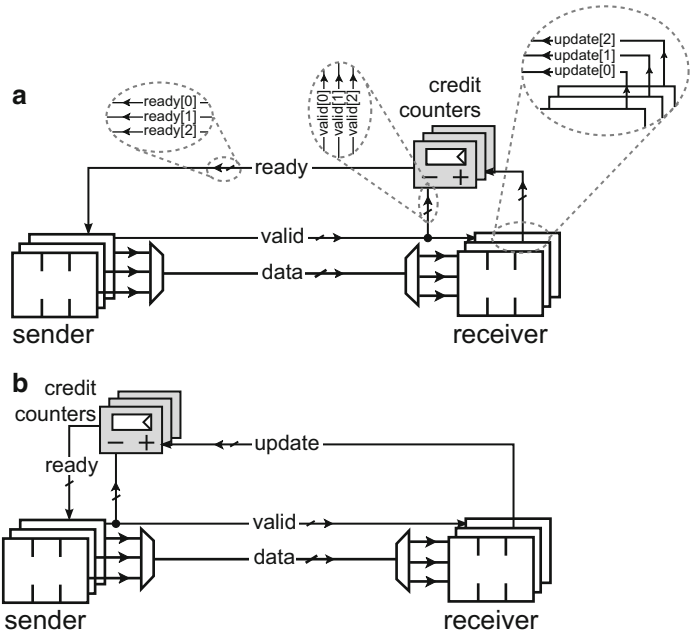

Fig. 6.3 The abstract flow control model enhanced for supporting virtual channels. One free slot (or credit) counter is added per VC that can be placed either at the receiver or at the sender. Multiple ready signals can be asserted per cycle showing which VCs is ready to accept new flits. In the simplest case only one VC will leave the receiver and update the status of the corresponding VC buffer

the id of the VC that returns a new credit; the encoding minimizes the flow control wires to $1+\log_2 V$ in each direction. This is possible since at most one VC will send valid data to the receiver and the receiver will drain at most one flit for its buffers thus updating the status of only one credit counter. Thus, in each direction a single valid/update bit is used and a VCid that encodes the index of the selected VC in $\log_2 V$ bits; the VCids arriving at the receiver via the valid signals and at the sender via the update signals should be first decoded before being used in either side of the link. The complete list of wires used in a physical channel that accommodates V VCs is shown in Fig. 6.4, including also the appropriate packet framing signals isHead and isTail that are used to describe the id of the flit that is currently on the link.

The basic property of VC-based flow control is the interleaving of flits of different packets. In each cycle, a flit from a different VC can be selected and appear on the link after having consumed its corresponding credit. The flit once it arrives at the receiver is placed at the appropriate VC buffer indexed by the VC ID of the forward valid signals. Since the buffering resources of each VC at the receiver's side are completely separated, interleaving the flits of different packets does not create any problems, assuming that the VC-based flow control mechanism does not involve

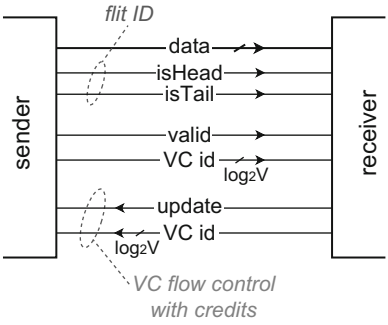

Fig. 6.4 When a physical channel supports many VCs the wires of a link that connect a sender and a receiver besides the flit's data and id should also include the necessary signals for VC-based flow control: a valid bit and a VC id that identify the outgoing flit and an update bit together with the corresponding VC id that addresses the VC of the returned credit

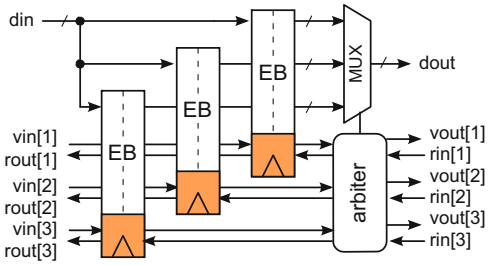

Fig. 6.5 A baseline VC-buffer architecture for 3VCs built by just replicating one 2-slot EB per VC and including an arbiter and a multiplexer at the read side of the VC buffer

any dependencies across VC, e.g., if the buffer of a certain VC is full to stop the transmission of flits from another VC. Examples of such dependencies arising from sharing the buffers used for the VCs will be discussed in the following sections.

6.2 Virtual-Channel Buffers

In the simplest form of single-cycle links the valid and the backpressure information needs one cycle to propagate in the forward and in the backward direction. Therefore, in a single-cycle channel without VCs, a 2-slot elastic buffer (EB) would suffice to provide lossless operation and 100 % throughput. Equivalently, a primitive VC buffer can be built by replicating one 2-slot EB per VC, and including a multiplexer, following the connections shown in Fig. 6.5 for the case of 3 VCs. Each EB will be responsible for driving the corresponding ready(i)/valid(i) signals while all of the them will be connected to the same data wires on the write side. When more buffering space is required a FIFO buffer per VC can be used in the place of a simple elastic buffer.

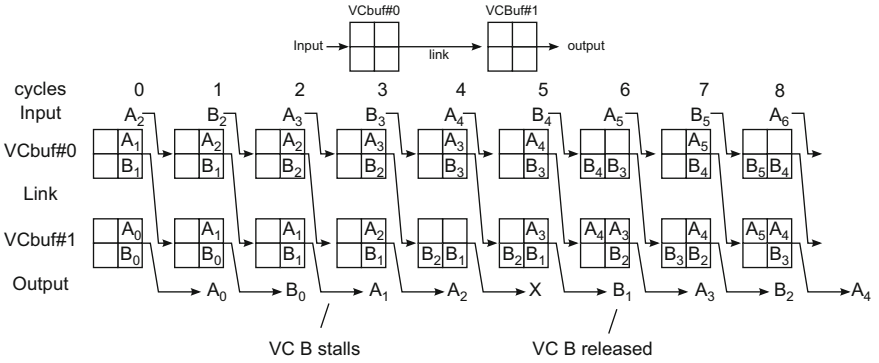

Fig. 6.6 Flit flow on a channel between two primitive VC buffers that employ a 2-slot EB for each VC

On the read side of the VC buffer, an arbitration mechanism will select only one of the valid VCs, that is also ready downstream, to leave the buffer. A packet that belongs to the ith VC can be hosted either to the same VC in the next buffer or to a different VC provided that it has won exclusive access to this VC. VC-based flow control does not impose any rules on how the VCs should be assigned between a sender buffer and a receiver buffer. Allowing packets to change VC in-flight can be employed when the routing algorithm does not impose any VC restrictions (e.g., XY routing does not even require the presence of VCs). However, if the routing algorithm and/or the upper-layer protocol (e.g., cache coherence) place specific restrictions on the use of VCs, then arbitrary in-flight VC changes are prohibited, because they may lead to deadlocks. In the presence of VC restrictions, the allocator/arbiter should enforce all rules during VC allocation to ensure deadlock freedom. The VC selection policies used inside the routers will be thoroughly discussed in the next chapter.

Figure 6.6 depicts a running example of a VC-based pipeline using a 2-slot EB per VC. The two active VCs each receive a throughput of 50 % and each VC uses only one buffer out of the two available per VC. The second buffer is only used when a VC stalls. This uniform utilization of the channel among different VCs leads to high buffer underutilization. The buffer underutilization gets worse when the number of VCs increases. In the case of V active VCs, although the physical channel will be fully utilized, each VC will receive a throughput of $1/V$ and use only one of the two available buffer slots since it is accessed once every V cycles. Only under extreme congestion will one see the majority of the second buffers of each VC occupied. However, even under this condition, a single active VC is allowed to stop and resume transmission at a full rate independently from the rest VCs. This feature is indeed useful in the case of traffic originating only from a single VC, where any extra cycles spent per link will severely increase the overall latency of the packet. However, in the case of multiple active VCs, whereby each

one receives only a portion of the overall throughput ($1/M$ in the case of M active VCs), allocating more than 1 buffer slot per VC is an overkill.

Larger buffers per VC are only needed for covering the increased round-trip time of the flow control mechanism as it will be described in the second part of this chapter or to absorb bursty traffic that rapidly fills up the buffers of the network in a specific direction. Still in these cases, only a subset of the total VCs will be active and the rest will stay idle leaving their buffers empty most of the time, thus increasing the buffer underutilization.

6.3 Buffer Sharing

Naturally, the answer to the underutilization of the VC buffers is buffer sharing; reuse some of the available buffer resources along many VCs (Tamir and Frazier 1992; Nicopoulos et al. 2006; Tran and Baas 2011). From a functional perspective, all variants of shared buffer architectures exhibit the same overall behavior: They manage multiple variable-length queues – one per VC in our case – and allow flits to be removed from the head or added to the tail of each queue. The individual variants differ in how they preserve the order of flits within each queue. At first, we will describe a generic shared buffer architecture utilizing credit-based flow control for each VC and then derive the minimum possible VC-based buffered architecture that employs sharing too and can work with ready/valid handshake similar to the primitive 2-slot EBs presented for the case of single-lane traffic.

In a shared buffer configuration, illustrated in an abstract form in Fig. 6.7, each VC owns a private space of buffers. When the private space of one VC is full the corresponding VC can utilize more space from a shared buffer pool. The minimum private space required is equal to 1 slot, while the shared buffer space can be larger.

When credit-based flow control is applied at the VC level the available number of credits for one VC directly reflects the available buffer slots for that VC. So, the maximum number of credits is fixed and equal to the buffering positions of each VC. In a shared buffer organizations any VC can hold an arbitrary number of buffer slots both in its private region as well as in the shared buffer space. Therefore,

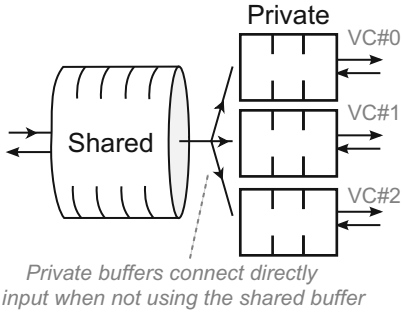

Fig. 6.7 The rough organization of a VC buffer that employs sharing of buffer space across different VCs. Each VC owns a private buffer space and all of them share the slots provided by a shared buffer module that is dynamically allocated to the requirements of each VC

the maximum allowed value for each credit counter may change dynamically. This feature may complicate a lot the update of the credit counters (Nicopoulos et al. 2006).

Instead, we present a different approach that keeps the depth of each credit counter constant and simplifies a lot the handling of the credits of each VC in a shared buffer configuration. The sender keeps one credit counter, for each downstream VC that refers to its private buffer space and a counter for the shared buffer that counts the available buffer slots in the shared region. A VC is eligible to send a new flit when there is at least one free position either at the private or the shared buffer ($creditCounter[i] > 0$ or $creditShared > 0$). Once the flit is sent from the ith VC, it decrements the credit counter of the ith VC. If the credit counter of the ith VC was already equal to or smaller than zero, this means that the flit consumed a free slot of the shared buffer and the counter of the shared buffer is also decremented.

Since the state of each VC is kept at the sender, the receiver only needs to send backwards a credit-update signal, including a VC ID, which indexes the VC that has one more available credit for the next cycle. On a credit update that refers to the jth VC, the corresponding credit counter is increased. If the credit counter is still smaller than zero, this means that this update refers to the shared buffer. Thus, the credit counter of the shared buffer is also increased. Please note that even if there is a separate credit counter for the shared buffer the forward valid signals and the credit updates refer only to the VCs of the channel and no separate flow control wiring is needed in the channel to implement a shared buffer at the receiver.

In this case, safe operation is guaranteed even if there is only 1 empty slot per VC. In the case of single-cycle links ($L_f = 1$, $L_b = 1$), each VC can utilize up to 2 buffer slots before it stops, and those positions are enough for enabling safe and full throughput operation per VC. Therefore, when each VC can utilize at least 2 buffer slots of either private or shared buffer space it does not experience any throughput limitations. If a certain VC sees only 1 buffer slot available then inevitably it should limit its throughput to 50 % even if it is the only active VC on the link.

The generic shared buffer architecture that includes a private buffer space per VC and a shared buffer space across VCs can be designed in a modular and extensible manner if we follow certain design rules (operational principles). First, any allocation decision regarding which VC should dequeue a flit from the buffer, is taken based only on the status of the private VC buffers; the private buffers act as parallel FIFOs each one presenting to the allocation logic just one frontmost flit per VC. Second, when the private buffer per VC drains one flit that empties one position, the free slot is refilled in the same cycle, either with a flit possibly present in the shared buffer, or directly from the input, assuming the new flit belongs to the same VC. Whenever the private buffer per VC cannot accommodate an incoming flit, a shared slot is allocated, where the flit is stored. As soon as the private space becomes available again, the flit is retrieved from the shared buffer and moves to the corresponding private buffer.

Every time a VC dequeues a flit from its private buffer, it should check the shared buffer for another flit that belongs to the same VC. Figure 6.8 demonstrates the

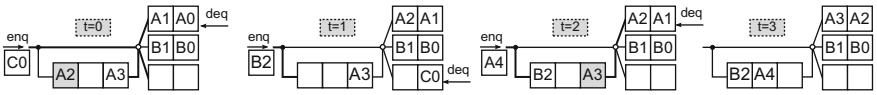

Fig. 6.8 An example of the interaction between the shared buffer (3 slots) and the private buffers (2 slots) per VC. Every time a flit is drained from the private VC buffer, the shared buffer is checked for another flit that belongs to the same VC. A VC uses the shared buffer only when it runs out of private buffer space

interaction between the shared buffer and the private buffer space per VC through a simple example. In this configuration, each VC owns a private buffer of 2 slots and all VCs share three extra buffer slots. In cycle 0, VC A owns 2 shared slots and dequeues a flit from its private buffer that was previously full. The empty slot in the private buffer of VC A should be refilled by a flit from the shared buffer. Therefore, VC A accesses the shared buffer to find the flits that match VC A and locate the oldest (the one that came first). The refill of the private buffer of VC A is completed in cycle 1. Then, in cycle 2, the same procedure is followed, effectively loading the private buffer of VC A with a new flit. The private buffer of a VC does not necessarily get new data from the shared buffer, but it can be loaded directly from the input, as done for VC C in cycle 0.

6.3.1 The Organization and Operation of a Generic Shared Buffer

Maintaining the flits' order of arrival among VCs is the main concern when designing a shared buffer. It should be as flexible as possible, without imposing any restrictions on the expected traffic patterns. The way this is achieved leads to different implementations. For example, the self-compacting shared buffers (Ni et al. 1998; Park et al. 1994) require the flits of a single VC to be stored in contiguous buffer positions. To enable this continuity in the storage of flits of each VC, every time a flit enters or leaves the queue of a certain VC the data of all queues should be shifter either to make room for the incoming flit or to close the gap left by the outgoing flit. To avoid shifting that may lead to increased power consumption, the shared buffer can employ pointers that allow locating flits by reference, although they may be stored in practically random addresses (Tamir and Frazier 1992; Katevenis et al. 1998).

An example of such an approach is presented in Fig. 6.9. Apart from the main buffer space used to store flits (Buffer Memory), the shared buffer uses a list of single-bit elements which tells whether an address actually contains data (Availability List) and a Linked List, used to track flits' order of arrival. If a flit for VC k is stored in address a of the buffer memory, then the next flit for VC k is stored in the address pointed by element a of the Linked List (a NULL pointer

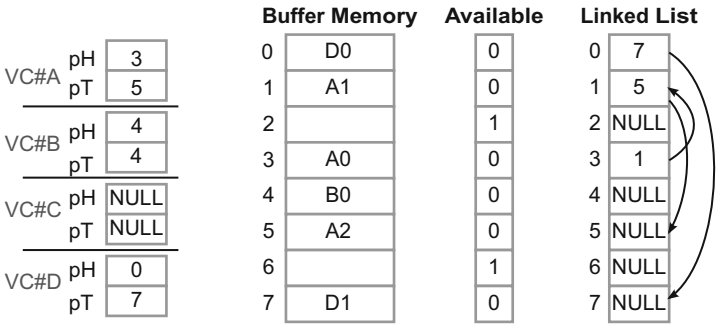

Fig. 6.9 Abstract organization of a shared buffer using pointer logic. pH and pT represent the head and tail pointer of each queue

Enqueue NewFlit at VC #k

```
fs = head(avail)          // Get the first free slot
buffer[fs] <- flit_new    // Store the new flit
l_list[pT[k]] <- fs       // Link previous tail of VC k to new flit
pT[k] <- fs               // Update tail pointer of VC k
l_list[fs] <- NULL        // Make new flit the last flit of the queue
avail[fs] <- 0            // Mark the slot as occupied
```

Dequeue OutputFlit from VC #k

```
flit_out <- buffer[pH[k]] // Read flit from the head address
avail[pH[k]] <- 1         // Mark the buffer slot as available
pH[k] <- l_list[pH[k]]    // Set the head pointer to point to the
                          // address of the next flit
```

Fig. 6.10 Operations involved when enqueuing or dequeuing a flit from the shared buffer

means that no other flit exists for VC k). Combined with its head and tail pointers (pH and pT), a VC can always find its flits in the shared buffer, in the order they initially arrived.

The necessary operations executed in case of enqueue or dequeue are shown through the abstract description of Fig. 6.10. Assume that a flit arrives for VC A at the input of the shared buffer, which is currently in the state of Fig. 6.9. A search is initiated and the first free slot is located using a fixed priority arbiter (in that case, $fs = 2$). Then, in parallel, (a) the actual flit data are stored in address 2 of the Buffer Memory, (b) the fs value is stored in slot 5 of the linked list, as pointed by the tail pointer of VC A and (c) VC A's tail pointer is replaced by the fs value. In the opposite case, where a dequeue is requested by VC A, the shared buffer's output is driven by the data stored in the address pointed by head pointer ($pH = 3$ for VC A). Then, the element 3 of the linked list is accessed to retrieve the address of the next flit, and it is used to replace VC A's head pointer, while the value of element 3 is reset to NULL. Finally, address 3 is marked as available in the availability list.

Notice that the use of a tail pointer is not mandatory. It would have been possible to locate it implicitly, after following the path of the linked list's pointers, starting

from the head position until a NULL pointer is found. However, this would add significant latency, that is not needed when being able access the last flit position directly, with almost no overhead.

6.3.2 *Primitive Shared Buffer for VCs: ElastiStore*

Buffer sharing can be pushed to the limit and design low-cost VC buffers that offer the minimum buffering possible and still allow to a single VC to enjoy 100 % throughput of data transfer. The buffering architecture, called *ElastiStore*, utilizes only $V + 1$ buffers for V VCs (Seitanidis et al. 2014a). Each VC owns a single buffer, which is enough in the case of uniform utilization, where each VC receives a throughput of $1/M$, with $2 \leq M \leq V$. Furthermore, when a single VC uses the channel without any other VC being active, i.e., $M = 1$, it receives 100 % throughput, and, in the case of a stall, it may use the additional buffer available in ElastiStore. This additional buffer is shared dynamically by all VCs, although only one VC can have it in each clock cycle. However, when all VCs, except one, are blocked, and the shared buffer is utilized by a blocked VC, then the only active VC will get 50 % of the throughput, since it effectively sees only one buffer available per channel. Note that the baseline VC-based buffer of Fig. 6.5, which allocates 2 buffers to each VC would allow this active VC to enjoy full channel utilization.

Figure 6.11 illustrates an example of flit flow between two ElastiStores that each one supports 2 VCs. In the first cycles, each VC receives 50 % of the throughput per channel ($M = 2$), and, at each step, they utilize only one buffer slot. In those cycles, the shared registers of the two ElastiStores are not utilized. The shared buffers are used between cycles 4 and 7 to accommodate the stalled words of VC B. In those cycles, VC A – which is not blocked – continues to deliver its words to the output of the channel.

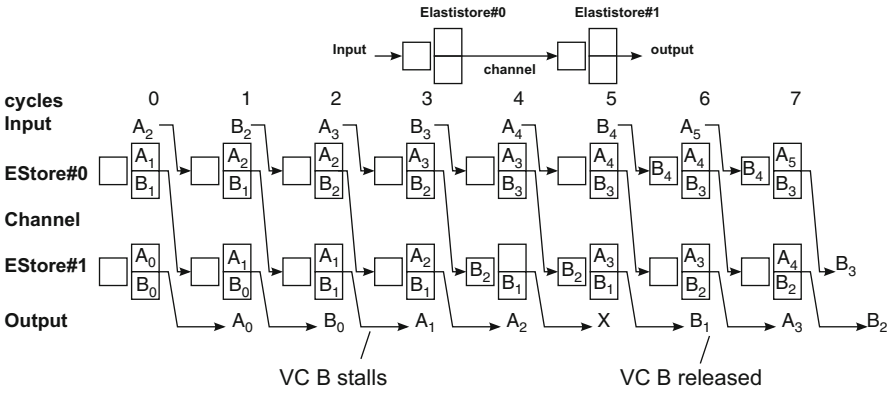

Fig. 6.11 An example of the of flow of flits on a channel that supports 2 VCs and utilizes ElastiStores at both ends of the link

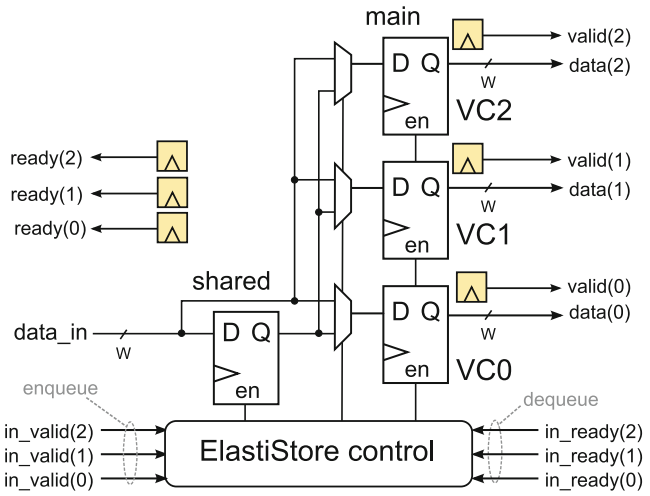

Fig. 6.12 The organization of an ElastiStore primitive for 3 VCs. ElastiStore consists of just a single register per VC (main registers) along with a shared register that is dynamically shared by all VCs

ElastiStore saves many buffer slots per VC buffer, as compared to the baseline VC-based EB of Fig. 6.5, and limits throughput only under heavy congestion that blocks all the VCs except one. In the case of light traffic, a single active VC receives 100 % throughput without any limitation.

ElastiStore can be designed using the datapath shown in Fig. 6.12, which consists of a single register per VC (main registers) along with a shared register that is dynamically shared by all VCs. The select signals of the bypass multiplexers, the load enable signals of the registers, as well as the interface ready/valid signals are all connected to ElastiStore control.

When a new flit that belongs to the ith VC arrives at the input of ElastiStore, it may be placed either in the main register of the corresponding VC, or in the shared register. If the main register of the ith VC is empty, or becomes empty in the same cycle, the flit will occupy this position. If the main register is full, the incoming flit will move to the shared buffer. Concurrently, once the shared buffer is utilized, all the VCs that have their main register full will stop being ready to accept new data, while those with an empty main register remain un-affected. In ElastiStore, any VC is ready to accept a new flit if at least one of two registers is empty: either the main register corresponding to said VC, or the shared one.

ElastiStore dequeues data only from the main registers. The shared register acts only as an auxiliary storage and does not participate in any arbitration that selects which VC should be dequeued. When the main register of a VC dequeues a new flit and the shared buffer is occupied by the same VC, the main register of this VC is refilled by the data stored in the shared buffer in the same cycle. The shared buffer cannot receive a new word in the same cycle, since its readiness – which releases

all VCs that have their main register full – will appear on the upstream channel in the next clock cycle. The automatic data movement from the shared to main buffer avoids any bubbles in the flow of flits of the same VC and achieves maximum throughput.

ElastiStore should be considered as the equivalent of the primitive the 2-slot EB used in the single-lane case where a main and an auxiliary HBEB are used for allowing full transmission throughput. It allows the implementation of VC-based flow control using close to the absolute minimum of one buffer slot per VC, without sacrificing performance and without introducing any dependencies between VCs, thus ensuring deadlock-free operation.

6.4 VC Flow Control on Pipelined Links

When the delay of the link exceeds the preferred clock cycle, one needs to segment the link into smaller parts by inserting an appropriate number of pipeline stages. In the case of single-lane channels, the role of the pipeline stages is covered by EBs, which isolate the timing paths (all output signals – data, valid, and ready – are first registered before being propagated in the forward or in the backward direction), while still maintaining link-level flow control, as shown in Fig. 6.13a, and discussed in Sect. 2.5. In the case of multi-lane channels that support VCs, we can achieve the same result by replacing the EBs with the VC buffers of Fig. 6.5 or with ElastiStores (see Fig. 6.13b). Although this approach works correctly and allows for distributed buffer placement, while still supporting VC-based flow control, it is not easily handled in complex SoCs, since the addition of many registers (at least one for each VC) in arbitrary positions, may create layout and physical integration problems.

Using VC buffers at the ends of the link and simple EBs on the link introduces dependencies across VCs, since the flow control information per VC needs to be

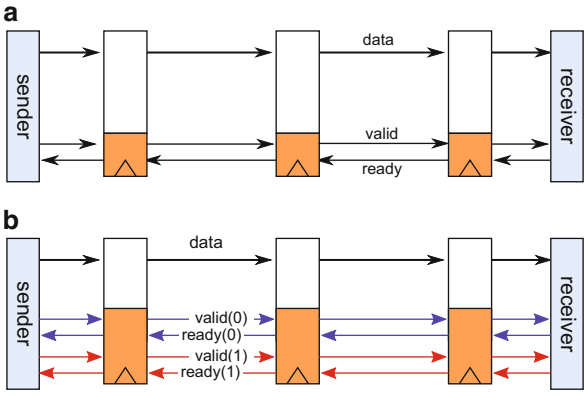

Fig. 6.13 Pipelined links with (**a**) EBs that support a single-lane operation and with (**b**) EBs for a link that supports multiple VCs. The VC-based buffers distributed on the link can be either ElastiStores or baseline buffers that use a separate 2-slot EB per VC

serialized under a common ready/valid handshake; if one VC stops being ready, all the words on the link should stop, irrespective of the VC they belong to. Such dependencies ruin the isolation and deadlock-freedom properties of the VCs and require ad-hoc modifications to the flow control mechanism.

6.4.1 Pipelined Links with VCs Using Ready/Valid Flow Control

In the case of pipelined channels that employ ready/valid flow control for each VC, we can rely on simple registers for pipelining the data and the ready/valid handshakes signals on the link, as shown in Fig. 6.14. In this case, the flits cannot stop in the middle of the link, since the pipeline registers do not employ any flow control. Many words may be in-flight, since it takes L_f cycles for the signals to propagate in the forward direction and L_b cycles in the backward direction. Therefore, the buffers at the receiver need to be sized appropriately to guarantee lossless and full throughput operation. In the case of pipelined links, as also done in the single-lane channels, any VC declares that it holds valid data after checking the readiness of the corresponding downstream buffer, else multiple copies of the same valid data will appear at the receiver's VC buffer.

First of all, assume that only one VC, i.e., the ith one, is active and the remaining VCs do not send or receive any data. When the buffer of the ith VC is empty, it asserts the ready(i) signal. The sender will observe that ready(i) is asserted after L_b cycles and immediately starts to send new data to that VC. The first flit will arrive at the receiver after $L_f + L_b$ cycles. This is the first time that the receiver can react by possibly de-asserting the ready(i) signal. If this is done, i.e., ready(i) = 0, then under the worst-case assumption, the receiver should be able to accept the $L_f - 1$ flits that are already on the link, plus the L_b flits that may arrive in the next cycles (the sender will be notified to stop with a delay of L_b cycles). Thus, when the ith VC stalls, it should have at least $L_f + L_b$ empty buffers to ensure lossless operation. Actually, the minimum number of buffers for the ith VC reduces to $L_f + L_b - 1$, if we assume that the sender stops transmission in the same cycle it observes that

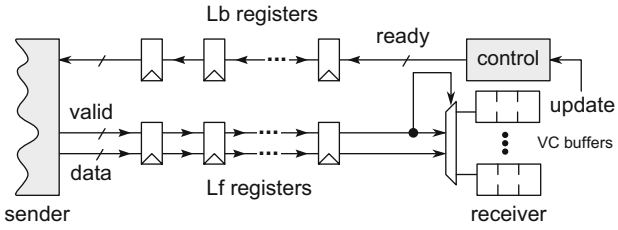

Fig. 6.14 Abstract model of a pipelined link with multiple VCs and independent ready/valid handshake signals per VC

ready(i) = 0. Thus, a channel with V VCs and a round-trip time of $L_f + L_b$ needs at least $V(L_f + L_b - 1)$ slots. When many VCs are active on the channel, their flits would be interleaved and the probability that all $L_f + L_b - 1$ flits belong to the same VC is small. However, the worst-case condition calls for providing as much buffer space to each VC as needed to prevent the dropping of any flit, independent of the traffic conditions on the remaining VCs.

Unfortunately, giving the minimum number of buffers to each VC has some throughput limitations. Assume that the ith VC has occupied all its buffer slots at the receiver and starts draining the stored flits downstream at a rate of one flit per cycle. After $L_f + L_b - 1$ cycles, the buffer will be empty (no more flits to drain) and the ready(i) signal will be asserted, causing the fist new flit to arrive $L_f + L_b - 1$ cycles later (the ready(i) signal is asserted in the same cycle that the last flit is drained). Therefore, in a time frame of $2(L_f + L_b - 1)$ cycles, the receiver was able to drain only $L_f + L_b - 1$ flits, which translates to 50 % throughput. Thus, a single active VC can enjoy 100 % throughput when it has $2(L_f + L_b - 1)$ buffers and is ready when the number of empty slots is at least $L_f + L_b - 1$. The baseline VC-based EB of Fig. 6.5 employed in single-cycle links ($L_f = L_b = 1$) is a sub-case of the general pipelined link and achieves 100 % throughput of lossless operation using 2 buffers per VC.

Buffer Sharing on Pipelined Links

In the case of pipelined links the required buffer space per VC grows fast with the increasing forward and backward latency of the flow control signals. Buffer sharing should be employed in this case too in order to minimize the buffering requirements. In the case of single-cycle links the private buffer space of each VC and the shared buffer space across VCs can drop down to one slot of private space and one shared buffer slot as shown by the primitive ElastiStore modules. In the case of large latencies different configurations should be followed.

In the general case of multi-cycle links, instead of having $2(L_f + L_b - 1)$ buffer slots for each VC, we dedicate $L_f + L_b - 1$ slots private to each VC needed for safe operation and $L_f + L_b - 1$ more, which can be dynamically shared by all VCs. In this way, we remove $L_f + L_b - 1$ of private buffer slots per VC and keep the extra $L_f + L_b - 1$ buffers needed only once in a dynamically shared manner. In this configuration, any VC is ready, as long as there are $L_f + L_b - 1$ empty slots either in its private buffer, or accounting for the free space in the shared buffer as well. Therefore, a single active VC can enjoy 100 % throughput, while, in the case where the shared buffer is full, every active VC cannot get more than 50 % of throughput (it can receive/send $L_f + L_b - 1$ flits at most every $2(L_f + L_b - 1)$ cycles). Keep in mind that when many VCs are active, the throughput per VC is much lower than 50 %. Under high utilization, the channel is already shared by many VCs, and achieving high-throughput per independent VC does not give much benefit, unless it is the only active VC.

If we try to minimize further the overall buffer space it means that we need to minimize private buffering too; the shared buffer remains the same holding $L_f + L_b - 1$ flits. If the private buffer space per VC drops below $L_f + L_b - 1$ slots, say k, it means that safety per VC cannot be guaranteed by the private buffer only. The $L_f + L_b - 1$ slots needed per VC for safety should be covered by using both the k private buffers of each VC and some positions of the shared buffer. This configuration, and using an independent ready/valid handshake for each VC, may create dependencies across VCs that can possibly lead to a deadlock. Assume, for example, that the ith VC uses its all of its private buffer, e.g., k slots with $k < L_f + L_b - 1$ and the rest needed to cover $L_f + L_b - 1$ buffers in total from the shared buffer; it leaves less than $L_f + L_b - 1$ free slots in the shared buffer. Then, every other VC must de-assert its ready signal, even if its private buffer is empty, since the available free slots for each VC are less than $L_f + L_b - 1$, which are needed to guarantee safe operation per VC. Under this scenario, the traffic on one VC is allowed to block the traffic on another VC, which removes the needed isolation property across VCs. Such dependencies are removed if the shared buffer has more buffers to share across VCs and each VC limits the maximum number of slots it can use from the shared buffer (Becker 2012a).

6.4.2 *Pipelined Links with VCs Using Credit-Based Flow Control*

Using credits, safe operation is guaranteed even if there is only 1 empty slot per VC of private space, but with very limited throughput due to the increased round-trip time; no flit can be in flight if it has not consumed a credit beforehand thus there is no minimum requirement for safe transmission. With credits, once a credit update is sent backwards for a VC it means that a new flit will arrive for this VC after $L_f + L_b - 1$ cycles. Therefore, offering to a single VC $L_f + L_b - 1$ buffers, means that at the time the last flit is drained from the VC the first new one will arrive thus leaving no gaps in the transmission and offering full throughput. A single active VC can utilize both its private space and all the positions of the shared buffer and achieve 100 % throughput, by effectively allowing this VC to use $L_f + L_b$ buffers in total, as needed by credit-based flow control.

The $L_f + L_b$ buffers needed for one VC to achieve 100 % throughput in a pipelined link with credit-based flow control can be achieved in many configurations between the private and the shared buffer space. For example, the VC buffer can allocate 1 buffer of private space per VC and have a shared buffer that can hold $L_f + L_b - 1$ flits. Equivalently, in another organization, each VC can have a 2-slots of private buffering and the shared buffer can be sized to host a total of $L_f + L_b - 2$ flits.

6.5 Take-Away Points

Virtual channels is analogous to adding lanes in a street network although implemented virtually and allowing flits that belong to different lanes to appear on the physical link in a time-multiplexed manner. The existence of multiple VCs requires the enhancement of the flow control mechanism and the associated buffering architectures. Sharing the buffers across different VCs, when applied with care, can increase buffer utilization and reduce the overall cost of supporting multiple VCs. The latency in the forward and the backward propagation of the flow control signals increases the minimum buffering requirements for supporting full transmission throughput and complicates buffer sharing.