Chapter 7
Baseline Virtual-Channel Based Switching Modules and Routers

In this chapter we describe the operation and the microarchitecture of a virtual channel based router by analyzing in detail the subtasks involved, the dependencies across these tasks, and the extra state needed for their implementation. This chapter covers single-cycle implementations of virtual-channel-based routers, while high-speed alternatives and pipelined organizations are left for the following chapters. Every router should support arbitrary connections between inputs and output ports that connect to independently flow-controlled links. The links in this case host many virtual channels (VCs) that are interleaved in a time-multiplexed manner.

We start our discussion on the implementation VC-based switching by describing the organization and the operation of a many-to-one connection that connects many input links to one output link that each one supports a set of virtual channels. Then, we generalize this design to a complete VC-based router that supports many-to-many connections, while still allowing the existence of many VCs in parallel.

7.1 Many to One Connection with VCs

The abstract organization of a many-to-one connection that supports multiple VCs at the input and the output channels is shown in Fig. 7.1. Each input is equipped with as many parallel buffers as the number of VCs. The switching module connects the input VC buffers to a single output via a simple physical link. The flits passing from the output of the switching module should be placed to a buffer that corresponds to the VC that they belong to. The parallel output VC buffers can be placed either at the output of the switching module or at the other side of the link. In this configuration we chose to include the output VC buffers at the other end of the link, and include at the output of the switching module only a pipeline register that just isolates the internal timing paths of the switching module from the link. Even if the output VC buffers are placed far from the output of the switching module, any state variables

G. Dimitrakopoulos et al., *Microarchitecture of Network-on-Chip Routers: A Designer's Perspective*, DOI 10.1007/978-1-4614-4301-8_7

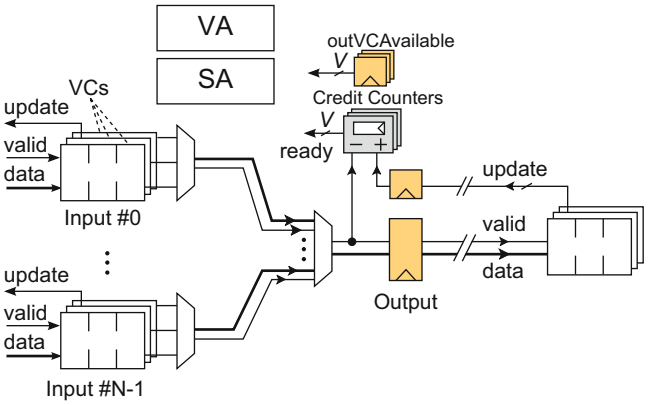

Fig. 7.1 Multiple inputs connect to a single output, with multiple parallel queues on each side, one for each VC

required per output VC are stored locally at the output of the switching module, but refer to state of the VC buffers at the other side of the link.

Each input can receive the flit of only one VC in each clock cycle. Therefore, it is enough for each input to try to send to the output at most one flit from a selected input VC. To support this rate of outgoing traffic per input the switching module consists of two levels of multiplexing. In the first level (per input) a multiplexer selects one VC from all input VCs, while, in the second level (at the output), the selected input VCs are multiplexed to the output. If we need a large rate of outgoing flits per input, multiple VCs of the same input should be able to reach the output multiplexer.

7.1.1 State Variables Required Per-Input and Per-Output VC

Similar to wormhole routers, the inputs and the outputs of the switching module should be enhanced with some extra state variables that allow scheduling, both at the VC-level and at the physical port level, to be performed and combined to the flow control mechanism of the input and the output channels.

First of all, the state needed involves the flow control mechanism. In the examples used in this chapter we adopt the credit-based flow control. Therefore, at the output of the switching module a set of credit counters is used; one for each VC. The maximum value of each counter is equal to the maximum number of positions available per VC at the output VC buffers. The credit counters produce the necessary ready signals for each VC, e.g., $ready[i] = 1$ when $creditCounter[i] > 0$. Once a flit from an input VC leaves the output of the switching module (or when it knows that it has gained access to the output) it consumes one credit by the corresponding

credit counter. The credit counters are incremented depending on the update signals they receive from the output VC buffers.

In wormhole routers, the output of the switching module in a many-to-one connection, kept an *outAvailable* variable that denoted if the output has been allocated to a certain input. The variable was set by the head flits and was locked for the rest flits of the packet; the tail flit released the availability of the output. In VC-based routers, each flit fights on its own for gaining access to the output port. Output locking is avoided and different kinds of flits can be interleaved at the output, provided that they belong to different output VCs and thus stored in different buffers at the other end. Such flit interleaving reduces effectively head-of-line blocking and increases the observed throughput per output.

In the case of VC-based routers, the output lock mechanism used in wormhole connections is maintained, but at the output VC level. Each packet has to choose a VC at the output before leaving an input VC. Matching input VCs to output VCs is done once per packet via the head flit by the VC allocator (VA), while the rest flits (body and tail) of the same packet inherit the allocated output VC. To support this ownership mechanism V *outVCAvailable* flags are maintained at the output of the switching module, each one corresponding to a different VC of the output. When $outVCAvailable[i] = 1$, it means that the ith output VC is available to be allocated to any input VC ($N \times V$ input VCs are eligible to connect to this output VC; V VCs per input). When $outVCAvailable[i] = 0$, it means that the ith output VC has been allocated to a packet of a certain input VC and it will be released when the tail flit of the packet passes through the output of the switching module. Allowing packets to change VC in-flight can be employed when the routing algorithm and/or the upper-layer protocol (e.g., cache coherence) do not place any specific restrictions on the use of VCs. In the presence of VC restrictions, the VC allocator will enforce all rules during VC allocation to ensure deadlock freedom.

Equivalently, the implementation of this input-output VC ownership mechanism, requires each input VC to hold two state variables per input VC: $outVCLock[i]$ and $outVC[i]$. When the single-bit $outVCLock[i]$ is asserted, the ith input VC has been matched to an output VC, while the id of the output VC assigned to this input VC is specified by the value of $outVC[i]$. In the opposite case ($outVCLock[i] = 0$), the ith input VC has not been assigned yet to an output VC and the value of $outVC[i]$ is irrelevant.

7.1.2 Request Generation for the VC Allocator

Each input is equipped with an input controller that is responsible for the orchestration of all the intermediate steps needed before a flit from an input VC is transferred to a certain output VC. The part of the input controller that is responsible for preparing the requests to the VC allocator and gathering the corresponding grants is shown in Fig. 7.2. Once the input controller detects the presence of a head flit of an input VC with un-assigned output VC ($outVCLock[i] = 0$), it should form

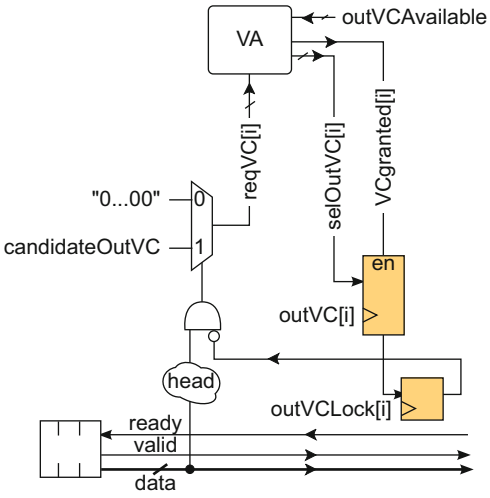

Fig. 7.2 The request generation and grant handling logic regarding the process of VC allocation. Each input VC is responsible for sending new requests to the VC allocator that matches input to output VCs according to output VC availability and the state of the requesting input VCs

the appropriate requests to the VC allocator. Each input VC i sends to the VC allocator a set of candidate output VCs $candidateOutVC[i]$ (V bits). If the packet is not allowed to change VC while traversing the network from source to destination, then $candidateOutVC[i] = i$. If there is no restriction on the selection of the output VC then $candidateOutVC[i]$ vector may have several bits asserted, even all V of them, meaning that it is requesting any available output VC.

The VC allocator should find a match between requesting input VCs ($reqVC[i]$) and the available output VCs. By merging the $reqVC$ vectors produced by all input VCs, we can represent the requests given to the VC allocator in a matrix of $N \times V$ rows and V columns. When $reqVC[i][j] = 1$ means that the ith input VC is requesting output VC j. The ith input VC belongs to the kth input where $k = i \div V$. The example of Fig. 7.3 shows the output VC requests for a 2-input switching module that hosts three VCs per physical channel. A valid match to the request matrix should contain at most 1 bit asserted per row and per column, meaning that an input VC cannot be assigned to more than one available output VC. Likewise, an available output VC cannot be assigned to more than one input VC. The match shown in Fig. 7.3 satisfies all required conditions. Please notice that the requests that correspond to unavailable output VCs are filtered from the allocation process.

The VC allocator returns its decision to all input controllers, where the information is organized per input VC, as shown in Fig. 7.2. Each input VC gets the $selOutVC[i]$ (V bits in onehot form) which is a subset of the $candidateOutVC[i]$ and indexes the output VC that the VC allocator selected for the ith input VC. It also receives a single-bit flag $VCgranted[i]$, that when asserted informs the ith input VC that the match with output VC $selOutVC[i]$ was indeed successful. In this case, $outVC[i] \leftarrow selOutVC[i]$ for use by the rest flits of the packet, while $outVCLock[i]$ variable is set to 1. Both variables will be reset once a tail flit is dequeued. If $VCgranted[i] = 0$, the ith input VC has not received an output VC

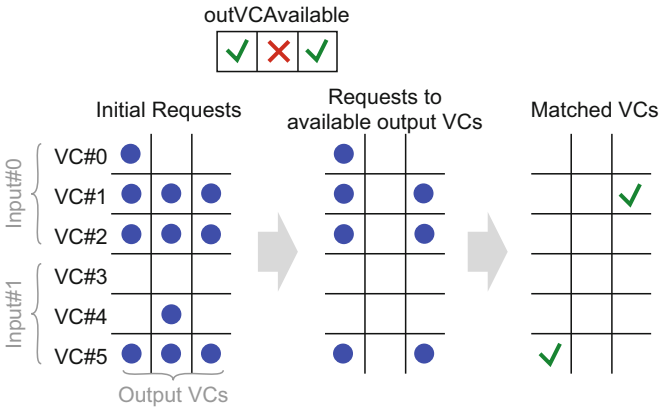

Fig. 7.3 An example output VC request matrix that feeds the VC allocator. The switching module connects 2 inputs that each one hosts 3 VCs. An input VC may request any of the output VCs, while the requests that correspond to unavailable output VCs are filtered from the allocation process. The requested and available output VCs will be assigned to one of the requesting input VCs, while making sure that no input VC is assigned to more than one output VC

yet and should retry in the next cycle. The assignment selected by the VC allocator, besides the input controllers, is also used to update the status of the *outVCAvailable* flags accordingly, so that no other input VC is allowed to request it on a future cycle, until it is released by the tail flit of the same packet.

Once an input VC succeeds in allocating an output VC it should stop issuing any requests to the VC allocator. This stop of requesting for an output VC is critical, since the VC allocator does not have any mechanism to understand that an input VC already holds an output VC, and may grant to it another available output VC. Therefore, a head flit that has succeeded in VC allocation, but still remains at the input VC buffer due to possible lack of available credit at the output VC buffers, should not make any further request. The only condition that qualifies $candidateOutVC[i]$ to reach the VC allocator, is when a head flit is present at the frontmost position of an input VC buffer and observes its local $outVCLock[i]$ being 0.

7.1.3 Request Generation for the Switch Allocator

Once an output VC is allocated, the packet is allowed to move to the next stage of switch allocation (SA), in which it has to fight with other input VCs for getting access to the output port. Unlike VA, which is performed once per packet, switch allocation is performed by every flit independently. The switch allocator of the

many-to-one switching module takes many requests and grants only one of them. An input VC can have a valid request to the switch allocator when three conditions are satisfied:

Valid flit: The ith input VC is not empty in the current cycle, i.e., there is a valid flit at the frontmost position of the corresponding input VC buffer.

Output VC already allocated: The input VC has been assigned to an output VC either in the same or in a previous cycle. When $outVCLock[i] = 1$, the ith input VC has already allocated an output VC with id equal to $outVC[i]$. In case that $outVCLock[i] = 0$ but $VCgranted[i] = 1$, it means that the flit (for sure a head flit) is allocated to an output VC in this cycle and the id of the output VC is equal to $selOutVC[i]$.

The output VC has enough credits: A given input VC can only request access to the output port if its destination VC has at least one credit available. Therefore, the ith input VC should check whether $ready[outVC[i]] = 1$. The ready signal of all output VC credit counters are distributed to all input VCs. The ready bit that corresponds to the matched output VC is selected by $outVC[i]$ or by $selOutVC[i]$, depending on whether the corresponding flit has already allocated an output VC in a previous cycle, or, it is allocated to a new output VC in the same cycle that prepares the requests for SA (Fig. 7.4).

The requests of all input VCs are gathered and sent to the switch allocator. The switch allocator is responsible for selecting one eligible input VC, and driving the per-input and output data multiplexers, according to that selection.

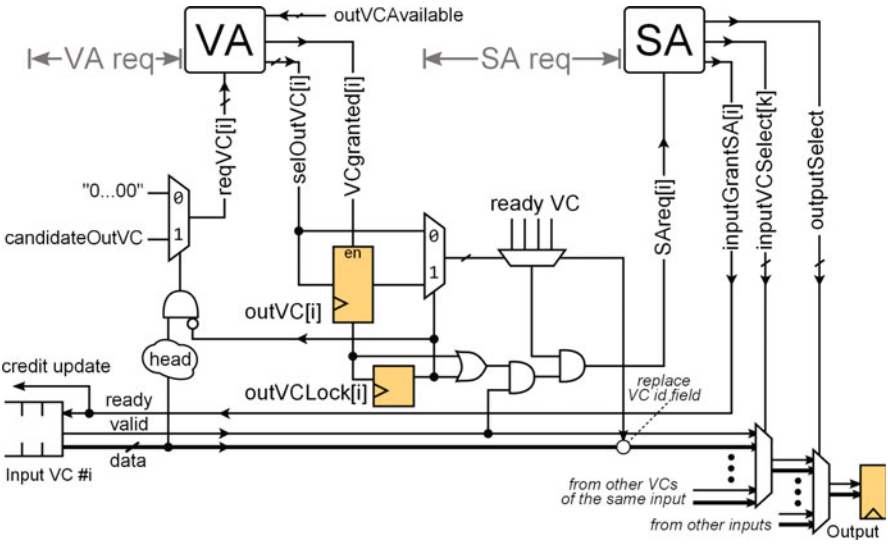

Fig. 7.4 The per-input-VC logic that implements request generation and grant handling for both VA and SA allocation stages in a many-to-one connection that supports VCs

7.1.4 Gathering Grants and Moving to the Output

Each input VC receives a vector of wires from the switch allocator called *inputGrantSA*. When $inputGrantSA[i] = 1$ it means that the ith input VC has been granted to move in this cycle to the output port. The flit from the selected input VC is dequeued and transferred to the data output of the input controller and from there to the output multiplexers.

In the most generic configuration, the input VCs are allowed to change VC in flight, i.e., when moving from input to output. Thus, the id of the input VC buffer that currently holds the outgoing flit may be different from the id of the output VC buffer that has been allocated to this packet. In this case, the departing flit, while moving to the output, should also change accordingly the VCid field that carries with it. The new VCid is needed at the output of the switching module for consuming the credit from the appropriate credit counter as well as at the output VC buffers for ensuring that the flit will be written to the correct buffer. The new VCid of the outgoing flit is equal to $outVC[sel]$ where *sel* is the input VC that won switch allocation, i.e., $inputGrantSA[sel] = 1$. Finally, keep in mind that when a tail flit is leaving the ith input VC, it de-allocates all resources reserved per packet at the input controller, by resetting both $outVCLock[i]$ and $outVC[i]$ variables.

The per-input and the output multiplexer of the switching module are driven by the switch allocator and manage to carry the winning flit from the selected input VC to the output. When the flit passes the output of the switching module it decrements the credit counter of the new VC and in the next cycle it is forwarded to the link. Since credit availability has been checked before switch allocation, the flit that arrives at the output will always leave in the next cycle and cannot stop there. In the case that the outgoing flit is a tail one, the output should also reset the corresponding *outVCAvailable*.

In the place of the output pipeline register one could have used complete VC buffers. In that case, the credits and the status of the output VCs would refer to these local output VC buffers and not to the VC buffers at other side of the output link. This configuration does not change the design of the VC-based switching module; the only changes involve the credit-based flow control mechanism and to which buffers it refers to.

7.1.5 The Internal Organization of the VC Allocator for a Many-to-One Connection

The VC allocator receives the output VC requests of all input VCs and tries to find a one-to-one matching between requesting input VCs and available output VCs. In the most general case, each input VC may have many candidate output VCs, some of which may refer to already allocated ones. Therefore, masking the requests with

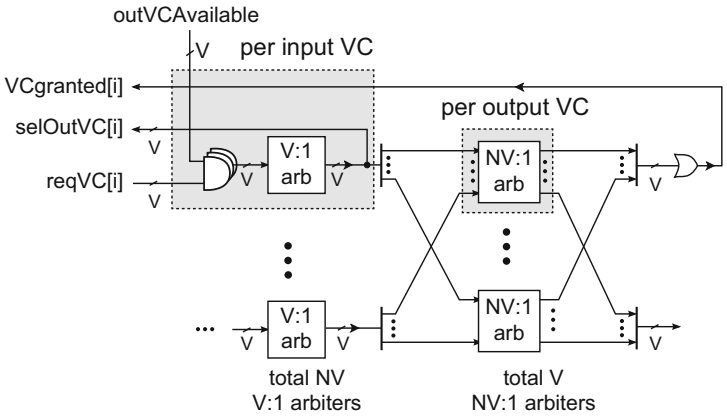

Fig. 7.5 The organization of a VC allocator for a many-to-one connection. In VA1, each input VC independently selects to request one of the available output VCs. Then, after VA2, each output VC selects to which input VC will be allocated

output VC availabilities should be performed first, before any arbitration occurs. The first arbitration, called VA1, is done per input VC with the goal to select the output VC that each input VC will finally ask for, thus limiting potential requests for output VCs to only one. Since the first stage of arbitration is done independently per input VC, many VCs may select the same output VC. As a result, a second arbitration step is required, called VA2, which is performed per output VC, selecting only one input VC to match the corresponding output VC. The organization of this two-step allocation process between input and output VCs is shown in Fig. 7.5.

The VC allocator in the case of a many-to-one connection includes a $V : 1$ arbiter per input VC and a $N \times V : 1$ arbiter per output VC, as shown in Fig. 7.5. The selected output VC ($selOutVC[i]$) for the ith input VC is decided during VA1. If the selected output VC is indeed allocated to the ith input VC, is revealed by $VCgranted[i]$ that is produced after reorganizing the results of the output VC arbiters and gathering the grants that correspond to the same input VC using a wide OR gate.

An example of the operation of the VC allocator, showing also the intermediate grants produced by the VA1 stage of arbitration, is shown in Fig. 7.6. Please notice that since VC allocation is done independently for each input VC, it is possible that multiple VCs of the same input to allocate an output VC in the same cycle. In the example shown in Fig. 7.6 VC#1 and VC#2, that both belong to input 0, are matched to output VC#2 and VC#0 respectively in the same allocation round.

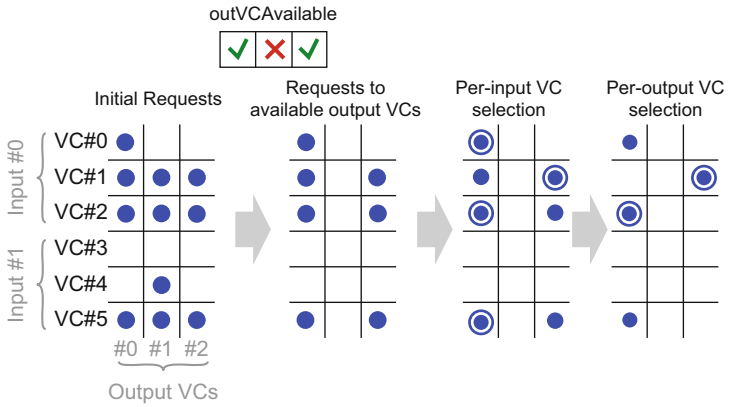

Fig. 7.6 An example of the operation of a VC allocator for a 2-input-1-output connection that hosts 3VCs. Multiple output VCs may be requested by a single input VC (Initial Requests), but only the available ones will qualify to the per-input VC allocation stage, in which only one will be selected. Then, each output VC will be assigned to one of the requesting input VCs. In this way, no output VC can be assigned to more than one input VC and no input VC can allocate more than one output VCs. The *circles* around the *bullets* illustrate the grants of VA1 (per-row) and VA2 (per-column) arbitration stages

7.1.6 The Internal Organization of the Switch Allocator for a Many-to-One Connection

Switch allocator services the requests of all input VCs that have been matched to an output VC and have also the available credits. Input VCs share an input port of the output multiplexer (only one VC per input can be served in each clock cycle). Therefore, switch allocation is done in two steps. The first step, called SA1, involves a local per input arbitration that selects which input VC to promote to the output arbiter. The second global arbitration step, called SA2, selects one valid input to connect to the output.

The organization of the switch allocator is shown in Fig. 7.7. It receives an output request bit per input VC and using a local $V : 1$ arbiter (SA1) selects one input VC from each input to participate to the next arbitration step. The global arbitration step (SA2) sees one request per input. An input has a valid request as long as the local arbiter gave at least one grant.[1] The grant signals of the output arbiter are given back to all inputs and also given to the output multiplexer for setting up the appropriate input-output connection. Each input receives 1 bit that denotes if any VC from this input was granted. Once this information reaches the input, it is combined with the decision of SA1 and prepares the winning input VC for sending a new flit to the crossbar, as shown in Fig. 7.7.

[1]This can be performed in parallel to SA1 by checking if the input arbiters have at least one request.

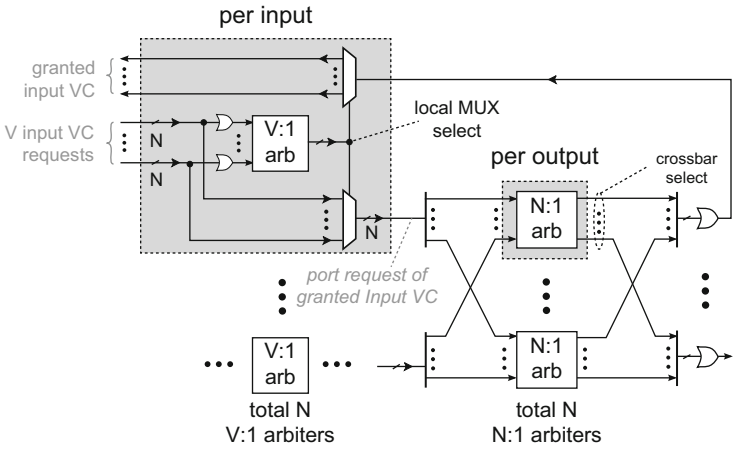

Fig. 7.7 The switch allocator for a many-to-one connection requires two stages of arbitration. One per input that selects one input VC from each input port, and, one per output that finally grants on the competing inputs

In the routers that do not support VCs and presented in Chaps. 3 and 5, an arbiter updates its priority whenever it delivers at least one grant. However, SA1 arbiters should update their priority only if a grant is also received by SA2, following the iSlip rules (McKeown 1999). The reason why this is crucial can be perceived by the following example, in which the arbiters of SA1 and SA2 update their priorities independent of the result of each other.

Assume that input VC#3, that belongs to input port 1, has already allocated an output VC and performs switch allocation using the round-robin arbiters of SA1 and SA2, respectively. Input VC#2, that belongs to input port 0, also owns an output VC of the output port and participate in SA as well. Both input VCs win in SA1 (they belong to different inputs) and advance to SA2, in order to fight for accessing the output port. SA2 arbiter's priority favors input port 0, thus, input VC#2 is granted and priority is updated to point to input port 1. However, the SA1 of input 1 has updated its priority to point to VC#4. As a result, in the next cycle, input VC#4, which is also allocated to an output VC, wins in SA1 and possibly in SA2 thus letting VC#3 loose for two consecutive cycles. Depending on the router's configuration and the traffic pattern, the above situation of VC#3 wining in SA1 but losing in SA2, may be repeated indefinitely. This situation can be avoided by guaranteeing that if an input VC wins in SA1, it will remain the winner input VC until it is granted in SA2 as well. Under this rule, an input VC may need to wait at most $N - 1$ cycles to be granted in SA2.

7.1.7 Output-First Allocation

The order of arbitration in either VA or SA can be changed from input first to output first. In the case of output-first allocation all input VCs forward first their requests to the output arbiters for SA and to the output VC arbiters for VA. In this way, in VA, it is possible that one input VC receives a grant from more than one output VCs. Selecting one of them requires an additional local per-input VC arbitration step. Equivalently, in SA, with output-first arbitration it is possible that two input VCs of the same input to receive simultaneously a grant from the same or a different output. Then, since only one input VC can be served from each input, an additional arbitration step should take place that would resolve the conflict.

Output first allocation has been proven superior in terms of matching quality when compared to input-first allocation (Becker and Dally 2009). However, in terms of hardware implementation input-first allocation is more delay efficient. The reason for this efficiency is that input-first allocation decisions allow the concurrent implementation of the necessary multiplexing. For example, the grants of SA1 can be used directly to multiplex the flit of the winning VC in parallel to SA2 arbitration. Thus, when SA2 finishes, the data to the output multiplexer are ready waiting for the corresponding grants. On the contrary, in output-first allocation, the input and the output multiplexers should wait both SA2 and SA1 to complete before switching the flits from input VCs to the output. In the pipelined implementations those differences are partially alleviated, while still observing that input-first allocation provides faster circuits.

7.2 Many-to-Many Connections Using an Unrolled Datapath: A Complete VC-Based Router

The design of a generic VC-based router that supports many-to-many connections using a fully unrolled switching datapath, i.e., a crossbar, can be easily derived as an extension to the already presented many-to-one switching module. The baseline datapath of the generic VC-based router is shown in Fig. 7.8. Similarly to the many-to-one case, a pipeline register is used at each output, which cuts off the timing path of the link from the paths of the router.

The presented router is just an unrolled version of the baseline switching module shown in Fig. 7.1. Every output is equipped with an output multiplexer, while it includes also V credit counters used for the link-level flow control and the V *outVCAvailable* flags that are used during VC allocation. The VA and SA stages operate in a separable manner taking local per-input or per-input VC and global per output or per output VC decisions that guide the assignment of input to output VCs and the allocation of the output ports of the router on cycle-by-cycle basis.

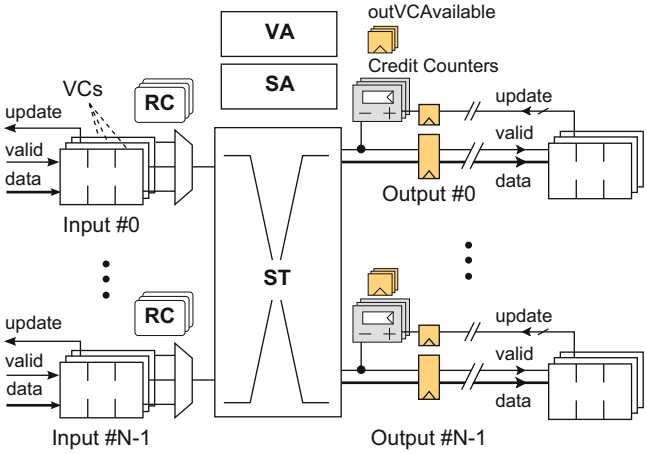

Fig. 7.8 The organization of a VC-based router connecting in parallel multiple inputs to multiple outputs that each own supports many VCs. Routing computation (*RC*) is responsible for selecting an output port for each input VC, while VC allocation (*VA*) and switch allocation (*SA*) handle the allocation of the output VCs and the output ports to the requesting input VCs. The per-output multiplexers of the crossbar implement the actual transfer of flits in the switch traversal stage (*ST*)

7.2.1 Routing Computation

The main difference of the generic many-to-many router versus the simpler many-to-one switching module is the role of routing computation and the selection logic that is involved. In the many-to-many organization every input VC is eligible to connect to the output VC of any output of the router. Therefore, each input VC is equipped with the *outPort*[i] variable that stores the output port that the packet, currently in the ith input VC, needs to follow in order to reach its destination. *outPort*[i] variable is updated after routing computation, which is performed only when the head flit of a packet reaches the frontmost position of the ith input VC buffer. The *outPort* variable is reset to zero once the last flit of the packet, i.e., the tail flit, is granted to leave the corresponding input VC buffer.

The simplest implementation would introduce a routing computation unit per input VC, as shown in Fig. 7.9a. Depending on the complexity of the routing computation unit this choice may not be the best one. Taking into account that at most one new head flit will arrive per clock cycle at each input then routing computation is needed only for one packet. Hence, the routing computation unit can be shared between all input VCs, as depicted in Fig. 7.9b. Although a shared routing computation unit seems like an area saver it does not represent the best choice in area-delay sense. The delay overhead of the multiplexer and the arbitration unit (just a simple fixed priority arbiter) may lead to increased implementation area when the design is synthesized under strict delay constraints. In the rest of this book we assume that each input VC is equipped with its own routing computation unit.

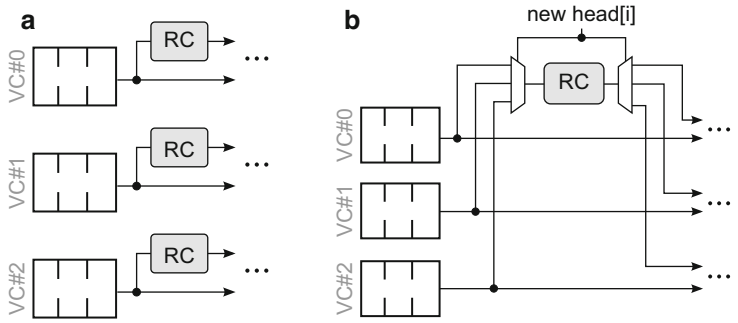

Fig. 7.9 (**a**) Each input VC can have its own RC unit for performing routing computation independent of the rest or (**b**) one RC unit can be shared by all input VCs of the same input

7.2.2 Requests to VC the Allocator

A head flit of a packet that has not been yet assigned an output VC to its destined output port should send a request to the VC allocator. Depending on the limitations imposed by the routing algorithm or some other upper level protocol, the packet can request from one to many output VCs. Besides the requested VC ($reqVC[i]$), each input VC should also send its destination output port ($reqPort[i]$), which will be used for the per-output arbitration stage of VA, as depicted in Fig. 7.10.

Similar to the many-to-one connection, the VC allocator returns per input VC the selected output VC derived by the local VC arbitration step and a flag that denotes if this input-output VC pair has been matched or not. Please keep in mind that since VC allocation is performed in parallel across input and output VCs many input VCs can be matched in parallel as long as they refer to different output VCs (or output VCs that belong to different output ports).

7.2.3 Requests to the Switch Allocator

The packets that have been successfully assigned to an output VC can participate in switch allocation. The output requests for the flits of each input VC are already stored in the *outPort* variable. The head flits do not use the stored variable but the one available via the bypass path of the *outPort* register shown in Fig. 7.11. This is necessary in the single-cycle router implementation described in this chapter. The $outPort[i]$ lines per input VC are actually driven to the switch allocator after being qualified by three conditions:

The request corresponds to a valid flit: The $outPort[i]$ variable that was set by the head flit of a packet may contain active output requests even if the buffer of

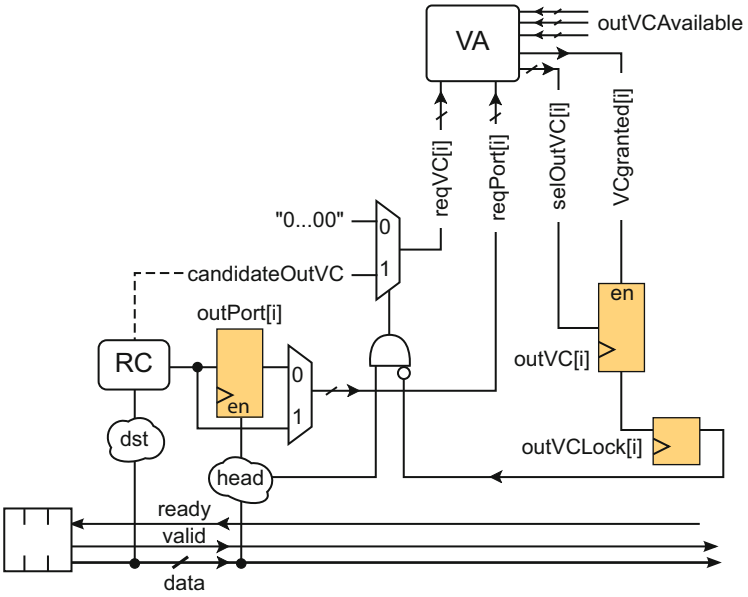

Fig. 7.10 The request generation and grant handling logic for the output VC allocation process. Each input VC forwards to the VA unit the candidate output VCs and the selected output port (as computed by the RC unit) and receives the id of the selected output VC along with a flag that reveals if the allocation process was successful or not

the ith input VC is empty in the current cycle. This can occur since flits are not guaranteed to arrive contiguously for a single input VC. Therefore, masking the requests with the $valid[i]$ bit solves this issue.

The packet has allocated an output VC: The second condition dictates that the input VC has been assigned an output VC. This is resolved by masking the $outVCLock[i]$ variable with the bits of the $outPort[i]$ bit vector (see right side of Fig. 7.11). Similar to the many to one connection, a head flit is allowed to use the VA result directly at the same cycle using $selOutVC[i]$ instead of $outVC[i]$ via a bypass multiplexer.

The output VC has enough credits: An input VC can send a request to the switch allocator if the selected output VC has at least one credit available. This checking requires first the selection of the appropriate ready signal. Therefore, the ith input VC checks if $ready[outPort[i]][outVC[i]] = 1$. Thus, each input VC should select from the $N \times V$ ready bits the one that corresponds to its destined output port and allocated output VC. This selection is done by the multiplexers shown in Fig. 7.11. In the first selection stage the ready bits that belong to the selected output are distinguished from the rest. In the second selection stage the ready bit that belongs to the assigned output VC is selected and it is finally masked with the $outPort[i]$ requests.

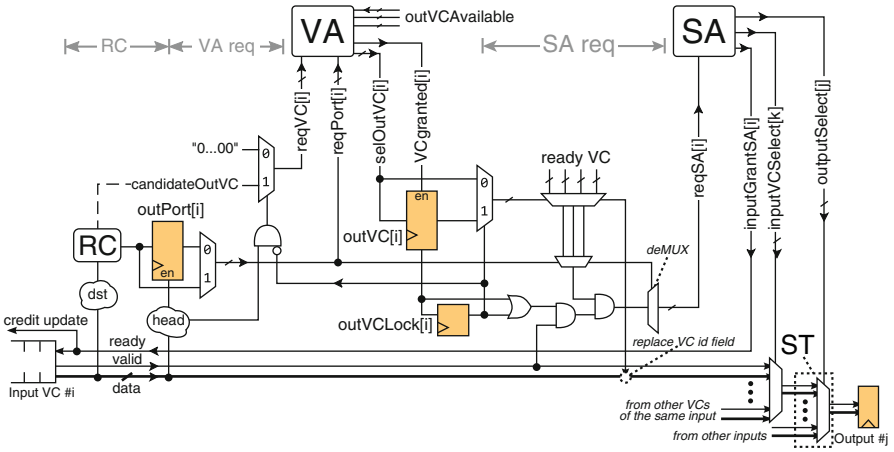

Fig. 7.11 The complete request generation and grant handling logic for a VC-based router that supports many-to-many input-output connections. The requests to the switch allocator are driven by the internal variables of each input VC that guarantee the allocation of an output VC and qualified by the ready signals of the per-output VC credit counters

The requests seen by the switch allocator can be graphically represented in matrix form: When $reqSA[i][j] = 1$ means that input VC i is requesting output port j. The ith input VC belongs to the kth input where $k = i \div V$. The example of Fig. 7.12a shows the output requests for a 3×3 router that hosts two VCs per physical channel. Please keep in mind that every active request of this matrix has already guaranteed buffer availability to the destined output VC.

First of all, a valid match to the request matrix should contain at most 1 bit asserted per row and per column meaning that an input VC cannot be assigned to more than one output port and an output port cannot be assigned to more than one input VCs respectively. If this was the only condition imposed by the switch allocator, then more than one VC of the same input could receive a grant in the same cycle. Satisfying multiple grants to the same input means that each input VC sees a private input port of the crossbar. In the baseline case, all input VCs of the same input share a common input port of the crossbar via a data multiplexer per input. Thus, the switch allocator should grant at most one input VC from the same input. Therefore, a valid match to the request matrix should contain at most one asserted bit to the group of rows that belong to the same input with index $i \div V$, where $i = 0 \ldots N \times V - 1$. The match shown in Fig. 7.12a satisfies all the required conditions. The requests of all input VCs and the corresponding grants are illustrated in Fig. 7.12b using an equivalent bipartite graph representation.

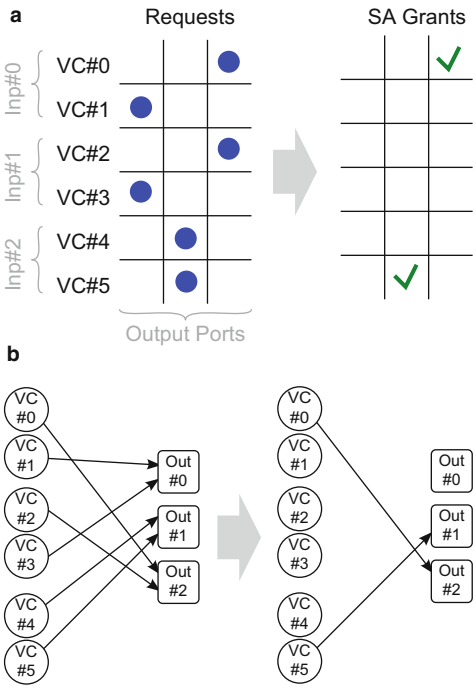

Fig. 7.12 (**a**) An example of the request and grant matrix of switch allocation, and (**b**) its equivalent bipartite graph representation. Although every input VC can request any output, only one VC per input can be granted

7.2.4 Gathering Grants and Moving to the Output

The switch allocator's decisions are distributed in the same cycle to the input controllers and the crossbar. Each input VC receives a flag bit showing if it has won access to the selected output port or not. Once granted, the corresponding input VC dequeues its flit from the input VC buffers, and sends a credit update backwards, informing that a buffer slot is emptied. The multiplexer that selects only one input VC per input is also driven by the switch allocator's output, so that only the selected input VC to reach the crossbar. On dequeue, the flit updates its VC id field by using the id stored in the local $outVC[i]$ variable (or the one just returning from the VC allocator). If a tail flit is preparing to leave the ith input VC, then it should de-allocate all resources reserved per packet at the input controller, such as the state variables $outPort[i]$, $outVCLock[i]$ and $outVC[i]$.

The crossbar knows how to handle the incoming flits from all input controllers since the switch allocator has transferred to the crossbar the switching configuration of the current cycle that describes the connections between inputs and outputs. As the flit traverses the crossbar and moves to the output pipeline register, its VC ID field is used to decrement appropriately $creditCounter[VCid]$. In the next cycle, the flit is forwarded to the link where it cannot be stopped, since credit availability has been checked before it was allowed to participate in switch allocation. In the case

that the outgoing flit is a tail flit the output controller should reset the corresponding output VC availability flags to available since the packet that used this output VC has left the current router.

7.2.5 *The Internal Organization of the VC Allocator for a VC-Based Router*

The VC allocator should be able to allocate in parallel the input VCs to the output VCs of the router. In this case, the router consists of many outputs that each one services a number of VCs. Therefore, the input VC should not only inform the VC allocator on the candidate output VCs but it should also declare the output that these candidate VCs belong to. Therefore, the VC allocator receives a pair of vectors from each input VC: A vector that indexes the requested output port *reqPort*[*i*], and the *reqVC*[*i*] that indexes the requested output VC(s). The first step of VC allocation is to filter out from the requested output VCs those that are not available. Each input VC sees $N \times V$ output VC availability flags. From those flags selects the ones that refer to the destined output port, that are later masked with the candidate output VCs of each input VC.

The allocation process evolves in two steps, according to the organization depicted in Fig. 7.13. In the first step (VA1) each input VC selects one of the

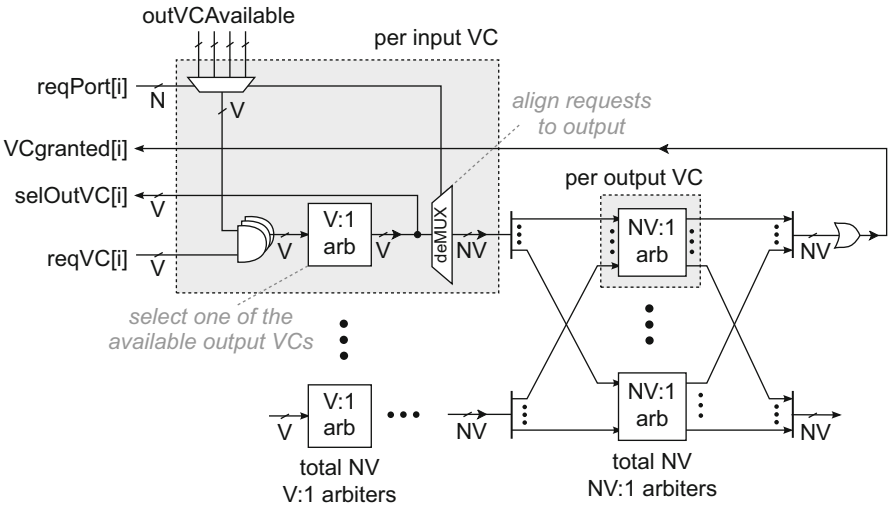

Fig. 7.13 The VC allocator of a complete VC-based router. The requested output VCs (*reqVC*) are masked with their corresponding *outVCavailable* flags and a single output VC is selected per input VC after VA1 arbitration. During VA2, each available output VC is assigned to at most one requesting input VC

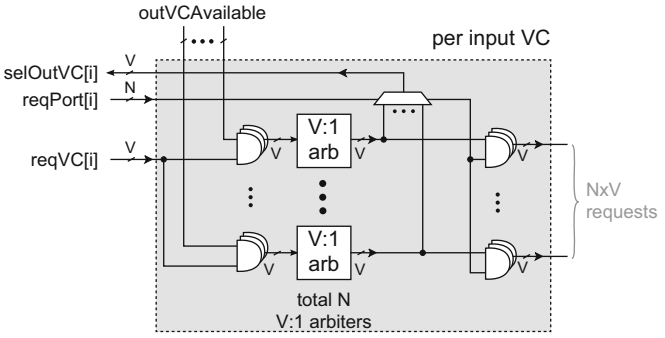

Fig. 7.14 An alternative organization of VA1 stage of the VC allocator that offers delay benefits, under small area overhead. It replaces a mux, one arbiter and a demux with N arbiters that run in parallel and prepare the output VC requests of each input VC in a form that fits directly the connections of the arbiters in the VA2 stage

available output VCs and then in VA2 each output VC selects at most one input VC. The input VCs are informed by the arbiters of VA2 if their request was finally accepted.

Faster Organization of the VA1 Stage

Implementation results prove that the (de)multiplexing logic at VA1 has a non trivial contribution to the overall delay of VC allocation. A simple microarchitectural change can completely eliminate this logic and speedup significantly VC allocation. The new fast organization of VA1 is shown in Fig. 7.14.

First all the output VC availability flags of all outputs are masked with the *reqVC* vector of each input VC without any pre-selection step. The resulting availability vectors, e.g., one for each output, are independently arbitrated by $V : 1$ arbiters selecting one available VC for each output. From the selected output VCs (one available VC per output), each input VC needs only one of them; the one that belongs to the destined output port. Selecting one does not require any multiplexing but just an additional masking operation with the output port request ($outPort[i]$) of the ith input VC. The selected output VC in all outputs will become zero except the one that matches the destination output port. Therefore, after this last step, the output VC request of an input VC is ready and aligned per output as needed by the output VC arbiters of the second stage. Thus additional demultiplexing/alignment logic is not needed and significant delay is saved. The cost of this method is that it replaces a mux (*outVCAvailable* multiplexer of Fig. 7.13), one arbiter and a demux (Fig. 7.13), with N arbiters that run in parallel and offer faster implementation.

Please notice also that since the $outPort[i]$ request bits are used only after the $V : 1$ arbitration step then routing computation can be overlapped in time with the

per-input VC arbitration and additional delay can saved. Certainly, this time overlap is only enabled if the generation of the *candidateOutVC* for each input VC is not strictly dependent on the routing algorithm.

7.2.6 The Internal Organization of the Switch Allocator for a VC-Based Router

Switch allocation involves both a per-input and a per output arbitration step. Differently from the case of a many-to-one connection, in this case, the switch allocator receives a request vector from each input VC that points to the requested output port. When an input VC has an active output port request (at least one bit of the request vector is asserted) it is eligible to participate in SA1. The input VC that is selected by SA1 carries its request vector to the next arbitration step (SA2). This is done via the multiplexer shown per input in Fig. 7.15. Each arbiter of SA2 independently from the rest selects which input to grant, based on its local priority status. The grant signals are distributed to the crossbar setting up the appropriate input-output connections and to the inputs. Once this information reaches the inputs it is combined with the decision of SA1 and prepares the winning input VC for sending a new flit to the crossbar. An example of the grants generation process by the switch allocator, including both arbitration stages' results, is presented in Fig. 7.16.

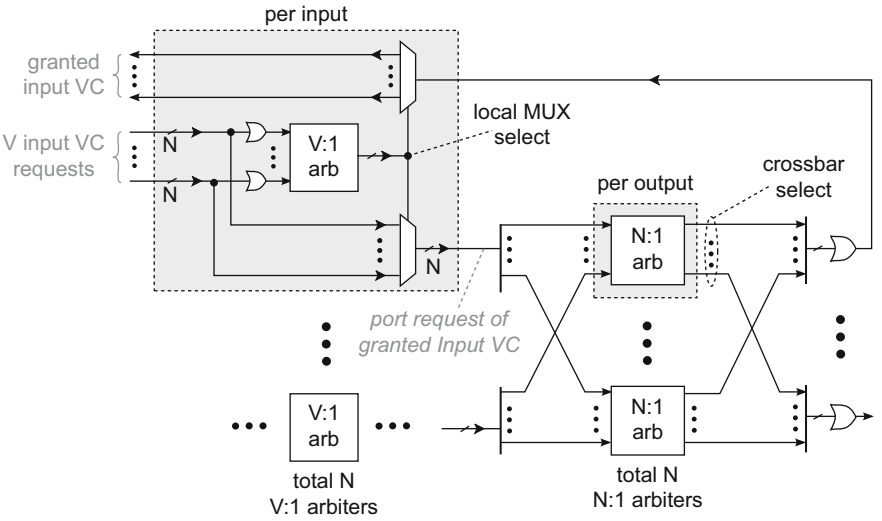

Fig. 7.15 The switch Allocator for a complete VC-based router that supports many input/output parallel connections. The SA1 stage promotes one VC per input that fights with the rest inputs in SA2 to get access to the requested output port

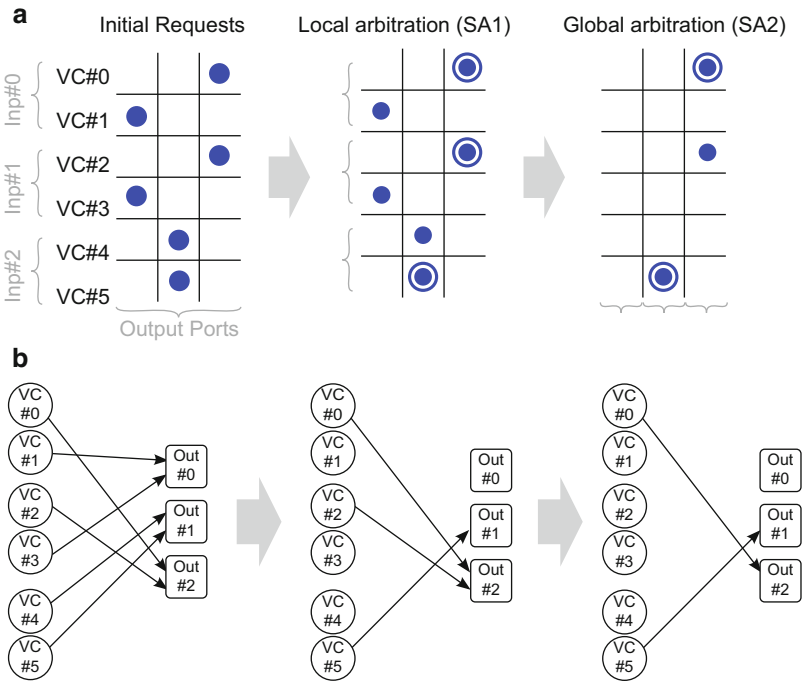

Fig. 7.16 An example of the requests seen by the switch allocator and the derived grants in SA1 and SA2 following (**a**) a matrix representation and (**b**) a bipartite graph model between input VCs and outputs. Local arbitration (SA1) allows the request of at most one input VC to qualify per input, while global arbitration (SA2) selects one input to access a specific output port

Please notice that, like in the many-to-one connection, the arbiters of the SA1 stage should update their priority only once their selected input VC is also granted at the SA2 stage. Even though the arbiters operate independently, their eventual outcomes in switch allocation are very much dependent each one affecting each other port separately, as well as the aggregate matching quality of the router (Mukherjee et al. 2002). In order to improve the efficiency of such separable switch allocators that rely on independent per-input and per-output arbitration steps we have two generic options. We can either try to "desynchronize" their bindings, so that each input (output) requests a different output (input) on every new scheduling cycle, or to employ multiple scheduling iterations until a good match with many input-output pairs is constructed. Desynchronization is hard to achieve in the context of NoCs since it requires the addition of many independent queues per input equal to the number of output ports (McKeown 1999). This is either prohibitive, or it may lead to very shallow buffers that will destroy throughput. On the other hand the execution of multiple scheduling iterations for converging to one allocation remains an unexplored alternative for NoCs mostly because it prolongs the scheduling time; SA evolves in multiple iterations, where in each iteration the set of already

matched input-output pairs is augmented with new matchings. Pipelining between iterations is a viable alternative (Gupta and McKeown 1999), however a more scalable solution is desirable.

Instead of letting a different input VC to connect to an output in each cycle, other allocation strategies try to prolong the duration of an input and output match by letting whole flows of packets to pass before changing the connection (Michelogiannakis et al. 2011; Ma et al. 2012). This approach allows for full output utilization for many cycles but may create starvation phenomena. The main implementation strategy for this exhaustive-like scheduling approach, involves some form of weighted arbitration that is biased in favor of certain input-output pairs that correspond to heavily backlogged flows (Ramabhadran and Pasquale 2003; Abts and Weisser 2007).

Switch Allocation in the Case of Adaptive Routing

The presented organization of the SA unit assumed that each input VC will never ask for more than one output port. In the case of adaptive routing this may not be the case and the adaptive routing algorithm may allow each input VC to select more than one possible output ports. In this case, SA1 and SA2 do not suffice for completing the switch allocation process and an additional selection step is needed. The additional selection steps can be done either at the beginning of SA letting each input VC to select one candidate output port or at the end. The selection unit (not shown in Fig. 7.11) either picks randomly a destination or decides after sensing the state of the network (Ascia et al. 2008) and taking into account other network-level criteria such as load balancing the traffic throughout the network or offering quality of service guarantees.

7.3 VA and SA Built with Centralized Allocators

Besides separable allocation strategies that implement the allocation process separately per input (or per input VC) and per output (or per output VC), VA and SA can be built in a centralized manner that solves allocation at once by actually merging input and output arbitration phases in one merged step.

A centralized allocator of N requesters and N resources receives the corresponding requests in a matrix form. Each row of the matrix corresponds to the requests of one requester that can ask for multiple resources. Equivalently, each column of the request matrix corresponds to one resource that can accept multiple requests. A valid schedule should contain at most a single 1 per row and per column guaranteeing in this way a unique requester-resource connection. A centralized allocator does not examine the requests independently per row and per column as done by the separable allocators but solves the problem concurrently for many requester-resource pairs. The request-resource pairs that can be matched at once

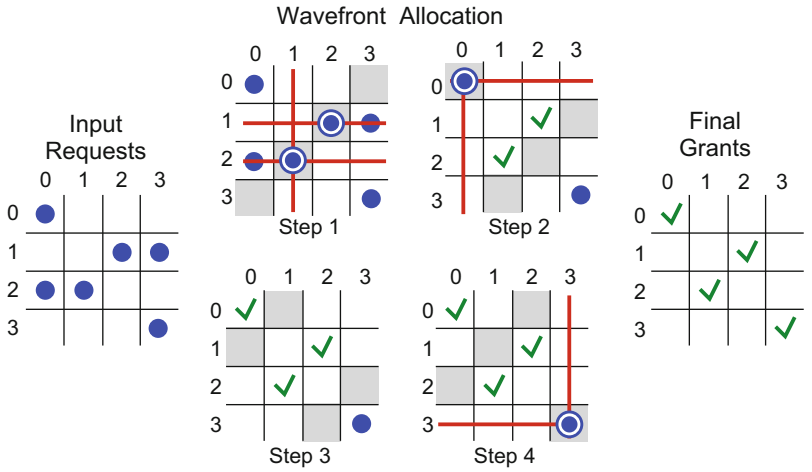

Fig. 7.17 An example of the operation of a centralized allocator. All requests are examined starting from the main diagonal of the request matrix. The active requests of a diagonal do not cause any conflicts and they can be granted at once. A grant given to a certain request-resource pair directly erases all the remaining requests of the same row and column

without any further checking are the requests that belong to the diagonals of the matrix. Every element that belongs to a matrix diagonal corresponds to a different request-resource pair and can be granted without causing any conflict. Once a request of the ith row and jth column is granted then all the requests of the ith row and the jth column should be nullified before moving to the next diagonal of the matrix. An example, of this diagonal-based scheduling mechanism is shown in Fig. 7.17. The allocation process evolves in 4 steps (equal to the number of diagonals) and at each step the non-conflicting requests are granted. The most efficient centralized allocator is the wavefront arbiter (Tamir and Chi 1993; Hurt et al. 1999; Becker 2012b).

Using a centralized allocator, such as the wavefront arbiter, we can design VC and switch allocators. A VC allocator is built around a $NV \times NV$ centralized allocator that receives the requests of all input VCs in parallel. Figure 7.18 illustrates this organization. The rows of the centralized allocator correspond to input VCs and the columns to output VCs, respectively. Each input VC may request many output VCs of the same output, i.e., it asserts a request to multiple columns of the centralized allocator. When allocation finishes the $N \times V$ grant signals coming from all output VCs are gathered per-input VC, while the OR function at each input VC just detects if at least one output VC granted the corresponding input VC.

Equivalently, a switch allocator can be built using a $N \times N$ centralized allocator, as illustrated in Fig. 7.19. The rows of the centralized allocator correspond to the inputs of the routers and the columns to the outputs. Since each input hosts many VCs, a row of the request matrix can have many active requests that correspond to the output requests of the input VCs. When the centralized allocator finishes, each

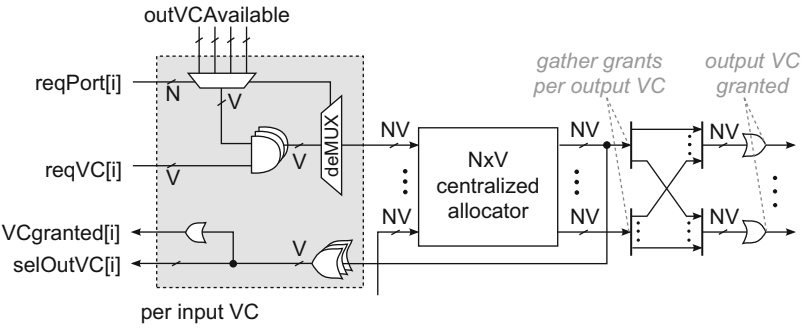

Fig. 7.18 The organization of a VC allocator that uses a $NV \times NV$ centralized allocator

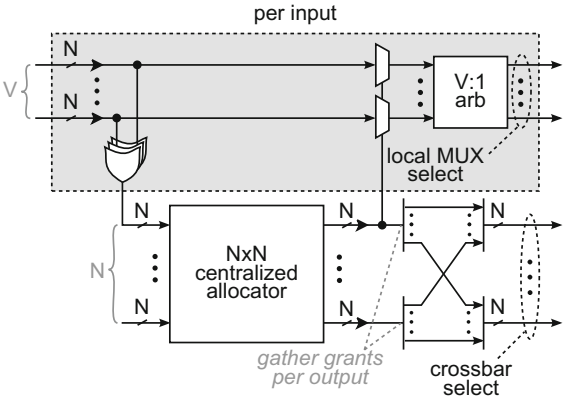

Fig. 7.19 The organization of a switch allocator that uses a $N \times N$ centralized allocator

output is matched to at most one input. The only thing that remains to be selected is the VC per input that actually won. For each input, the VCs that requested the matched output are kept alive through a masking process. Since these can be more than one input VCs that requested the selected output port, a final $V : 1$ arbiter is used to select which input VC will finally send a flit to the selected output port.

7.4 Take-Away Points

The operation of a VC-based router includes multiple steps that should be executed in the correct order in order to allow the packets placed at the input VCs to move to their selected output port. The execution of each step such as routing computation, VC allocation and switch allocation is supported by additional per-input VC and per-output VC state variables that guide request generation and grant handling for the allocation steps, and implement the virtual-channel flow control mechanism of the input and output links. Input VCs allocate an output VC to their selected output

port (decided by the routing computation logic) and then after checking the available credits of their selected output VC proceed to switch allocation and traversal that guides them to their destined output port. The organization of per-input VC and per-output VC logic as well as the internal organization of the allocators has been presented in detail and in a ready-to-use manner.