Des Autom Embed Syst
DOI 10.1007/s10617-011-9075-5

Application driven network-on-chip architecture exploration & refinement for a complex SoC

Jean-Jacques Lecler · Gilles Baillieu

Received: 20 May 2010 / Accepted: 4 March 2011

Abstract This article presents an overview of the design process of an interconnection network, using the technology proposed by Arteris. Section 2 summarizes the various features a NoC is required to implement to be integrated in modern SoCs. Section 3 describes the proposed top-down approach, based on the progressive refinement of the NoC description, from its functional specification (Sect. 4) to its verification (Sect. 8). The approach is illustrated by a typical use-case of a NoC embedded in a hand-held gaming device. The methodology relies on the definition of the performance behavior and expectation (Sect. 5), which can be early and efficiently simulated against various NoC architectures. The system architect is then able to identify bottle-necks and converge towards the NoC implementation fulfilling the requirements of the target application (Sect. 6).

Keywords Multimedia system-on-chip (SoC) · Network-on-chip (NoC) · Memory-mapped transaction interconnect · Dynamic memory scheduling · Quality-of-service (QoS) · Performance verification · Architecture exploration · SystemC transaction level modeling (TLM)

1 On chip interconnect fundamentals

On a SoC, intellectual property blocks (IPs) communicate over an interconnect system using *transactions*. A transaction is the operation of reading or writing some bytes of data at some addresses in a memory space. Transactions are composed of a *request* which goes from an *initiator* to a *target* and contains at least the address and data in the case of a write operation; and a *response* flowing back from the *target* to the *initiator*, with some status and data in case of read operation. The physical interface between IPs and the interconnect system are *sockets*, usually synchronous, with a *master* side and a *slave* side. A number of different

J.-J. Lecler (✉) · G. Baillieu
Arteris, Parc Ariane, Immeuble Mercure, Boulevard des Chênes, 78284 Guyancourt Cedex, France
e-mail: jean-jacques.lecler@arteris.com

G. Baillieu
e-mail: gilles.baillieu@arteris.com

Published online: 07 April 2011

socket protocols have been defined by the SoC industry (OCP, APB, AHB, AXI, STBus, DTL, etc.). Sometimes, transactions have to be converted between the initiator socket and the target socket of an interconnect, for instance if both socket protocols are not identical. Such conversions must preserve the byte transfer semantics, i.e. the functionality of the transaction.

Interconnect systems have gradually evolved from simple busses to more scalable networks-on-chip (NoC). It could be tempting to consider a NoC as a down-scaled TCP/IP or ATM network; it is not so. TCP/IP and ATM are connection based networks, where connections are dynamically opened and closed between producer and consumer ports. History shows that small systems are transaction-based, while large systems are connection-based. Attempts to design small connection-based systems (INMOS transputers), or large transaction-based systems (Distributed Shared Memory PC clustering), have up to now failed. A time may come when SoCs are so huge that a connection-based network is required between transaction-based clusters, but it is not the case yet.

2 Trends

Beginning a micro-electronic article with a reference to the Moore law is a common place. Yet, the growth of the number of functions which can be integrated on a single die is truly, for designers, the source of many troubles. Designers troubles induce costs, risks and delays which are certainly among the most hated words of companies' decision makers.

2.1 IP reuse

First, the massive reuse of existing IP blocks is an absolute necessity. The shapes those to-be-reused IPs can take is quite variable: hard (layout-ready) or soft, in house or third-party, simple (USB) or complex (complete H.264 decompression engine). The NoC interconnect has to convey transactions between sockets of different protocols (OCP, AXI, proprietary, etc.) each protocol being highly parameterized.

This variety has major impacts from a performance perspective. The IPs may have to be clocked at different frequencies and their sockets may present different data widths, leading to a wide range of peak throughputs which must be adapted between initiators and targets. Initiators have different traffic patterns, in terms of transaction lengths, address alignments and regularity. Their reaction and resistance to latency or transient back-pressure differ as well. Also, meeting the quality of service (QoS) required by each initiator despite the presence of the other traffic becomes more challenging as the number of IPs increases.

2.2 Power management

In most SoCs, aggressive power management is becoming a requirement, not only for hand-held devices where it is critical for the system autonomy, but also for desktop or automotive applications as well, where it has direct consequences on the chip packaging and system cooling. This power management includes local clock gating on groups of several flip-flops, global clock gating saving the energy dissipated in an idle block and active power supply extinction of some IPs or subsystems. As the NoC crosses power domains, it must collaborate with IPs and the SoC power controller, so as to guarantee the to-be-shutdown domain is empty of transactions, or request the power supply restoration.

Dynamic voltage and frequency scaling (DVFS) is also becoming a common practice, again as the NoC is likely to cross voltage domains, the places to insert required level-shifters must be cleanly identified. The sensitivity to power consumption also calls for a tight control of the gate and wire count.

2.3 Physical constraints and timing closure

The size of most IPs shrink with the CMOS process, the NoC does not: it is by definition spread across the die, making the back-end phase a difficult challenge. On the one hand, depending on clock frequencies, pipeline stages may have to be inserted, just to leave margin for spatial propagation delay. On the other hand, long wires are an expensive resource; the ability to locally adjust the number of wires in the communication bundles to back-end constraints and performance requirements is a key.

2.4 Software observability and security

If the hardware complexity rises, the software complexity explodes, with consequences on the hardware. Many different software layers collaborate on a SoC: Firmware, operating system, drivers, middleware, applications, sensitive data subsystems, etc., all distributed on the different IPs. The interconnect has to provide mechanisms to grant or deny the right for transactions to reach certain targets, depending on initiators and in-band security qualifiers.

For software debugging reasons, the NoC must be observable to a certain extent. Error detection and reporting is a minimum, but transaction probing and sometimes on chip performance measurements are required as well. As traffic is more and more distributed, observation collection becomes more expensive. The whole traffic on the NoC cannot be observed in real-time, and observation must focus on strategic traffic such as DRAM accesses. In the end the cost versus observability trade-offs must be left in the hands of the designer.

2.5 Verification

In general, IP quality verification is a well-known issue. Because of its wide configuration space (thousands of parameters is a good order of magnitude) verifying the implementation of a particular NoC is a much more complex problem. Checking the conformance of the sockets to their respective protocol is not enough: the correct translation of transactions between the initiator side and the target side must be verified as well. Depending on the initiator and target socket configurations, transactions may have to be split, realigned, unwrapped, etc. On the top of this functional verification, the performance of the NoC also has to be validated.

2.6 EDA design flow

The NoC development overlaps the whole SoC design project lifetime: From early SoC architecture phase, to final place and route, passing through performance qualification, RTL design, IP integration, verification, software integration. One must pay special attention that a decision taken during an early phase, such as design architecture, does not jeopardize the feasibility of a later phase, such as back-end. Moreover, it is not unusual that marketing requirements for a given SoC change while the chip is being designed, adding or removing some IPs, and thus impacting the NoC specification. As the complexity and cost of brand

new architecture increases, organizing projects around reusable platforms becomes more efficient. Small variations of the platform or derivatives may then be brought to the market in shorter cycles. The interconnect system design methodology must make sure that those late specification changes are smoothly integrated in the design and verified.

3 Approach

The main NoC designer's challenge is to navigate between the flexibility required to only pay for useful resources and the risk of getting oneself lost, entangled in the middle of the numerous caveats. Designers should make sure to stay away from three major pitfalls:

- Focusing on one subject and forgeting about other aspects of the design. Typically concentrate on performance, and neglect design cost and timing closure.
- The opposite attitude, trying to tackle all the issues at once, is not a solution either.
- Assuming a strategy was valid for a previous design is not proof it is still valid for another. A typical example is the case where most initiators are able to sustain the dynamic memory peak bandwidth. When the very same set of initiators is integrated as a subsystem of the next generation SoC, sharing the DRAM bandwidth with others, the NoC architecture may have to be entirely rethought.

Arteris proposes a layered, top-down approach.

- In a first phase, we define the *functional specification* of the NoC: What are the different sockets? The transaction protocols? Subsets of the protocols used? What is the memory map? The objective is to have clear statements about the possible transactions initiators can request and targets can accept. Remember some initiators may emit unaligned or wrapping transactions towards targets which do not support them. This implies conversions and it is important to know whether that underlying logic must be implemented or not. The specification phase leaves completely aside *how* the functionality is implemented.
- The *performance specification* is the definition of the quality of service expected from the NoC. One should note that performance of an interconnect is not in anyway defined by its internal resources (number of pipeline stages, number of possible pending transactions, etc.), but by the fact that a system using this NoC behaves correctly or not: Will a displayed video image freeze, the loud speaker click, or the embedded web-browser be too slow? At this stage we only define the objectives. Again, the actual implementation is left to a further step. This phase is also a good time to collect data on the actual behavior of the IP blocks around the NoC: What are the transaction patterns? How do they react to transient back pressure? To latency? This behavior of the system around the interconnect is defined in *scenarios*.
- In the *architecture* phase, we now allocate hardware resources in the NoC, defining how transaction flows are merged and scattered, tuning wire bundle serialization, buffer size, arbitration schemes, pipeline stages location, clock and power crossings location, etc. The architecture may be simulated against the previously defined scenarios in order to determine, on one hand, if the scenario performance criteria are met, and on the other hand, what are the saturated or under-used resources in the NoC.
- In the *NoC synthesis* phase, the NoC is translated into hardware blocks. The translation is automated, yet some minor but crucial details may be fine-tuned, such as the presence and/or reset values of configuration registers. At the end of this phase, the Register Transfer Level (RTL) description of the NoC can be exported together with synthesis scripts, ready to be integrated, simulated and synthesized.

- In the *verification* phase, the RTL description is checked against its functional and performance specifications. For the functional verification, a test bench and stimuli are generated out of the specification. The test bench uses third-party protocol checkers and ensures that transactions are correctly translated. Protocol coverage is measured to ensure the stimuli have stressed the interconnect in a number of predefined corner cases. For performance verification, the scenarios are played against the RTL description of the NoC to verify their performance criteria are met.

This article illustrates those phases on an example, focusing on the scenario and architecture phases.

4 Functional specification

The proposed case is the central SoC for an hand-held gaming device. It is DRAM centric: In functional mode, the vast majority of the traffic is geared towards a dynamic memory.

An overview of the SoC is presented Fig. 1. Arrows represent the sockets, pointing from the master to the slave. Orange ones are the sockets which will be activated in Sect. 5. It comprises the following subsystems:

- The DRAM is a DDR2 device with a capacity of 1 GB. The data bus is 64 bit wide running at 666 MHz. As the chosen memory controller operates at half the DDR frequency, the socket to the main interconnect is actually 128 bit wide (16 Bytes) running at 333 MHz. The memory device has 8 banks. The peak bandwidth of such a device is then 5.333 GB/s.
- The CPU subsystem runs the operating system and the applications. As modern games require significant processing power, this CPU subsystem is composed of four CPU cores, each of the cores being able to run two threads. Each core have a dedicated level 0 cache, and all four cores share a common level 1 cache. The CPU cluster maintains cache coherency between level 0 caches, but this is not visible from the main NoC. The interface of the CPU subsystem appears as two AXI sockets: One—read-only—for instructions and one read/write for data accesses. Both sockets are 8 bytes wide and run at the DRAM frequency of 333 MHz.
- The display subsystem drives either the embedded LCD screen or an external HDMI port. This subsystem was actually a former stand-alone chip, embedded as a hard-macro IP block. It handles not only the video scanning, but also display pre-processing such as frame de-interleaving, 100 Hz overscanning and contrast enhancing. It uses 4 OCP-based sockets, each being 4 bytes wide: Three are read-only and one is write-only. The subsystem uses a rather slow clock operating at 133 MHz.
- The device includes a 3G modem, enabling network-connected gaming. The modem is actually not part of the main SoC, and is implemented as a separate device. 3G modems require significant data-processing, however, dedicating a memory device to the modem is not economically optimal once pin-count, bill of material and printed-circuit board foot prints are taken into account. The system is thus designed so that the modem can actually access the main DRAM. The interface is dual thread 4 byte wide OCP sockets: One thread for data accesses and the other for modem CPU accesses. Those interfaces are run at half the DRAM frequency (167 MHz).
- The image subsystem implements the 3D mapping engine. It uses four 8 byte wide AXI sockets running at the DRAM frequency of 333 MHz.
- The peripheral subsystem implements the "small" peripherals of the system: SATA for disk accesses, ethernet networking, USB2.0, Bluetooth, IEEE 1394 (FireWire), as well as

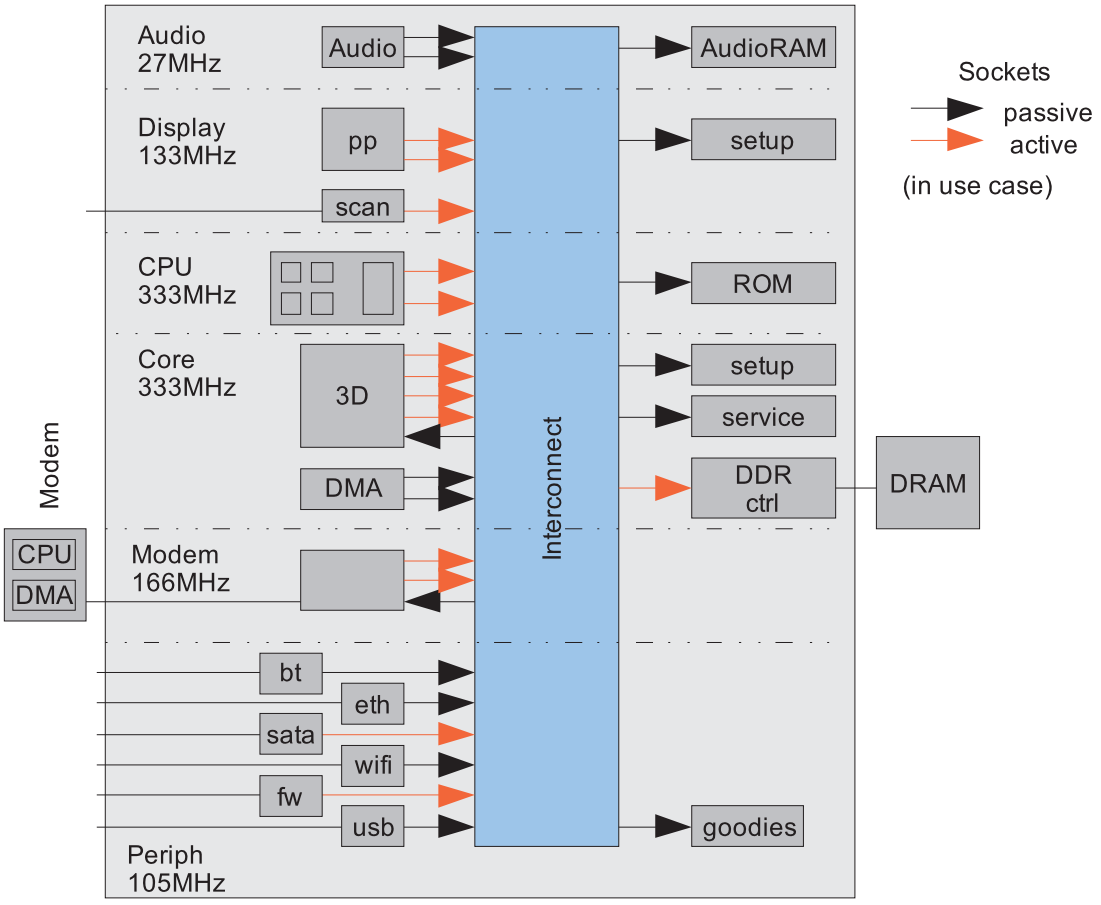

Fig. 1 SoC overview

all the necessary tiny blocks like UARTs, timers, interrupt controllers, etc. All those use various socket protocols (OCP, AHB, AXI) and run at 105 MHz.

All this information can be entered in the specification phase together with more detailed features of each socket, specifically their respective ordering model constraints (tags of OCP, AIDs for AXI) and the memory map and security information.

5 Performance specification

Let's now focus on the performance aspects, and pick up an application use case. Here we will consider a scenario where a network game is playing while a file is downloading in background. What does this mean for the various subsystems?

In the following sections, we present excerpts of a script file used to feed the scenario description in the Arteris tool. The objective is to obtain an executable template of the *outside* of the NoC, both capable of stimulating the interconnect and stating whether or not the system is running with satisfactory performance.

5.1 CPU subsystem

We will assume the game software has been written to take advantage of the multiple CPU cores using software threads. The Level 1 cache uses 32 byte lines. A CPU has a behavior

that when data is not available in a cache (cache-miss) the corresponding thread is stalled until data comes back into the cache. This makes a CPU latency sensitive. Multi-threaded CPUs are precisely designed to partially mask part of this penalty by switching to an alternate thread.

With the following script-based scenario description:

```
Process('Process_CPU0'
,  procedure =
      Choose([
            ( Read(32,ExplicitDequeue)              + Dequeue(32) ) ∧ qData
         ,  ( Read(32,ExplicitDequeue) + Write(32) + Dequeue(32) ) ∧ qData
         ,  ( Read(32,ExplicitDequeue)              + Dequeue(32) ) ∧ qInst
      ])  % 250**MBps
)
```

a CPU does either a data refill (execute a 32 byte read, and wait for data to return) or a data refill with eviction (execute a 32 byte read, then a 32 byte write, and wait for the read data to return) or an instruction refill (execute a 32 byte read, and wait for data to return, but this time on the instruction socket). In this case the total average traffic represents 250 MB/s. Though not described here, it is possible to define the dual threaded aspect of each core: The CPU does not freeze until a cache line refill is pending per thread.

This behavior is replicated for each of the 4 cores, so that the spontaneous throughput of the CPU subsystem represents 1 GB/s.

5.2 Display subsystem

The display subsystem behavior can be depicted as hard real-time. Here, *real-time* signifies that the advancement of the behavior is in sync externally to the interconnect, in this particular case by the video pixel, line and frame rates. If some transactions are stalled, which will at least temporarily happen in the DRAM controller, the subsystem is able to buffer some data in order to recover from the delay. But the amount of buffering is limited, so that if transactions are blocked too often or for too long, the subsystem will have to display black pixels, which is considered unacceptable.

Note that the avoidance of such an overflow condition does *not* translate into a latency requirement at the boundary of the NoC. Let's take an example. The initiator spontaneous behavior is to send a request every 100 ns, and with a maximum latency of 500 ns between each transaction request and response. At time 0, a first transaction A is requested on the socket. For any reason, transaction A request is not accepted by the interconnect until 450 ns, it is then processed and responded at 480 ns. Latency requirement is met for transaction A. Yet, the initiator should have sent another request B at 100 ns, but because request A has not been accepted, it cannot do so until 450 ns. With a 500 ns latency allowance, the interconnect could send back response B at 950 ns, while the initiator actually needs it at 600 ns. In other words, even met, the latency requirement does not prevent the initiator's failure.

This behavior is modeled like this:

```
Process('Process_DispRd0'
, procedure = Read(64)(randomDramAddr) / 320**MBps
     >> Queue('Queue_DispRd0', depth = 1024 , flow = 'display.rd0/I/0')
, efficiency = 1
)
```

Here, the initiator will attempt to regularly request 64 byte read transactions, with an average throughput of 320 MB/s, i.e. a read request every 200 ns. If for any reason the NoC temporarily does not accept requests, the initiator is able to buffer up to 1024 bytes (16 requests) until it stalls. When the NoC accepts requests again, the initiator will transmit the requests which had been stowed in the queue as fast as allowed to, until the queue is empty again. The *efficiency* parameter has no consequence on the initiator behavior, it is a performance level criterion, stating the minimum percentage of time the initiator must not be stalled, to consider its behavior as acceptable. Setting the efficiency to 1.0 states that stalling is not acceptable: Display is a *hard* real-time subsystem

Similar descriptions are set up for the second read-only socket, for the write-only socket, and for the scan socket.

5.3 Modem subsystem

The behavior of the modem subsystem is a combination of the two previous ones. The modem chip contains a single thread CPU and a some DMA engines with hard real-time requirements. It is preferable to provide a maximum independence between the traffic and from a performance perspective using two different sockets is the optimal solution. The OCP protocol allows the possibility of multiplexing multiple logical independent sockets on the same wires using the multi-threads non-blocking (MTNB) semantics.

This "virtual channel" mechanism is designed to share most of physical wires between two or more logical sockets. This reduces the number of wires carrying the socket signals at the expense of some gates (the inter-thread multiplexing and arbitration logic) a significant timing closure penalty (arbitration logic is added on the top of the already critical flow control signals) and a performance impact (only one word can be exchanged per cycle, instead of two on different sockets). Since the IP block is usually physically located near the NoC the socket wires are short. This makes the MTNB socket not worth it in light of the performance and timing closure impacts. The situation is completely different *between* the chips, where pin and pads are extremely expensive resources: Implementing virtual channel logic at that level is definitely a good option.

In our example, the inter-chip logic presents a two threaded OCP socket to the interconnect.

The modem CPU model may look like:

```
Process('Process_Modem_CPU'
, procedure = ( Read(32,ExplicitDequeue) + Write(32) + Dequeue(32) ) %
100**MBps
  ^ Queue('Queue\_Modem\_CPU', depth = 64 , flow = 'modem/cpu/0')
, efficiency = .7
)
```

Here the modem CPU is considered to be single-threaded with a 100 MB/s spontaneous throughput. For the modem to operate correctly its CPU must be at least 70% efficient, i.e., it must not be stalled more than 30% of its time waiting for cache refills.

The modem also contains several hard real-time traffic flows, this time in sync with the radio frames:

```
Process('Process_Modem_Data'
, procedure = Choose([Read(64),Write(64)]) % 300**MBps
  >> Queue('Queue\_Modem\_Data', depth = 1024 , flow = 'modem/data/0')
, efficiency = 1
)
```

The data-processing part of the modem randomly does a 64 byte read or write at 300 MB/s.

5.4 Image subsystem

As the display subsystem, the image subsystem is synchronized to the video frames: It is a real-time subsystem. However, two major differences have to be taken into account.

First, the amount of buffering present in a display subsystem is of the order of magnitude of a video *line*, around 1 KB, the display subsystem can thus overcome transient interconnect unavailability of some tens of microseconds. An imaging subsystem operates several *frames* in advance, i.e. around 20 to 50 milliseconds, this allows a much higher tolerance to stalling.

In addition, this particular imaging system is able to adjust its quality. When its timing compared to the display time stamp is too low, it reduces its quality (typically throwing away least significant polygons) in order to recover its timing. We can thus consider the traffic generated by this IP as *soft* real-time.

```
Process('Process_Image'
, procedure = Choose({
    Choose([Read(256) , Write(256)]) >> qImageShade   : 800
  , Choose([Read(256) , Write(256)]) >> qImageTexture : 500
  , Choose([Read(256) , Write(256)]) >> qImageMotion  : 350
  , Choose([Read(256) , Write(256)]) >> qImageRaster  : 650
  })(randomDramAddr) % 2.3**GBps
, efficiency = .8
)
```

The 4 queues (qImageShade, qImageTexture, qImageMotion and qImageRaster) have been previously defined as 2 KB deep, connected to their respective sockets. Note that those queues are just a construction to model the reaction of the initiator to back pressure. They do not necessarily represent physical memory present in the initiator; in any case they are not part of the interconnect.

Here, we consider a single process, randomly requesting 256-byte read or write transactions on different sockets, with a weighted probability, for a total traffic of 2.3 GB/s. The rendering is considered smooth enough if the process obtains 80% efficiency, around 1.85 GB/s.

5.5 Background file download

As stated above, a file is downloaded in background, between the SATA port and the ethernet IPs. Those are considered as *best effort* traffic: it can be stalled without compromising the experience of the end user, hence the *efficiency* is set to 0.

```
Process('Process_Sata'
,  procedure = Choose([Read(1024),Write(1024)]) / 200**MBps
   >> Queue('Queue_Sata', depth = 2048 , flow = 'periph.sata/I/0')
,  efficiency = 0
)

Process('Process_Eth'
,  procedure = Choose([Read(128),Write(128)]) / 200**MBps
   >> Queue('Queue_Eth', depth = 2048 , flow = 'periph.eth/I/0')
,  efficiency = 0
)
```

5.6 Dynamic memory subsystem

To execute the scenarios on the SoC architecture, a model of the memory target must be provided. In our case, the transactions directed to the DRAM are handled by two different entities.

The *controller* connects to the interconnect system through a standard socket (with flow control on both request and response sides) and formats transactions in order to comply with the complex DDR protocol. It keeps transactions in order between the NoC socket and the DDR signals. However, compliance to DDR timings sometimes requires it to introduce temporal spacing between transactions, depending on their read or write operation and addresses. In such cases, the controller injects back-pressure on its request channel.

The *scheduler* is part of the interconnect system and arbitrates between different transactions to be sent to the controller. Since the controller may have to introduce penalties depending on the transaction's pattern, a correct interleaving of transactions is crucial from a performance perspective.

In our design, the scheduler is embedded in the NoC, but not the controller. The scenario describes the target model which actually represents the controller and the DRAM device. The figures have been extracted from a DDR2 memory data sheet, with 8 banks and a 1.5 ns data-bit length (666 MHz).

```
DRAMTargetModel(
    'DramModel'
,   frequency = 333.333**MHz
,   width     = 16
,   rowMask   = 0x3FF80000
,   bankMask  = 0x00001C00
,   penaltyActToAct = 54**ns
,   penaltyRdToMiss = 33**ns
,   penaltyWrToMiss = 48**ns
,   penaltyBurstLen = 6**ns
,   penaltyRdToWr   = 6**ns
,   penaltyWrToRd   = 18**ns
,   flows = 'DRAM/T/0'
)
```

The model takes into account the read-to-write (2 cycles) and write-to-read (6 cycles) penalties, as well as the penalties involved in opening and closing a row in a given bank. To do so, the model has to be informed of which bits of the address will actually be mapped on row and bank bits in the DRAM controller.

5.7 Performance specification summary

In the previous sections we described the performance behavior and requirements of the system in a particular use case, without specifying how the interconnection system is architected or implemented. We are now in a position to simulate this scenario against any implementation of the functional specification, and see whether and why its performance efficiency criteria are met or not.

This early traffic scenario modeling phase is not mandatory. For some projects, depending on the team's methodology, some IP behaviors are available early enough in the project lifetime. However, because the IPs come from different sources, the integration of their models is usually a burden by itself. Here, the lightweight (the total script description is less than 100 line long) and integrated aspects of a scripted scenario are a definite added value.

Obviously, more than one scenario can be given for a specification. Using Arteris tools, each scenario can be translated into a SystemC module, with TLM 2.0 interfaces.

6 Architecture exploration

After the definition of the functional and performance specifications, it is now time to allocate resources in the NoC so that it can achieve the expected performance.

A critical aspect is the level of abstraction of the architecture definition: It must be low enough to allow for fine tuning and at the same time high enough for the exploration of the possible implementation space not to be a burden. Arteris has chosen to describe the architecture using a path-based approach: Abstract resources called *links* can be created, and the architect can choose which links are traversed, in which order, indicating the request path from an initiator socket to a target socket followed by the response path back to the initiator. This description actually draws an underlying network topology, and resources can be tuned using link parameters such as serialization, clocking or buffering. The actual instantiation of building blocks is left to the later structure phase. The major advantage is that any architecture is valid from a functional perspective, and can thus be simulated. It is possible to smoothly refine the architecture from a very crude one to the final version, without de- and reassembling part of the NoC at each step.

For the data path, the architecture description offers the following degrees of freedom: Topology, Serialization, Clock, Buffering, Arbitration Schemes, Pipeline Stages, Transaction Contexts. A NoC is usually somewhat reconfigurable at run time using configuration registers, which are usually dispersed throughout the design. The architecture abstraction embeds a description of the *service* network used to reach those registers, as well as the description of an *observation* network for collecting errors and possibly software debug information or run-time performance measurements. This document will focus on the data-path description, leaving service and observation out of scope.

A NoC must keep track of the transactions pending on the various sockets. In the Arteris NoC library, those contexts are distributed at the boundary of the interconnect in the network interface units (NIU), so that the transport does not have to implement complex request/response association logic. Furthermore, the volume of contexts can be set differently for each socket. The amount of information to bookkeep per transaction depends on the capabilities of the socket (especially its ability to disorder or interleave responses), while the number of transactions worthwhile to track depends on the performance expected from *that* socket.

For the time being, Arteris has chosen to leave the NoC architecture in the hands of system or interconnect architects, as opposed to try and automatically find out a “reasonable” architecture fulfilling requirements. Three major reasons lead to this choice:

- The design space is huge. A *topology* represents by itself a large configuration space. Many additional parameters enable cost/performance/feasibility trade-offs; thousands of parameters for an architecture description is a good order of magnitude. Furthermore, the parameters are heterogeneous in their consequence on the design. A blind optimization algorithm is not likely to find a solution in acceptable time.
- A more reasonable approach would certainly be to select parameter values with a certain *strategy*. Unfortunately we have encountered roughly as many strategies to architecture an interconnect as companies we are in touch with. Most of those strategies are perfectly accurate in their own design space, and entirely irrelevant in others.

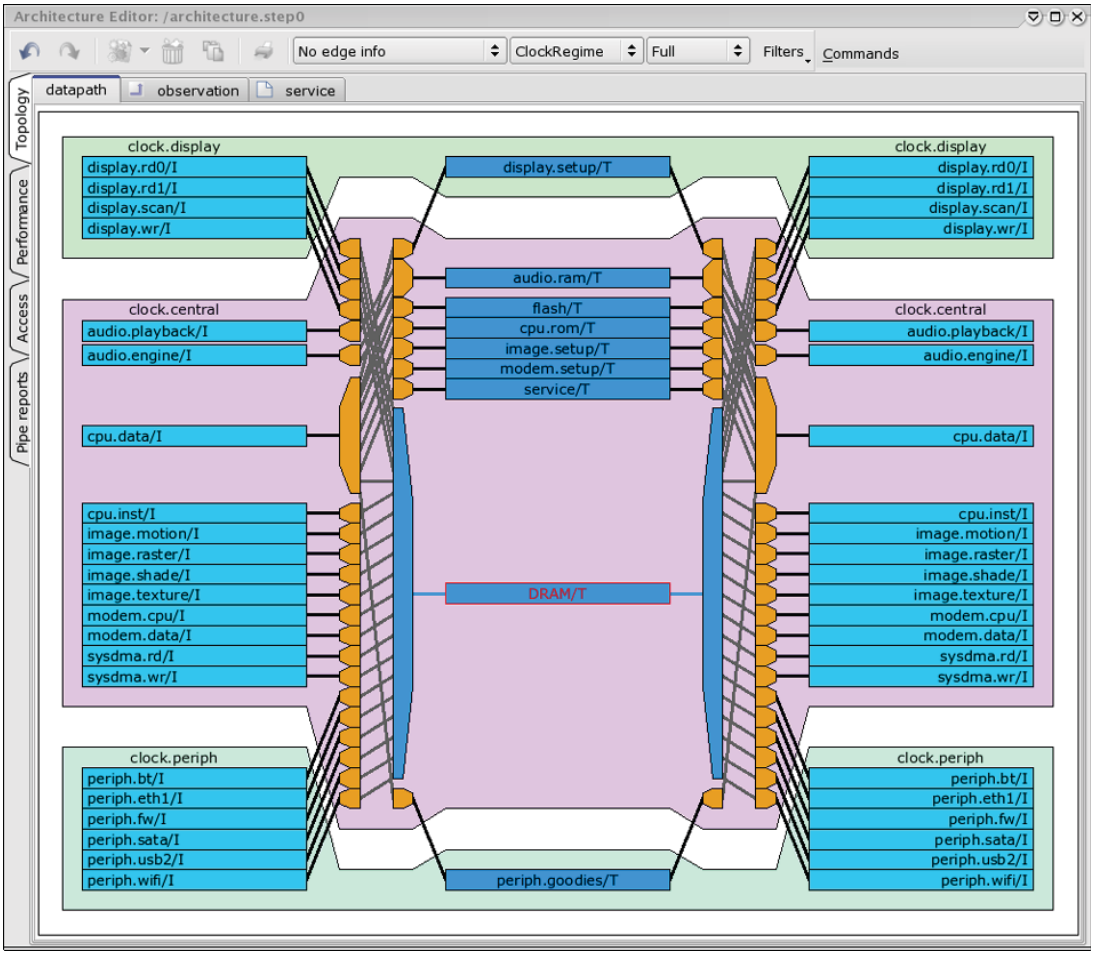

Fig. 2 Blank architecture

- The set of constraints shaping architecture decisions is much richer than the sole performance requirements. Clock domain crossing, frequency scaling, voltage shifting, power domain crossings, observability, integration in the tool flow, constitute as many feasibility constraints which differ from one project to another.

Eventually it has been considered more realistic to rely on system architects: *they* are the experts of *their* SoCs. The following sections sketch an example of possible progressive architecture refinement.

6.1 Step 1, topology

Figure 2 presents the blank architecture diagram of our interconnect. Nothing has been done yet, and the tool inferred the necessary routes from the connectivity table.

This figure and the following ones, are actually snapshots from the Arteris tool architecture editor window. They represent the possible routes for packets carrying transactions from the initiator sockets (light blue on the left-hand side) to the target sockets (dark blue in the middle) and back to the initiators (light blue on the right-hand side). Packets always flow from the left to the right: the topology is actually a directed acyclic graph. The first column of orange *cones* represents de-multiplexers, which will route packets through one path or another depending where the are headed to. The second column of orange *cones*

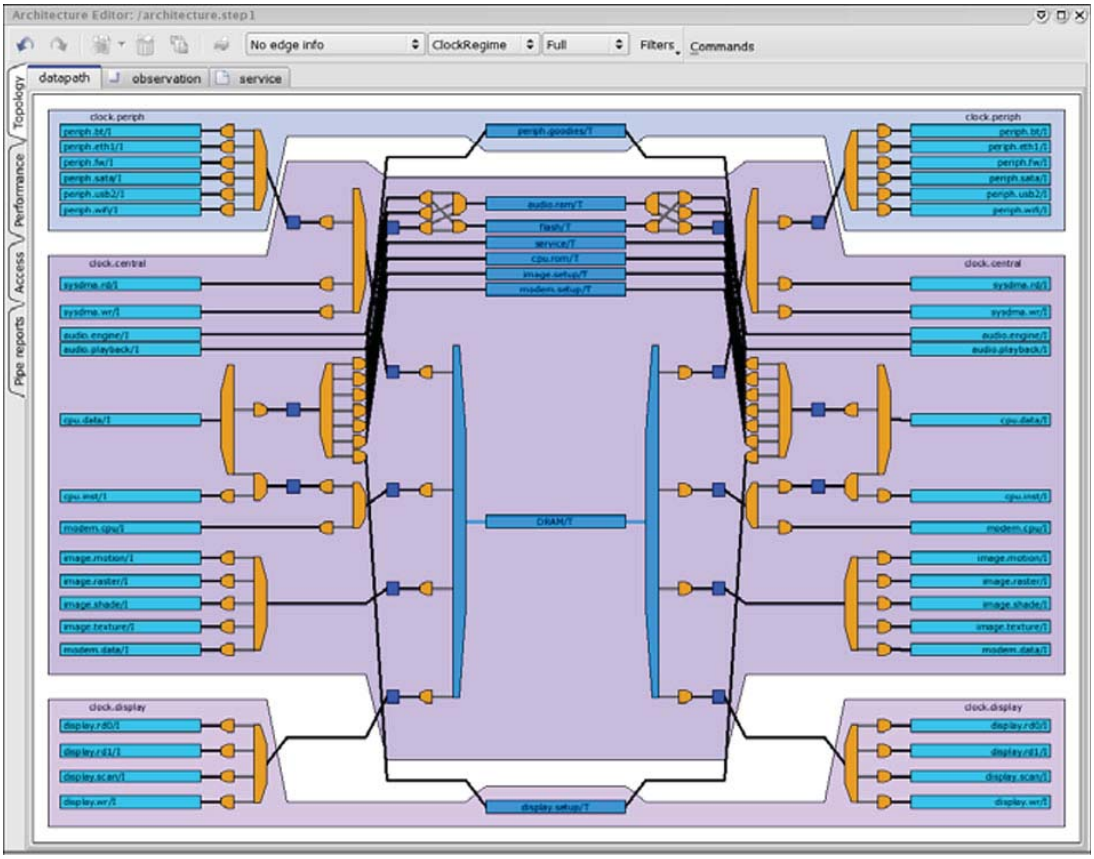

Fig. 3 Topology

represents multiplexers, which arbitrate among several potential packets. The background color represents here the clock regime of each element.

All initiators are able to send requests to the DRAM scheduler and responses can be sent back to the initiators. At the same time, the CPU data socket is able to reach all the small targets. It is already apparent that the massive concentration on the DRAM will be difficult to handle. Elements have been grouped by clock regimes. This architecture presents many (24) asynchronous domain crossings, which are quite expensive.

A first step is to sketch a topology, grouping some traffic towards the DRAM controller and limiting the number of domain crossings. But beforehand we intentionally oversize the contexts of the initiator NIUs, making all of them able to handle four pending transactions. We will fine-tune them in a later cost reduction step (Sect. 6.4).

Figure 3, shows the architecture diagram, once traffic has been grouped. The grouping is done by the insertion of *links*, represented as small dark blue squares. A *link* is a data-path element shared by packets flowing along several initiator to target routes. As any initiator or target, a link is preceded by a multiplexer and followed by a de-multiplexer. Choosing the links lets the architect decide what are the shared resources and in which order they are shared. At the extreme, the introduction of a single link shared by all the routes would create a bus structure where the requests from all initiators would first be arbitrated to a single point, before being dispatched to the right target. The architect can decide the number of bytes per cycle (the serialization) and the clock domain of each individual link.

The grouping involved the introduction of 8 links on the request side, and their respective duals on the response side. The number of asynchronous crossings has been reduced to 8,

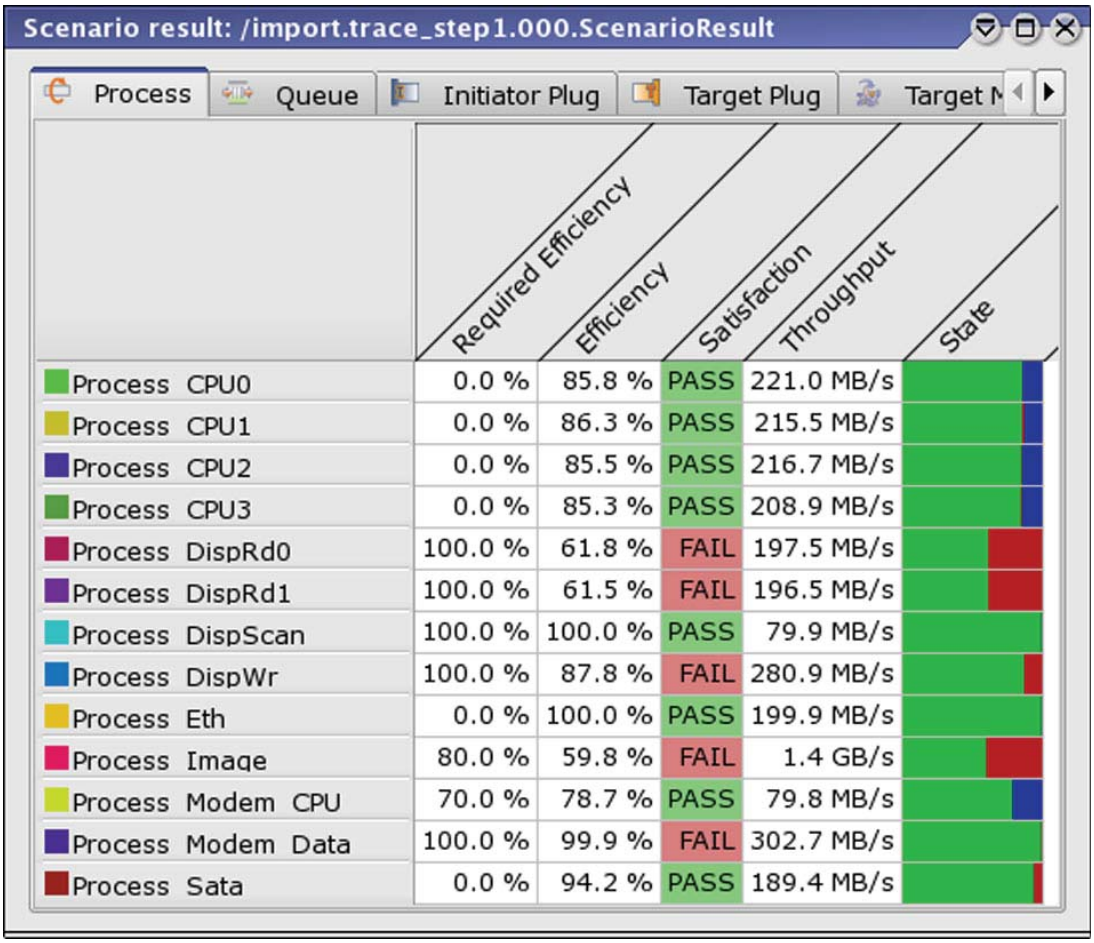

Fig. 4 Step 1, scenario results

and the number of ports of the DRAM scheduler to 4. The grouping is actually done by traffic class: Low latency, hard real-time, soft real-time and best effort.

As stated earlier, the DRAM scheduler is in charge of interleaving the requests before sending them to the controller. As the controller, it must be aware of the DRAM penalties in order to select adequate patterns. The quality of the results depends on the number of candidate transactions because finding a good candidate is more likely in the presence of many candidates. Arteris scheduler can handle one candidate per port, or more if they come from different initiators, or if the ordering policy of the initiator they come from allows it. As usual, the number of entries to be allocated for each port can be tuned in the architecture. For the time being, let's fix them to four entries per port, leading to a maximum of 16 candidates for the scheduler.

The serialization of the created links has been set to 8 bytes per cycle.

The tool is designed to place the SoC architect in the center of the exploration phase, and the SoC architect should not be limited by the simulation times of the scenarios of his designs. Taking account of the fact that reasonable simulated times are in the millions of cycles, a simulation speed of 10 to 100 thousands of cycles per second is required.

Such speeds are difficult to reach in pure RTL simulation. Furthermore, long RTL simulations generate huge volumes of data, making analysis all the more difficult. Therefore, it is much more convenient to export a behavioral model from the architecture description.

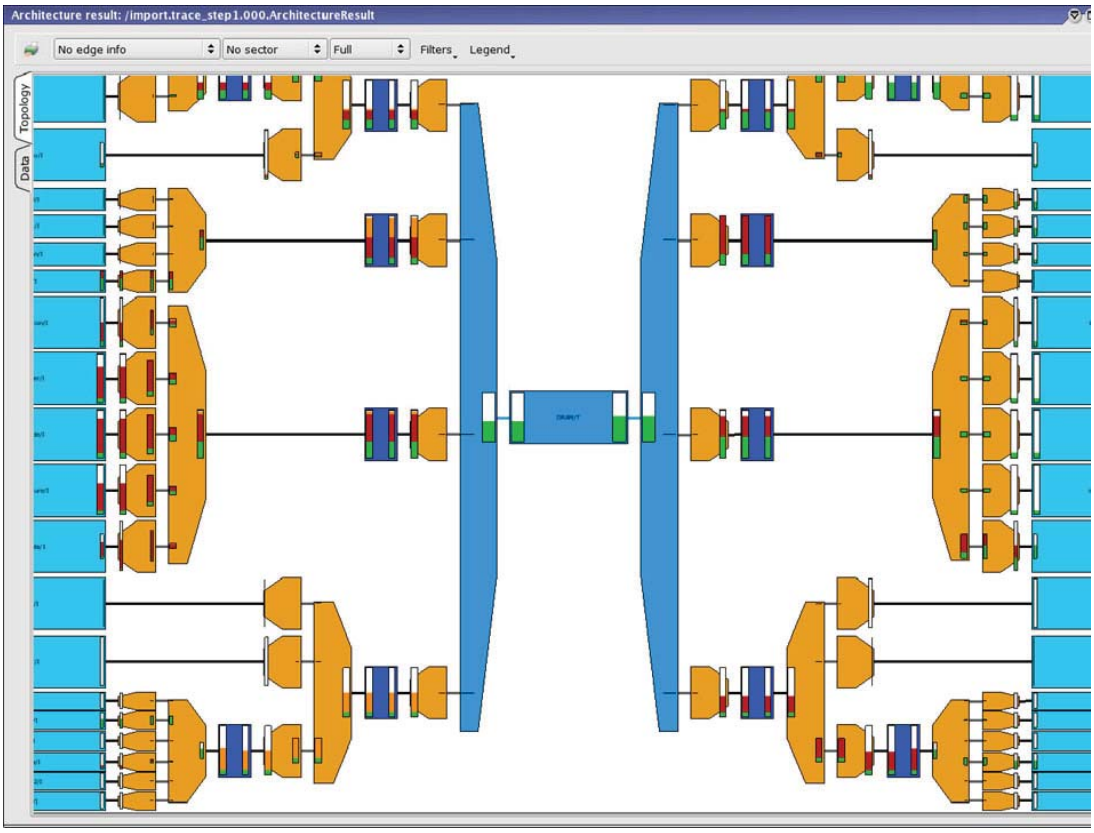

Fig. 5 Step 1, architecture results

The proposed architect's view (AV) model of the NoC is cycle-based, and takes into account most of the architecture parameters. Its interfaces are based on TLM2.0, and can be connected directly to the scenario model. Once the simulation has been run, the generated traces may be imported back in the tool framework where it is possible to browse through the aggregated statistical behavior of the various components for both the scenario and the architecture. For our use case, running a 100 µs simulation takes 20 seconds, including generating, compiling, running, and post-processing.

Figure 4 presents the scenario results, and more precisely the behavior of the various processes. Apparently, no real-time traffic constraint is met, and the image process only obtains 1.4 GB/s, when it expects 1.8 GB/s. The measured DRAM throughput is 3.777 GB/s, i.e., 71% of the total DRAM peak bandwidth. The architecture side of the same simulation is presented Fig. 5, where link states are annotated on the architecture topology: Green (XFER) when a word is actually transferred, red (BUSY) when a word is available to be transferred but stalled by the receiver, orange (WAIT) when the emitter is starving for data in the middle of a packet and white (IDLE) between packets. The dimensions of the bar-graphs represent the peak throughput: the height is proportional to the (local) clock frequency, and the width to the number of bytes per word, so that the area is proportional to the peak throughput.

The fact that the DRAM presents the highest peak bandwidth of the system is visible on the diagram. The case of the display subsystem (second port of the DRAM, starting from the top) is striking. On the response side, passing from 2.6 GB/s (8 B @ 333 MHz) at the output of the DRAM scheduler to 533 MB/s (4 B @ 133 MHz) on the sockets without any buffering necessarily creates busy states. Conversely, on the request side, the peak through-

put ratio introduces wait states on the DRAM port, preventing a correct use of the available bandwidth.

6.2 Step 2, rate adaptation

Those effects can be mended by the insertion of buffering. In the case of peak bandwidth reduction, a simple FIFO does the job: Busy states present at the output of the FIFO do not propagate back to the input until the FIFO is full. For a peak bandwidth increase, the situation is a bit more complex. In a FIFO, wait states present at the input are only absorbed when the FIFO is not empty. Arteris proposes a mechanism called rate adaptation, which stalls packets just enough to remove wait states from the packets, preserving a low latency.

In this second step, the architecture is modified to introduce some buffering. In our example 760 bytes of memory have been distributed across the topology. Some have been put on existing links; some required the creation of new links.

Just like the topology itself, the decisions about buffering distribution are left to the SoC interconnect architect. It actually took several trials by the authors of this paper to come to a final result we considered to be reasonable. Such a what-if analysis may only be efficient if simulation run times are short enough. This iteration loop is the best occasion to get familiar with this particular architecture, and especially to learn which are the most sensitive parameters.

The behavior of the updated architecture is presented Fig. 6. The bar graphs represent the buffers. Their width is proportional to the number of bytes per word, while their height is proportional to the number of words. The darker a given level is, the longer it has been occupied during the simulation. Notice the effect of buffers on busy and wait states.

Figure 7 shows the process scenario results. Once the buffers have been introduced in the architecture, the performance improves significantly compared to the first architecture (Fig. 4). However, the scenario performance requirement is yet not met. The traffic on the DRAM represents 79% of the available bandwidth (4.2 GB/s out of 5.333 GB/s).

Taking into account the DRAM read-to-write and precharge penalties, this functional point is certainly close to the possible ceiling, The satisfaction of the scenario is more likely to be obtained by a rational sharing of the traffic between initiators, rather than by increasing further the DRAM bandwidth. In particular, traffic considered as best effort (SATA and Ethernet) are almost entirely served, while a hard real-time one (DispRd0) is not.

6.3 Step 3, quality of service

In the Scenario section, the expected quality of service has been defined, yet nothing has been said about its actual impact in the architecture. As a general rule, since transactions cannot be thrown away, the only possible lever is the *order* transactions get to resources; in other words arbiters have to be biased by *priorities*.

The Arteris solution enables a priority to be attached to a transaction (it is then named *urgency*), or to all the transactions pending on a socket (*hurry*). This second mechanism permits us to dynamically raise the priority of a transaction which has already been requested.

We propose here to handle 4 priority levels: Low, normal, important, and critical. We allocate them with the following strategy:

- The low level is used by best effort traffic to steal idle apertures left by all the others.
- Most traffic is sent using a normal level.
- Latency sensitive traffic (specifically the read transactions of the CPUs) use the important level.

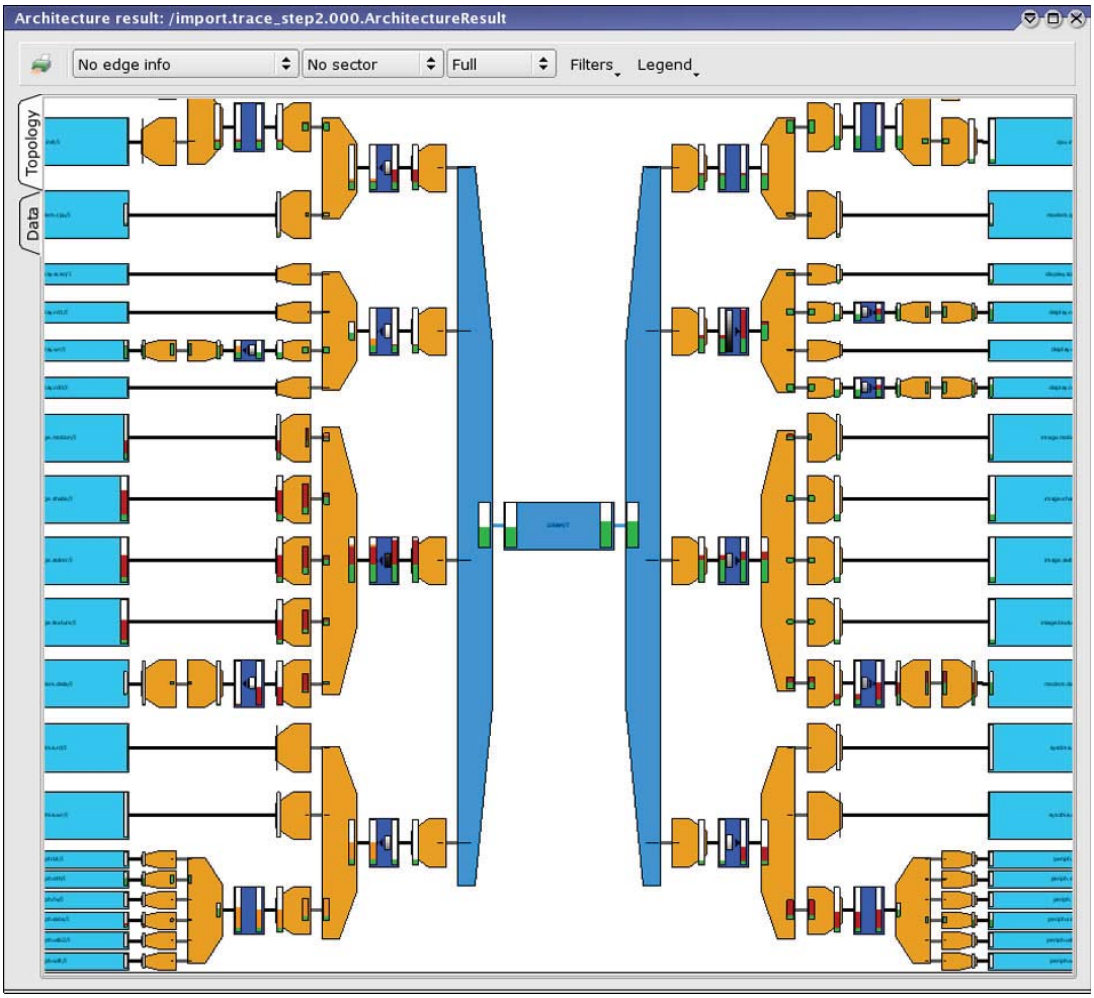

Fig. 6 Step 2, architecture

- The critical level is reserved for real time traffic on the verge of failure.

The best place to drive this priority information is inside the initiator itself. For hard real-time traffic, where the failure condition is usually a full or empty buffer, the fact that the buffer reaches a critical level is unknown just outside of the initiator. Because the priority information is usually not passed through standard socket protocols, it is sometimes difficult or even impossible to have access to reliable information. In these cases, heuristics have to be put in place.

For our study case, we will consider that the priority information is available for a display subsystem,typically carried by side band signals in addition of the standard OCP socket. The scenario is modified to represent this:

```
Process('Process_DispRd0'
,  procedure = Read(64)(randomDramAddr) / 320**MBps
      >> Queue(
            'Queue_DispRd0'
         , depth = 1024
         , flow = 'display.rd0/I/0'
         , hurryThresholds = [0,256,256]
         )
```

```
,  efficiency = 1
)
```

The hurryThresholds attribute on the queue modulates the hurry level as a function of the queue filling level. It is set to 1 (normal) if the level is equal to or higher than 0 (i.e. always), and changed to 3 (critical) if the level is equal to or higher than a quarter of the queue depth.

For the other traffic, the configuration can be done in architecture.

- Best effort traffic can be left untouched.
- Latency sensitive traffic may have its urgency modulated as a function of the transaction: *Normal* for writes and *important* for reads.
- Soft real-time traffic may have its hurry level modulated as a function of the bandwidth it receives: *Critical* until a specified bandwidth is obtained on a sliding 4 microsecond window, and *normal* thereafter. These settings are set through configuration registers and may be modified while the interconnect is running. The mechanism is called a bandwidth regulator.
- On the real-time modem data port, the hurry is fixed at a critical level.

Those last two points are fallback positions because of the lack of information on the IP sockets that could be used to drive the priority of each transaction. These fallback positions are actually pessimistic; a number of transactions are marked as *critical* for fear they *could*

Scenario result: /import.trace_step2.000.ScenarioResult

Process | Queue | Initiator Plug | Target Plug | Target

	Required Efficiency	Efficiency	Satisfaction	Throughput	State
Process CPU0	0.0 %	78.2 %	PASS	201.3 MB/s	
Process CPU1	0.0 %	79.2 %	PASS	198.2 MB/s	
Process CPU2	0.0 %	78.5 %	PASS	198.2 MB/s	
Process CPU3	0.0 %	77.1 %	PASS	189.4 MB/s	
Process DispRd0	100.0 %	99.3 %	FAIL	317.0 MB/s	
Process DispRd1	100.0 %	99.6 %	FAIL	317.9 MB/s	
Process DispScan	100.0 %	100.0 %	PASS	79.9 MB/s	
Process DispWr	100.0 %	100.0 %	PASS	319.8 MB/s	
Process Eth	0.0 %	100.0 %	PASS	199.9 MB/s	
Process Image	80.0 %	70.5 %	FAIL	1.7 GB/s	
Process Modem CPU	70.0 %	74.8 %	PASS	75.7 MB/s	
Process Modem Data	100.0 %	100.0 %	PASS	302.9 MB/s	
Process Sata	0.0 %	86.8 %	PASS	174.1 MB/s	

Fig. 7 Step 2, scenario results

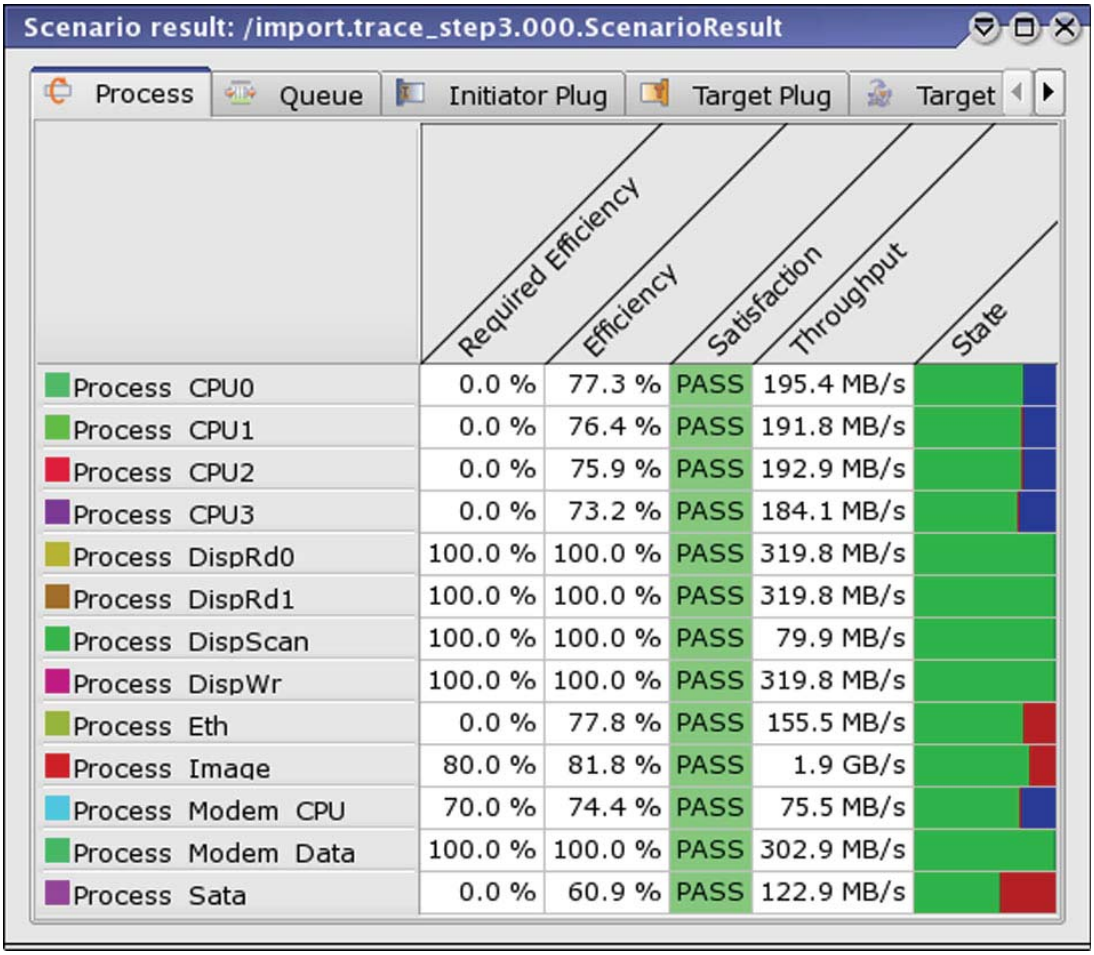

Fig. 8 Step 3, scenario results

be critical. Those uselessly critical transactions will pass ahead of the latency-sensitive CPU accesses, thus slightly impacting the overall performance. The priority information is reconstructed inside the initiator NIU and is used to bias arbitration not only in the memory scheduler but also the other arbitration points of the request network.

Figure 8 shows the scenario results once the modifications have been applied. As desired, the display subsystem is entirely satisfied. The image subsystem has obtained more bandwidth, impacting the two best effort ones. The DRAM achieved 4.347 GB/s, i.e. a 82% efficiency, compared to its absolute 5.333 GB/s peak bandwidth.

Now that a functional point has been found, it is interesting to see how the system reacts when traffic conditions vary. The traffic with the highest variability is certainly the CPU behavior; the cache miss rate highly depends not only the software activity, but also on the way the software is written. In Fig. 9, each point on the X-axis is a separate simulation. Each simulation lasts 1 ms. The parameter is the CPU spontaneously requested bandwidth, correlated to the cache miss rate expressed in B/s. The point in the middle (1 GB/s) is actually exactly the scenario which has been simulated beforehand.

First, we verify that in all those simulations, the scenario performance requirements are met. Then, we report the bandwidth actually obtained by each subsystem on the Y-axis. The Y-scale has been set to the DRAM peak bandwidth (5.333 GB/s). Note how CPU bandwidth is gained against *image* and *best-effort* traffics, while *modem* and *display* are untouched. However, due to the bandwidth regulator, the *image* traffic is not reduced below 1.85 GB/s.

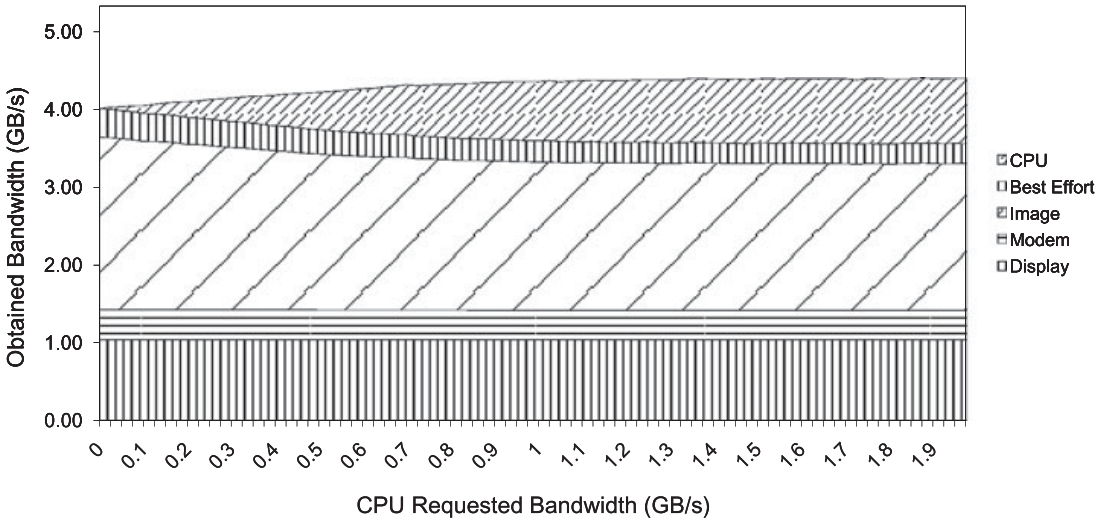

Fig. 9 Reaction to CPU variation

6.4 Step 4, cost reduction

When we began the performance exploration, we willingly over-dimensioned some resources in the initiator and target NIUs. We are now in a position to adjust those resources. Figure 10 shows a table of initiator resource usage when the architecture is simulated against our reference scenario. The *transactions* column is the histogram of the number of transactions pending. For instance, the line "display.scan/I" states that the NIU has been dimensioned to handle up to 4 transactions, but only a maximum of 2 have been seen pending at the same time along the simulation, with an average of 0.53. The *Routes* column is the histogram of different routes usage, keeping in mind that transactions with different OCP tags or AXI AIDs are considered on different routes. Similar reports are available for transport and target NIU resources.

With such reports it is possible to understand how resources are used, and allocate them accordingly. The exact saving cannot be truly known before the gate level netlist is available, however, the gate count may be estimated at the structure phase, see Sect. 7.

6.5 Step 5, pipeline stages for timing closure

In SoC designs, timing closure is the phase where true hardware constraints strike back against designer's best intentions. This is all the more true for the NoC interconnect, which is spread across the die. Without some attention, it is not unusual to see a flow control signal with a combinatorial dependency from all sockets! Arteris' NoC IP library has been designed with a special attention to limit the intrinsic complexity of building blocks, avoiding big context tables and complex state machines. The transaction and packet protocols themselves have been specified to avoid corner cases, with the very same objective.

Nevertheless, inserting pipeline stages everywhere is not a solution either, because it induces cost (a 64-bit wide pipeline stage for both data and flow control costs over thousand gates) and has a performance impact, especially on latency-sensitive paths. In the architecture description, it is possible to insert pipeline stages almost anywhere along routes, as depicted Fig. 11. Each triangle is an occasion to insert a pipeline stage, green ones have actually been inserted while gray ones have not. Distinction is made between the forward stages, which break combinatorial dependencies on data and framing signals, represented

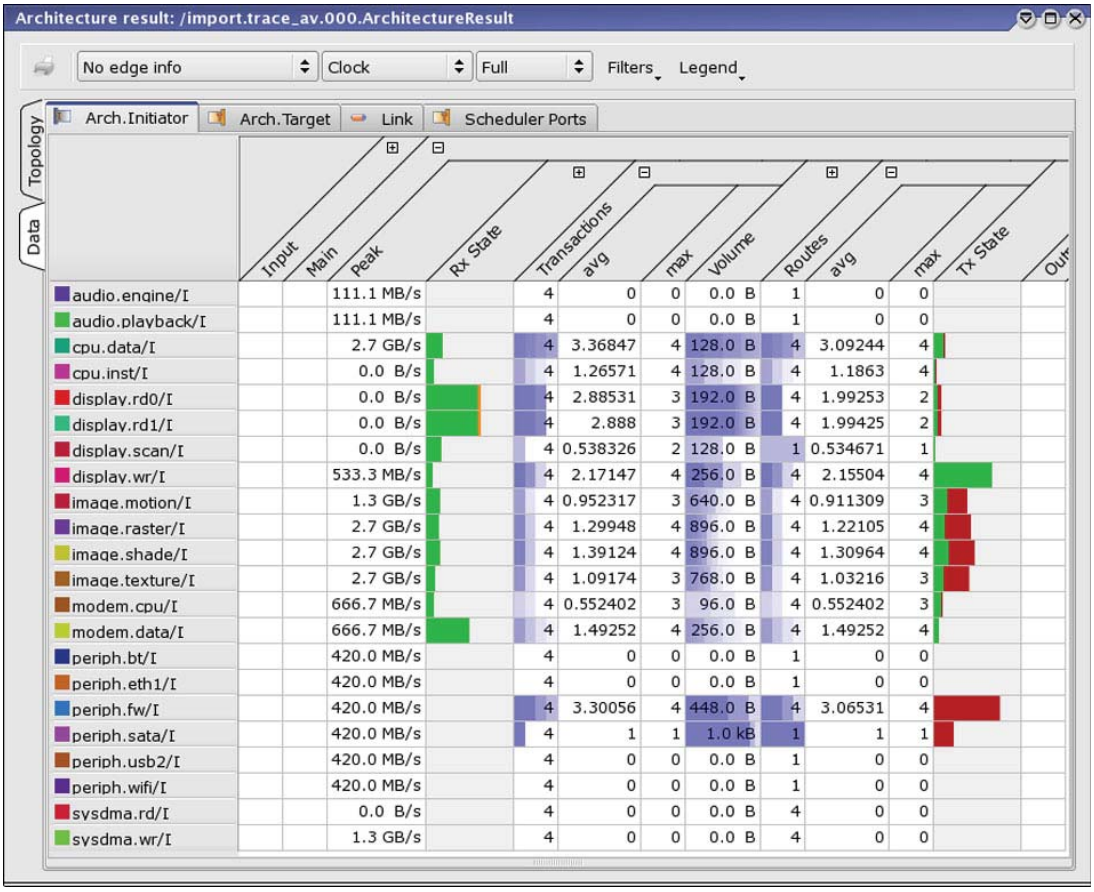

Fig. 10 Initiator NIU resource usage

as right-pointing triangles, and the left-pointing backward stages acting on the flow control signal.

The usual initial strategy is to minimize the pipeline stages on the main CPU to memory route(s) and to slightly over-provision other routes. If, at the back-end phase, long paths prevent timing convergence, it is still possible to react by inserting pipeline stages as required.

6.6 Late project changes

The architecture description we reached looks reasonable. We are ready to pass to the structure phase, where RTL is created. Beforehand, let's say a few words about late project changes. Even though time to market is a key aspect of modern SoC design, the duration of projects remains more or less flat, 18 months being a good ballpark figure. The market pressure is such that changes in component objectives are inevitable, impacting the NoC interconnect. Arteris' objective is of course to shorten the overall interconnect design time, but special attention has been paid to minimize the impact of these late changes.

The architecture description is organized to provide a minimum redundancy with the specification description. If the specification is changed, the information entered at architecture level does not become obsolete. Obviously the architecture sometimes has to be augmented to take specification changes into account, in the case of a socket creation, for example. But most changes such as socket destruction, renaming, clock changes, etc., are easy and seamless to implement.

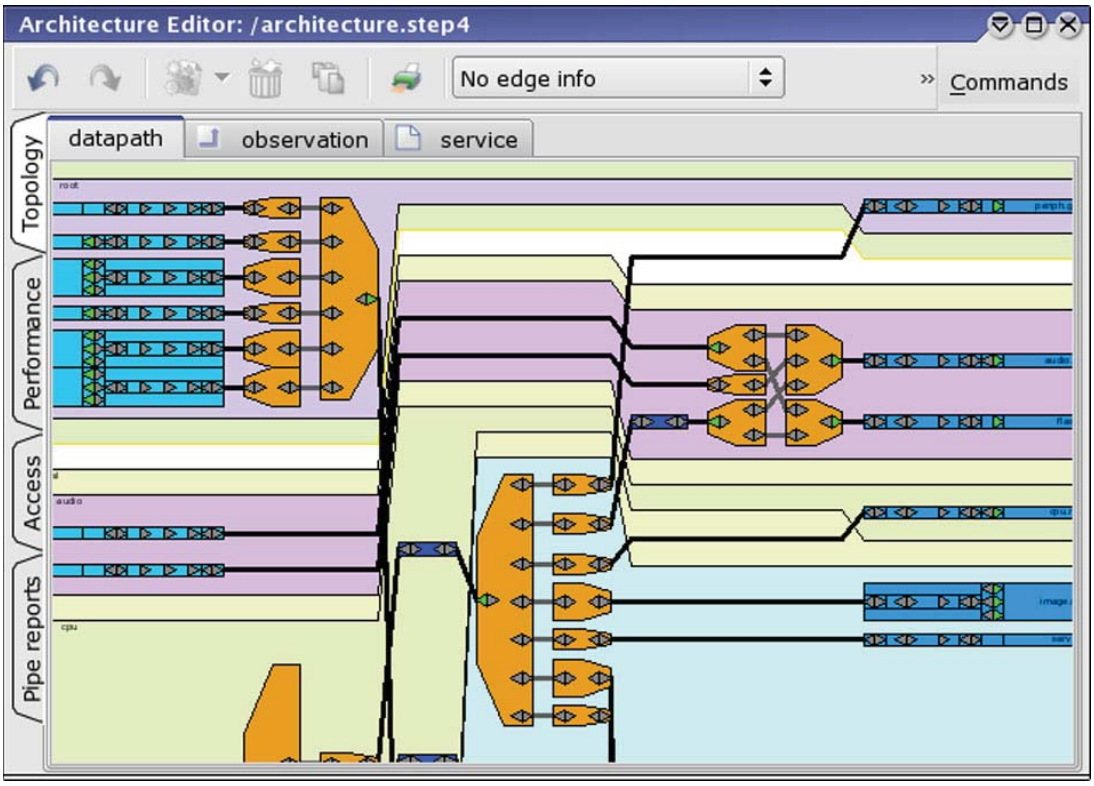

Fig. 11 Architecture pipelines stages

7 Structure synthesis

The structure description is another layer of additional, more refined information, on the top of an architecture description. The level of abstraction is shifted to a classical hardware netlist: A hierarchy of instances whose ports are interconnected by nets. This structural synthesis task is entirely automated in the tool, mapping the architecture on a library of pre-verified hardware building blocks.

Figure 12 shows approximately one half of the obtained netlist.Block colors indicate their function: Network Interface Unit—NIU (blue), transport (green), service (orange) or clocking (pink). Such a view is certainly not adequate for a design phase. The abstraction level far too low. It is, however, quite convenient for control purposes. Depending on the backend process it is sometimes necessary to separately synthesize, place and route parts of the NoC. The best way to prepare for this is to cast the different instances into a user-controlled hierarchy of modules.

The netlist may be exported, ready to be simulated or synthesized in various hardware languages: Verilog, VHDL or cycle-accurate, bit-accurate SystemC, all of these coming from a common, internal, hardware description of the library. Synthesis scripts can be exported together with the behavioral description, placing the right timing constraints correctly, including on the tricky clock-domain crossing paths.

Fine tuning the netlist is still possible to some extent. For example, any configuration register may be frozen to a fixed value, saving the flip-flops and propagating the constants in the logic. Some tactical connections are possible as well. In our case, we decided to drive the *hurry* of the display subsystem sockets from the outside of the NoC, but this signaling does not exist on standard OCP. On Fig. 12, the four selected, bold signals in the top left corner, connect specially created input ports of the interconnect system to the display NIUs.

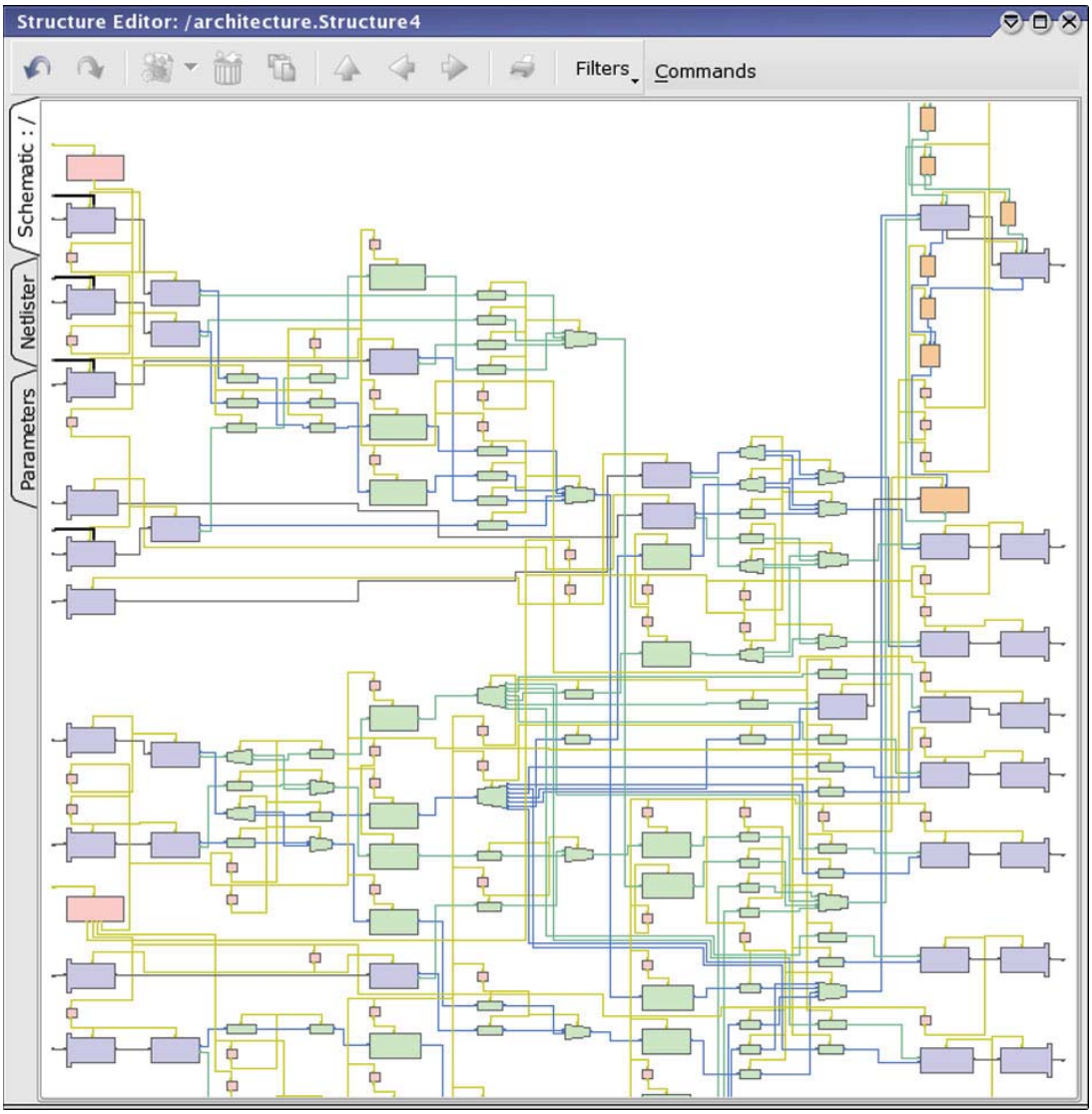

Fig. 12 Partial structure view

As hardware IP blocks become available, it is possible to get an estimate of the generated NoC gate count. Since the (usually physical aware) gate synthesis has not been run yet, such a count can only be an estimate. It is useful though for the architect to have feedback on where the silicon area is consumed in the design. For instance, the area saved in the cost reduction phase (Sect. 6) is estimated to be 15%.

8 Verification phase

As with any piece of IP, the interconnect system must be thoroughly verified before being cast into silicon. The verification phase is split into functional verification and the performance verification.

8.1 Functional verification

Functional verification must be checked at two different levels.

First, each socket behavior must locally comply with its protocol. In other words, the NoC must accept any legal transaction request on the initiator side and any transaction response on the target side. Conversely, it must drive legal transaction requests on the target side and legal transaction responses on the initiator side. The socket protocols also include ordering rules, which must be verified as well.

Second, the global functional specification must be verified. Accesses on the initiator side must be correctly translated to the right address of the right target, taking the memory map, connectivity table and security settings into account. When dynamic partial power supply switching capability is enabled, it must be globally verified as well.

The Arteris tool suite distinguishes between the notions of supervision and stress. A *supervised* NoC may be exported. It embeds the full RTL description, and has the very same signal interface; but it also includes socket monitors for local protocol verification and a scoreboard for global functional verification. The supervised NoC does not expect any particular transaction to take place, but is able to tell if a protocol rule has been violated, or if a transaction has not been translated in conformance to the memory map. As the supervised NoC has the same signal interface as the RTL NoC, it may be substituted one to one, including in a global SoC platform, where IP behaviors are embedded as well. Separately, a *test bench* can be exported. The test bench attempts to *stress* the interconnect in various ways, using constrained random transaction generation. Additionally, the supervised NoC also includes a number of coverage measurement points in order to verify the quality of the stimuli.

Both the supervised NoC and the test bench are based on System Verilog, using VMM or OVM environments. The protocol checkers are actually not developed by Arteris, reducing the risk of having a protocol rule coherently misunderstood in both the implementation and the verification logic. The tool exports a consistent and sufficient set of stimuli, coverage measurements and score-boarding rules. However, as the OVM/VMM environments are open, it is possible to extend those at will.

8.2 Performance verification

For performance verification, the Arteris tool enables the creation of a Verification View (VV) SystemC module with the very same TLM2.0 interface as the AV model used during architecture exploration. It can thus be simulated against the scenarios defined in Sect. 5. However, unlike the Architect View (AV) model, the VV model embeds the complete bit-accurate, cycle-accurate description of the NoC produced by the structure phase. Additional logic is added to convert abstract TLM2.0 transactions into signals for the various sockets around the interconnect.

The scenario behavior against the VV model can be imported in the tool framework, just like we did in the architecture phase. Actually the scenario does not *know* the abstraction level of the interconnect system it is simulated against. Figure 13 shows the aggregated results of our nominal scenario, to be compared with the Fig. 8.

Results are extremely close between the AV model and the RTL exact VV behavior. The measured throughput on the DRAM is 4.349 GB/s, compared to 4.347 GB/s with the AV model. Table 1 compares the measured average latencies, on CPU sockets, between the AV and VV abstractions, for a 1 ms simulation.

The cycle-based, pipeline exact AV model runs very fast: the complete 1 ms simulation lasts 15 s on a laptop. It can yet be used in confidence during architecture exploration, and cross checked with the slower (1 ms simulated in 250 s) but RTL exact VV behavior during performance verification.

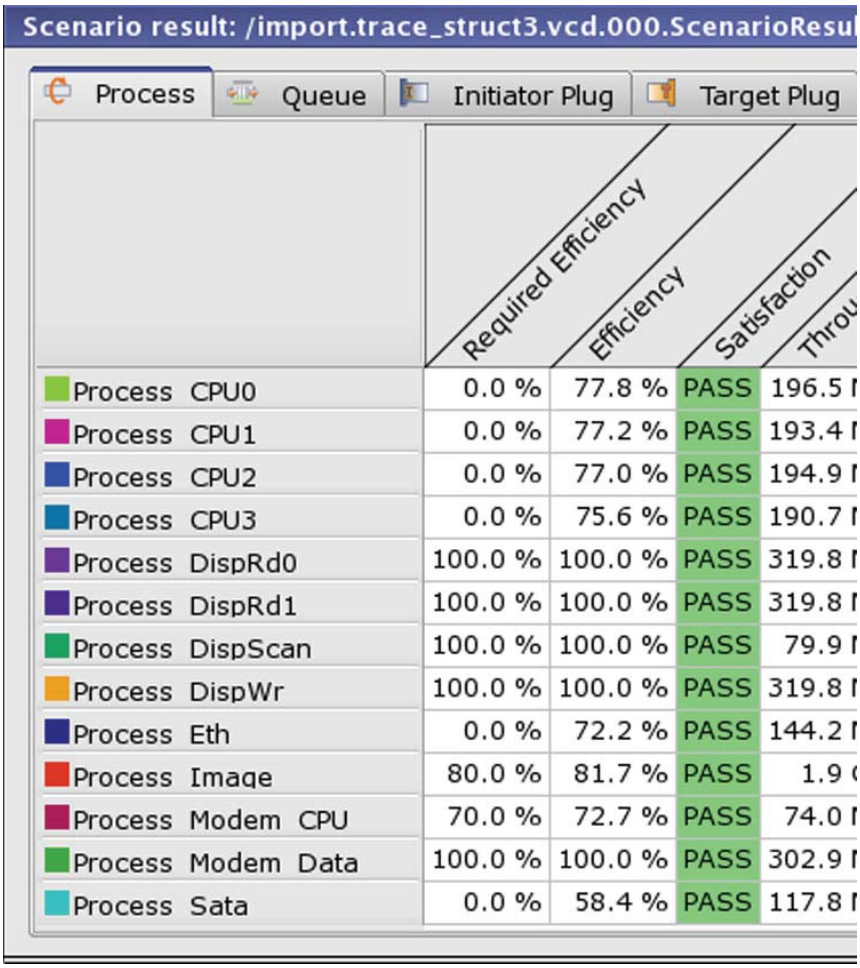

Fig. 13 Verification scenario results

Table 1 AV & VV latencies

Socket	Architect View (AV)	Verification View (VV)
CPU Data	226 ns	222 ns
CPU Instruction	184 ns	185 ns
Modem CPU	234 ns	241 ns

9 Conclusion

In less than a week of work for a single person, we have been able to define, analyze and refine a NoC interconnect architecture for a particular SoC application. Such a powerful approach can easily be adapted to other applications or other market segments. The reduced delays permit to bring forward the integration phase of a SoC design project cycle, where IP blocks are connected and tested altogether. The proposed use case is just an example. The quality of service strategy may have to differ in other environments, such as multiple dynamic memory devices, initiators with different performance requirements, or 3D chips with through-silicon-vias (TSVs) where the pads are not the expensive resource anymore. Yet the layered top-down methodology will not lead you astray, the flexibility of the Arteris

library will fit your needs, and the associated tool framework will permit a clear description and validation of your NoC.