Des Autom Embed Syst (2011) 15:159–190
DOI 10.1007/s10617-011-9073-7

A quantitative evaluation of a Network on Chip design flow for multi-core consumer multimedia applications

Andreas Hansson · Kees Goossens

Received: 25 May 2010 / Accepted: 28 February 2011 / Published online: 30 March 2011

Abstract A growing number of applications are integrated on the same System on Chip in the form of hardware and software Intellectual Property (IP). Many applications have firm or soft real-time requirements and require bounds on latency and throughput. To accommodate the growing number of application requirements, the on-chip interconnect must offer scalability on the physical, architectural and functional level.

Networks on Chip (NoC) are proposed as a scalable communication architecture that is also able to deliver guaranteed performance. Traditionally, NoCs focus on delivering physical and architectural scalability. The functional scalability, i.e. the ability to satisfy an increasing number of increasingly demanding requirements with a constant cost/performance ratio, is often overlooked. The onus is on the interconnect design flow that translates user requirements to an interconnect instance. While mature tooling exists for many of the IPs, interconnect design flows are an active research area, with few concrete examples, and few large-scale case studies.

As the main contribution of this work, we demonstrate a complete operational interconnect design flow for multiple real-time applications, and quantitatively evaluate the functional scalability on two large-scale industrial case studies. We illustrate the steps of the flow, going from requirement specification all the way to simulation of synthesised netlists in a 90 nm and 65 nm low-power standard-cell technology. We show that the interconnect and design flow offer scalability, on the physical, architectural as well as the functional level.

Keywords System on Chip · Network on Chip · Design flow · Case study

A. Hansson (✉)
University of Twente, Enschede, The Netherlands
e-mail: andreas.hansson@ieee.org

K. Goossens
Eindhoven University of Technology, Eindhoven, The Netherlands

1 Introduction

Systems on Chip (SoC) grow in complexity with an increasing number of *independent applications* on a single chip [67]. The applications are realised by hardware and software Intellectual Property (IP), e.g. processors and application code, and many applications have *real-time* requirements on their end-to-end behaviour [15]. One of the major challenges in SoC design is ensuring that all application real-time requirements are fulfilled under all circumstances [25]. In a firm real-time application, such as a software-defined radio [53], deadline misses are *highly undesirable* due to standardisation, e.g. upper bounds on the response latency in wireless standards, or steep quality reduction in the case of misses. Soft real-time applications, e.g. an MPEG-2 decoder [55], can tolerate occasional deadline misses with only a modest quality degradation. The growing number of applications and the increasing requirements require a system architecture that is scalable on the physical, architectural and functional level [39].

Networks on Chip (NoC) are proposed to overcome the scalability issues of traditional interconnects like crossbars and buses [18]. NoCs address the physical scalability by splitting the interconnect into multiple segments with relaxed timing requirements [16, 64] and offer architectural scalability through a structured and modular architecture [6]. It is thus possible to implement a network that scales with the number of IPs in the design [43, 72]. NoCs also offer guaranteed services, i.e. latency and throughput bounds, through various resource-reservation schemes [3, 9, 26, 40, 45, 50, 59, 60, 66, 74]. However, physical and architectural scalability coupled with the availability of guaranteed services does not necessarily imply that the interconnect scales with increasing requirements.

The difference between architectural scalability and functional scalability lies in the ability to not only provide a hardware capability, but actually translate that capability into useful work, i.e. satisfied requirements. Compare this situation to e.g. a modern general-purpose multi-core computer architecture where advanced compiler technology is needed to exploit the hardware [20]. Similarly for a NoC, we need design tools that can turn requirements into an interconnect instance. While mature tooling exists for the computation and storage elements of the SoC, interconnect design tools is an active research area [25]. Compared to traditional interconnects, a NoC is typically tailored for a given set of applications, and one of the more complex parts of the SoC. As such, it also plays an important role in defining the area and power consumption, and design effort of the overall system [48]. Thus, a operational tool flow is a critical, but often overlooked, part of on-chip interconnect design, and key to functional scalability.

As the main contributions of this work, we: (1) describe a complete design flow to instantiate and verify application-specific interconnect instances, (2) quantitatively evaluate the entire architecture and design flow using the communication requirements of two large-scale industrial case studies, and (3) investigate the functional scalability of the proposed interconnect by looking at the cost/performance ratio of hundreds of complete interconnect instances as the requirements change. Our two examples are both high-end chips with a high degree of IP integration. The first example resembles a digital TV set-top box ASIC in the line of designs described in [24, 41]. The second example is based on an automotive info-tainment ASIC [51, 61]. In both examples, the architecture consists of one or more microcontrollers, supported by several domain- or function-specific hardware cores, needed to achieve a high computational performance at an acceptable power consumption.

The remainder of this paper is structured as follows. First we review related work in Sect. 2, also highlighting the important properties of our interconnect architecture. Then, Sect. 3 gives an overview of the proposed design flow, using the digital TV SoC as the running example. We continue with an in-depth evaluation of the TV SoC in Sect. 4, followed by the automotive system in Sect. 5. Finally, we conclude in Sect. 6.

2 Related work

We review related work in a bottom-up fashion, based on the levels of scalability. First, we look at how different interconnects address physical scalability, i.e. the ability to scale the interconnect to larger die sizes (transistor count, wire delay) without impairing the performance. Thereafter we raise the level and look at architectural scalability, i.e. the ability to add more IPs without negatively affecting the performance. Lastly, we turn to the functional scalability, i.e. the ability to translate a growing number of (more demanding) application requirements into a hardware and software interconnect instance.

2.1 Physical scalability

Many networks [26, 45, 50, 60] rely on a (logically) *globally synchronous* clock. While techniques such as link pipelining have been proposed to overcome link latency [64, 70], the cycle-level synchronicity severely constrains the clock distribution [78] and negatively affects the scalability [12]. A range of *asynchronous* networks [3, 10, 66] completely remove the need for a clock, and aim to leverage the characteristics of asynchronous circuits to reduce power and electromagnetic emissions. While offering physical scalability, these networks rely on not yet well-established design methodologies and verification techniques. They also complicate the provision of guaranteed services. To overcome the disadvantages of global synchronicity, but still enable a traditional synchronous design style, the networks in [12, 27, 36, 47, 59, 75, 78] use *mesochronous* links between synchronous network elements. By allowing local phase differences, clock distribution and placement is greatly simplified [78]. Moreover, the global phase difference is allowed to grow with chip size.

The constraints on the clock distribution within the interconnect apply also to the (global) control infrastructure. Thus, dedicated interconnects [49, 80] that rely on global synchronicity impair the overall physical scalability and must be avoided. By reusing one and the same interconnect for both user data and control, as proposed in [28, 60], the problem is reduced to providing one scalable interconnect.

In addition to relaxing the constraints on the clock distribution within the interconnect and control infrastructure, it is crucial for physical scalability that IPs are able to use independent clocks (from each other as well as the interconnect). While asynchronous networks natively offer a Globally Asynchronous Locally Synchronous (GALS) design approach at the IP level, e.g. through the use of pausable clocks [4], synchronous and mesochronous networks must add Clock Domain Crossings (CDC) at the interfaces to the IPs. Solutions based on bi-synchronous FIFOs are proposed in [11, 26, 29, 59]. Such a clock domain crossing, e.g. [76], is robust with regards to metastability and offers a simple protocol between synchronous modules, while still allowing every module's frequency and voltage to be set independently [42].

In this work we use the logically synchronous Æthereal [26] network with mesochronous links [34]. Figure 1 shows a part of an example instance showing the architectural building blocks, e.g. routers (R), Network Interfaces (NI), shells, and buses. The blocks are presented in detail in [29] and we return to discus their functionality in Sect. 3.1. The mesochronous (and possibly pipelined) links inside the network alleviate the designer from strict requirements on clock skew and link delay, thus enabling an effective distributed placement of the

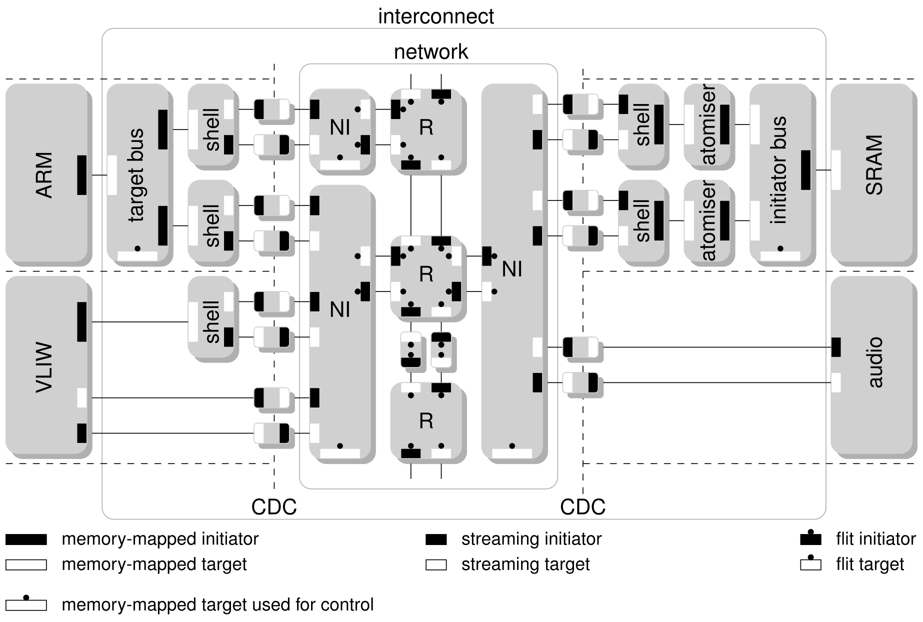


Fig. 1 Interconnect architecture overview

network components even for larger die sizes. Furthermore, clock domain crossings between the network and the IPs enable a GALS design approach. The same interconnect is used for both data and control, as described in [28].

2.2 Architectural scalability

In bus-based interconnects, e.g. the Silicon Backplane [79], the performance decreases as the amount of sharing increases. In crossbar-based interconnects, e.g. Prophid [44], a growing number of IPs lead to a rapid growth in silicon area and wire density. This in turn causes layout issues (e.g. timing and packing) and negatively affects performance [64]. Although the problem can be mitigated by using a partial crossbar [54], the key to architectural scalability is a *distributed multi-hop interconnect*.

In a multi-hop interconnect, e.g. Sonics MX [69] that uses a hybrid shared-bus and crossbar approach, the individual buses and crossbars are not scalable on their own. Rather, the architectural scalability comes from the ability to arbitrarily add more buses and crossbars without negatively affecting the performance of other parts of the interconnect. A segmented bus [16, 62] achieves architectural scalability by partitioning a bus into two or more segments, operating in parallel, and connected by bridges. NoCs extend on the concepts of segmented buses and offer a structured approach to multi-hop interconnects [6, 18].

From the perspective of architectural scalability, there is an important distinction to be made between NoCs where arbitration is performed at every hop (routers), e.g. [3, 10, 59, 64, 66] and those where arbitration only takes place only at the edges (NIs) of the network, as done in Nostrum [50], aSoC [45], Æthereal [26], and TTNoC [60]. The works in [3, 10, 59, 64, 66] use virtual channels to provide connection-based services. As a consequence, the router area grows quadratically with the number of connections passing through the router. Moreover, the maximum frequency of the router is also reduced, and the router is likely to be part of the critical path of the network. In contrast to the NoCs that rely on virtual channels, the NoCs in [26, 45, 50, 60] all have *stateless* [71] routers. Thus, the router is independent of the number of connections in terms of area (number and depth of buffers) and speed (arbitration). This moves the router out of the critical path and pushes the issue of scalability to the NIs, which can be heavily pipelined without being in the critical path [29].

In this work we use a small and fast router design [34], with no restrictions on the network topology. The architectural scalability stems from the ability to arbitrarily add more links, routers, NIs, shells and buses without negatively affecting the services of the existing components [29]. Memory-mapped initiator ports that use distributed memory communication, e.g. the ARM in Fig. 1, may even use different NIs for the connections to different targets. Similarly, a memory-mapped target port that is shared by multiple initiators may distribute the connections across multiple NIs. Thus, as we shall see in Sects. 4 and 5, it is possible to push the point of contention all the way to the IP ports and offer arbitrarily low latency and high throughput in the interconnect, limited only by the intrinsic throughput and latency of a single link.

2.3 Functional scalability

Much work focus on generating, exploring, evaluating, formalising, and comparing NoC architectures and instantiations [2, 5, 8, 14, 22, 25, 38, 39, 45, 48, 52, 63]. However, only a few works address guaranteed services [25, 39, 45], and most works are limited to evaluations of a single small-scale example, or only a few steps of the design flow, as exemplified by the overview in [48]. Moreover, the evaluation of the design flows is predominantly done using high-level simulation or analytical models rather than actual implementations.

A few NoC design flows have been demonstrated on FPGA [2, 22, 33, 43, 52], predominantly using simple media-centric applications like M-JPEG [33] and MPEG-2 [43]. In addition to FPGAs, complete NoC-based SoC ASICs are presented in [7, 17, 72] together with a post-layout evaluation. However, neither of the works evaluate the scalability and instead focus on a single design point. Moreover, rather than considering the IPs given, the works in [2, 22, 43, 52, 72] customise the entire system to fit with the NoC, e.g. by modifying the IPs [43] or adding NoC-specific instructions [72]. In an industrial context this is often not possible and the interconnect has to accommodate a wide range of IP interfaces [29]. Most works also do not provide any real-time guarantees.

A number of works move from the architecture and connection level and look at performance analysis of actual applications when mapped to the NoC [13, 35, 68]. By modelling the applications, the works also solve the problem of buffer sizing, which is closely linked to the end-to-end performance requirements. Probabilistic analysis, using *stochastic self-similar traffic models*, is used in [13, 68], whereas [35] is based on *dataflow models* and analysis proposed in [77]. The two approaches are complementary and can be used in parallel in our current work, depending on the application. All works [13, 35, 68] focus on application models and buffer sizing and do not discuss the implications on scalability.

The composable and predictable multi-processor SoC presented in [33] demonstrates an interconnect design flow, based on [25], that is able to provide real-time guarantees, also on the application level [35]. The work focuses on the qualitative concepts and studies a few applications in detail. No evaluation is made of the scalability in terms of requirements.

In this work, we focus on quantitatively proving functional scalability of NoC design, using large-scale examples that represent state-of-the-art SoCs for consumer multimedia applications. We consider the IPs and the requirements given and demonstrate a complete flow that is able to satisfy the real-time requirements. Using the examples, we evaluate the

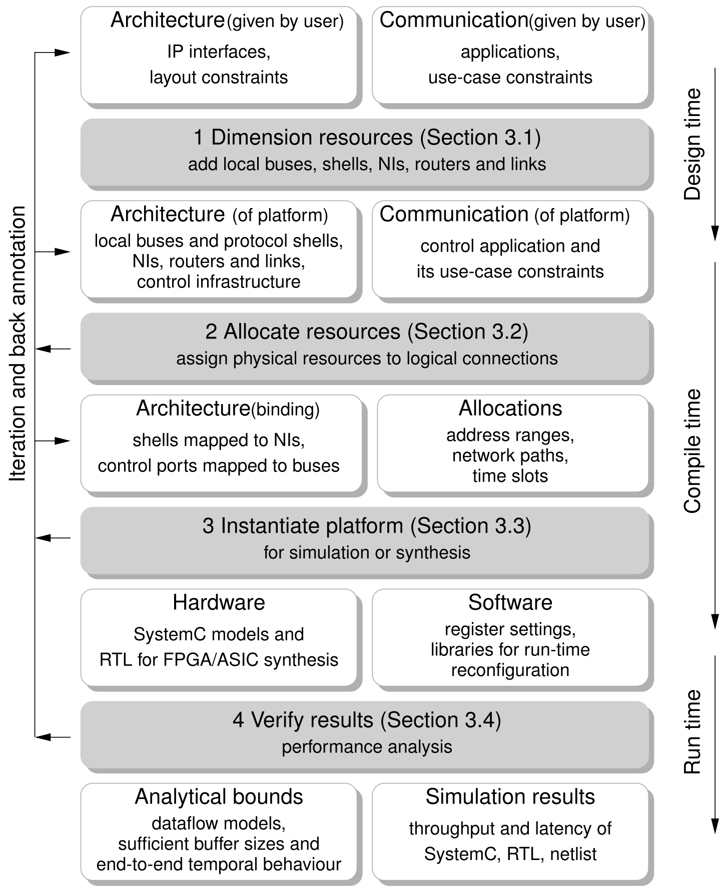

Architecture (given by user)
IP interfaces, layout constraints
Communication (given by user)
applications, use-case constraints
1 Dimension resources (Section 3.1)
add local buses, shells, NIs, routers and links
Architecture (of platform)
local buses and protocol shells, NIs, routers and links, control infrastructure
Communication (of platform)
control application and its use-case constraints
2 Allocate resources (Section 3.2)
assign physical resources to logical connections
Architecture (binding)
shells mapped to NIs, control ports mapped to buses
Allocations
address ranges, network paths, time slots
3 Instantiate platform (Section 3.3)
for simulation or synthesis
Hardware
SystemC models and RTL for FPGA/ASIC synthesis
Software
register settings, libraries for run-time reconfiguration
4 Verify results (Section 3.4)
performance analysis
Analytical bounds
dataflow models, sufficient buffer sizes and end-to-end temporal behaviour
Simulation results
throughput and latency of SystemC, RTL, netlist
Iteration and back annotation
Design time
Compile time
Run time

Fig. 2 Interconnect hardware and software design flow

quantitative aspects of the proposed interconnect and also look in greater detail on the scalability of the hardware architecture as well as the design flow, about which more presently.

3 Design flow

One of the main contributions of this work is the design flow depicted in Fig. 2, introduced here and discussed in detail in Sects 3.1 through 3.4. Note that many of the steps are previously published in part, but no complete overview is given, and the design flow as a whole is not evaluated. In the figure, white boxes denote specifications (stored in XML files [23]) and the grey boxes denote (collections of) tools that manipulate the specifications. As shown in the figure, the flow takes as its starting point the specification of the physical interfaces that are used by the IPs, i.e. everything outside the interconnect in Fig. 1. An example (based on the digital TV SoC in Sect. 4) is shown in Listing 1. The specification starts by declaring the clocks, giving them symbolic names and setting the period (in ns). This is followed by the various IPs and their ports. Each IP (and potentially port) either specifies a clock name (as done by *U2* on line 17) or uses the default global clock (like the *host*). A port

```
 1<architecture id='85500_inspired'>
 2  <!-- Global parameters -->
 3  <parameter id='clk' type='string' value='global_533MHz'/>
 4
 5  <!-- Clocks (period in ns) -->
 6  <clk id='global_533MHz' period='1.87' />
 7  <clk id='tm_450MHz' period='2.19' />
 8  <clk id='vid_200MHz' period='5' />
 9  <clk id='mem_200MHz' period='5' />
10  ...
11
12  <!-- IPs (port width in bits) -->
13  <ip id='host' type='Host'>
14    <port id='pi' type='Initiator' protocol='MMIO_DTL' />
15  </ip>
16  <ip id='U2' type='IP'>
17    <parameter id='clk' type='string' value='vid_200MHz' />
18    <port id='s1' type='Initiator' protocol='MMIO_DTL' />
19    <port id='s2' type='Initiator' protocol='MMIO_DTL'>
20      <parameter id='width' type='int' value='8' />
21    </port>
22    ...
23  </ip>
24  <ip id='mem_ddr' type='IP'>
25    <parameter id='clk' type='string' value='mem_200MHz' />
26    <port id='p1' type='Target' protocol='MMIO_DTL'>
27      <parameter id='width' type='int' value='64' />
28    </port>
29    <port id='p2' type='Target' protocol='MMIO_DTL' />
30    ...
31  </ip>
32  ...
33
34  <!-- Placement constraints (as regular expressions) -->
35  <constraint id='group1' port='U2.*' />
36  <constraint id='group2' port='mem_ddr.*' nis='NIx(0|2)y0n[0-1]'/>
37  ...
38 </architecture>
```

Listing 1 Architecture specification

specifies whether it is an *initiator* or a *target*, and what protocol it uses [1, 19, 57]. A data width of 32 bits is assumed for the ports unless the default is overridden by a parameter (line 21 and 27). The last part of the user architecture specification is the placement constraints that enable the user to group IP ports and thereby enforce a common NI (selected from a subset that can also be specified). The constraints use regular expressions to specify both the ports and the NIs (as shown on line 36).

The architecture description is complemented by the communication requirements of the applications, specified per application as a set of logical interconnections between IP ports, each with bounds on the maximum latency and minimum throughput. Listing 2 shows a subset of the connections for the digital TV SoC. Each connection is assigned a logical identifier and specifies which initiator port and target port it connects. The temporal behaviour is specified separately for reads and writes (both or only one as shown on line 23). In both cases, the throughput specification is given in Mbyte/s and the latency requirement in ns. Together with the ports' protocol, the (minimum) burst size (given in bytes) is used to bound the overhead of the interconnect serialisation, as we shall see in Sect. 3.2. The last part of the communication specification is the use-case constraints, determining how the applications

```
 1 <communication>
 2   <!-- Applications -->
 3   <application id='storage'>
 4     <connection id='0'>
 5       <initiator ip='U2' port='s1' />
 6       <target ip='mem_ddr' port='p1' />
 7       <read bw='150' burstsize='256' latency='1500' />
 8       <write bw='100' burstsize='256' latency='900' />
 9     </connection>
10     <connection id='1'>
11       <initiator ip='U2' port='s2' />
12       <target ip='mem_ddr' port='p1' />
13       <read bw='100' burstsize='256' latency='1500' />
14       <write bw='100' burstsize='256' latency='900' />
15     </connection>
16     ...
17   </application>
18
19   <application id='vid_dec'>
20     <connection id='0'>
21       <initiator ip='vliw' port='instr' />
22       <target ip='mem2_ddr' port='ptm' />
23       <read bw='50' burstsize='128' latency='50' />
24     </connection>
25     ...
26   </application>
27
28   <!-- Use-case constraints -->
29   <constraint type='allow' appl='storage' with='.*' />
30   <constraint type='allow' appl='vid_dec' with='.*' />
31   ...
32 </communication>
```

Listing 2 Communication specification

may be combined temporally. In this example, all applications are assumed to be running concurrently (e.g. allow to run with all on line 29 and 30), but it is possible to benefit from mutually exclusive applications [31].

The architecture and communication specification are assumed to be given by the application and system designer, referred to as the user in Fig. 2. The interface specifications follow directly from the choice of IPs, and the use-case constraints are part of the early system design. It is less obvious, however, where the temporal requirements on the interconnect, i.e. the latency and throughput requirements of the connections, stem from. Depending on the application, the desired throughput and latency may be analytically computed [35], the outcome of high-level simulation models of the application, or simply guesstimates based on back-of-the-envelope calculations or earlier designs [41]. The design flow makes no distinction of the real-time character, e.g. if the requirement reflects an average case or a worst case. The requirement will be satisfied (assuming the IPs have the right characterisation) and the interconnect is robust with respect to potential over-asking from the IPs.

Section 3.1 begins our bottom-up description of the design flow, introducing the design-time dimensioning of the interconnect. This is followed by a discussion on the compile-time allocation of resources in Sect. 3.2. Section 3.3 describes how the resources and allocations come together in an instantiation at run time. Next, Sect. 3.4 outlines how the interconnect hardware and software comes together in an analysis model, and how it is applied in buffer sizing and formal verification of the end-to-end application behaviour. Throughout the design flow, there are many opportunities for successive refinement, as indicated by the iteration in Fig. 2. Iteration is either a result of failure to deliver on requirements or a result of cost assessment, e.g. silicon area or power consumption. It is also possible that the specification of requirements changes as a result of the final evaluation. The outcome of the proposed design flow is a complete SoC interconnect architecture (hardware) and a set of resource allocations (software) that together implement the requirements specified by the user. As we shall see plenty examples of in Sects. 4 and 5, the resulting platform is instantiated (in SystemC or synthesisable HDL), together with the software libraries required to configure and orchestrate use-case switches (in Tcl or ANSI C).

3.1 Dimensioning

The dimensioning constitutes Step 1 in Fig. 2 and is responsible for adding and connecting hardware resources, pushing the complexity of allocating and using the resources to the later parts of the design flow. Nevertheless, the dimensioning is split into three steps, corresponding to three tools in the design flow.

The first step of the dimensioning (`busdim`) bridges between the network and the IP ports by dimensioning buses together with atomisers and protocol shells, i.e. everything belonging to the interconnect but not the network in Fig. 1. Initiator and target buses are added to the IPs that make use of shared and distributed memory, respectively. The dimensioning of these buses is driven by the maximum number of concurrent connections for the IPs (based on the communication requirements). In Fig. 1, for example, the ARM has two concurrent connections to distributed memories, and the SRAM is shared by two concurrent connections. Shells are added based on the IP port protocol and the atomisers are placed in front of shared target ports (in combination with initiator buses) to ensure a fixed burst size [29]. Clock domain crossings are inserted where necessary. Listing 3 shows examples of the modules that are added to the architecture, all described in more depth in [29]. Note that the entire specification from Listing 1 is still included and the interconnect components are appended. The initiator bus (line 21) serves to exemplify a parametrisable IP. The ports of the initiator bus also show how identifiers are added to distinguish between the ports. At this point, the architecture also contains a number of connects, representing wires between two (compatible) ports.

The following step (`netwdim`) dimensions the network topology, adding the NIs, the routers and links, as shown in Listing 4. This step is based on a user specification of the router topology, the link pipeline stages and the number of NIs per router. As detailed in [34], the network uses three-word flits, wormhole switching and source routing. A few basic topology types, e.g. meshes, tori and rings, are supported by the dimensioning to simplify the task. It is possible to customise the output or to specify a fully custom topology by editing the XML file. The resource allocation guarantees deadlock freedom both on the network and transaction level [32]. Note that the NIs, at this point, are connected to a router and have a control port, but no ports connecting them to the IPs.

The final part of the interconnect dimensioning is the addition of the control infrastructure (`ctrldim`), as shown in Listing 5. One control bus (line 28) is instantiated for each NI and the control ports of all (user and interconnect) IPs are connected to the control buses. The necessary ports are added to the NIs (line 8 to 15), and shells are added to connect the FIFO ports of the NIs and the memory-mapped ports of the control buses. The appropriate connects are added to enable a host IP access to the control infrastructure and the address map of the control buses is determined according to user specification. This step also adds control connections to the communication requirements [28]. The control connections are automatically added as an independent application (not shown for brevity) and allocated resources like any user connection by the succeeding steps of the design flow.

```
<architecture id='85500_inspired'>
  <!-- Everything from Listing 1 -->
  ...

  <!-- Shells -->
  <ip id='U2_s1_shell' type='TrgtShell'>
    <parameter id='clk' type='string' value='vid_200MHz' />
    <port id='pt' type='Target' protocol='MMIO_DTL' />
    <port id='pst' type='Target' protocol='FIFO_AE' />
    <port id='psi' type='Initiator' protocol='FIFO_AE' />
  </ip>
  <ip id='U2_s2_shell' type='TrgtShell'>
    <parameter id='clk' type='string' value='vid_200MHz' />
    <port id='pt' type='Target' protocol='MMIO_DTL' />
    <port id='pst' type='Target' protocol='FIFO_AE' />
    <port id='psi' type='Initiator' protocol='FIFO_AE' />
  </ip>
  ...

  <!-- Buses -->
  <ip id='mem_ddr_p1_bus' type='InitBus'>
    <parameter id='clk' type='string' value='mem_200MHz' />
    <parameter id='arbiter' type='string' value='CCSP' />
    <port id='pi' type='Initiator' protocol='MMIO_DTL' />
    <port id='pt0' type='Target' protocol='MMIO_DTL'>
      <parameter id='portid' type='int' value='0' />
    </port>
    <port id='pt1' type='Target' protocol='MMIO_DTL'>
      <parameter id='portid' type='int' value='1' />
    </port>
    ...
  </ip>
  ...

  <!-- Clock domain crossings -->
  <ip id='mem_ddr_p1_bus_pt1_shell_pst_cdc' type='CDC'>
    <port id='psi' type='Initiator' protocol='FIFO_AE'>
      <parameter id='clk' type='string' value='mem_200MHz' />
    </port>
    <port id='pst' type='Target' protocol='FIFO_AE' />
  </ip>
  ...

  <!-- Connects -->
  <connect>
     <endpoint ip='mem_ddr_p1_bus' port='pi' />
     <endpoint ip='mem_ddr' port='p1' />
  </connect>
  <connect>
     <endpoint ip='mem_ddr_p1_bus_pt0_shell' port='pi' />
     <endpoint ip='mem_ddr_p1_bus' port='pt0' />
  </connect>
  ...
</architecture>
```

Listing 3 Architecture with shells and buses

At this point, the streaming ports of the IPs and shells are not yet connected to NIs. It is more efficient to do the mapping of IPs to NIs as part of the path finding and slot allocation [30] and we therefore delay this step, about which more next.

```
<architecture id='85500_inspired'>
  <!-- Everything from Listing 3 -->
  ...

  <!-- Routers -->
  <ip id='Rx0y0' type='Router'>
    <port id='p0_in' type='Target' protocol='FLIT_AE'>
      <parameter id='portid' type='int' value='0' />
    </port>
    <port id='p0_out' type='Initiator' protocol='FLIT_AE'>
      <parameter id='portid' type='int' value='0' />
    </port>
    <port id='p1_in' type='Target' protocol='FLIT_AE'>
      <parameter id='portid' type='int' value='1' />
    </port>
    <port id='p1_out' type='Initiator' protocol='FLIT_AE'>
      <parameter id='portid' type='int' value='1' />
    </port>
    ...
  </ip>
  ...

  <!-- Pipelined links -->
  <ip id='Rx0y0_p1_out_0' type='Link'>
    <port id='trgt' type='Target' protocol='FLIT_AE' />
    <port id='init' type='Initiator' protocol='FLIT_AE' />
  </ip>
  ...

  <!-- NIs -->
  <ip id='NIx0y0n0' type='NI'>
    <port id='p0_in' type='Target' protocol='FLIT_AE' />
    <port id='p0_out' type='Initiator' protocol='FLIT_AE' />
    <port id='pgm' type='Target' protocol='MMIO_DTL' />
  </ip>
  ...

  <!-- Connects -->
  <connect>
     <endpoint ip='Rx0y0' port='p0_out' />
     <endpoint ip='Rx0y1' port='p0_in' />
  </connect>
  <connect>
     <endpoint ip='NIx0y0n0' port='p0_out' />
     <endpoint ip='Rx0y0' port='p2_in' />
  </connect>
  <connect>
     <endpoint ip='Rx0y0' port='p1_out' />
     <endpoint ip='Rx0y0_p1_out_0' port='trgt' />
  </connect>
  ...
</architecture>
```

Listing 4 Architecture with network

3.2 Allocation

After dimensioning the interconnect, the application requirements area translated into resource allocations, based on the temporal requirements and the use-case constraints. This constitutes Step 2 in Fig. 2 and involves deciding what bus ports, NIs, paths and time

```
<architecture id='85500_inspired'>
  <!-- Everything from Listing 4 -->
  ...

  <!-- NIs -->
  <ip id='NIx0y0n0' type='NI'>
    ...
    <port id='NIx0y0n0_cfgbus_rmt_shell_ps_in' type='Target'
     protocol='FIFO_AE'>
      <parameter id='iqsz' type='int' value='3' />
      <parameter id='portid' type='int' value='0' />
    </port>
    <port id='NIx0y0n0_cfgbus_rmt_shell_ps_out' type='Initiator'
     protocol='FIFO_AE'>
      <parameter id='oqsz' type='int' value='3' />
      <parameter id='portid' type='int' value='0' />
    </port>
  </ip>
  ...

  <!-- Shells -->
  <ip id='NIx0y0n0_cfgbus_rmt_shell' type='TrgtShell'>
    <port id='pt' type='Target' protocol='MMIO_DTL' />
    <port id='pst' type='Target' protocol='FIFO_AE' />
    <port id='psi' type='Initiator' protocol='FIFO_AE' />
  </ip>
  ...

  <!-- Control buses with fixed address decoders -->
  <ip id='NIx0y0n0_cfgbus' type='TrgtBus'>
    <parameter id='default_port' type='int' value='0' />
    <port id='pt' type='Target' protocol='MMIO_DTL' />
    <port id='lcl' type='Initiator' protocol='MMIO_DTL'>
      <parameter id='addr_equal' type='int' value='0x80000000' />
      <parameter id='addr_mask' type='int' value='0xfffffc00' />
      <parameter id='portid' type='int' value='0' />
    </port>
    <port id='rmt' type='Initiator' protocol='MMIO_DTL'>
      <parameter id='addr_equal' type='int' value='0x80004000' />
      <parameter id='addr_mask' type='int' value='0xffffc000' />
      <parameter id='portid' type='int' value='1' />
    </port>
  </ip>
  ...

  <!-- Connects -->
  <connect>
     <endpoint ip='host' port='pi' />
     <endpoint ip='NIx0y0n0_cfgbus' port='pt' />
  </connect>
  <connect>
     <endpoint ip='NIx0y0n0_cfgbus' port='rmt' />
     <endpoint ip='NIx0y0n0_cfgbus_rmt_shell' port='pt' />
  </connect>
  ...
</architecture>
```

Listing 5 Architecture with control infrastructure

slots to use. Allocations are created and verified at *design or compile time*. Hence, *run-time* choices are confined to choosing from the set of fixed allocations. While limiting the run-time choices to a set of predefined use-cases, this is key as it enables us to guarantee, at compile time, that all application constraints are satisfied, and that all the use-cases fit on the given hardware resources.

The allocation part of the flow is split into two tools. The first step (`busalloc`) allocates ports on the initiator and target buses. In Sect. 3.1 we only dimensioned the buses, i.e. we only decided the number of bus ports, and did not yet decide what connection uses what port in what use-case. In case of distributed memory communication, we allocate an initiator port on the target bus. Similarly, we allocate a target port on the initiator bus in case of shared memory communication. The selection of a bus port implicitly selects a shell on each side of the network (as there is a one-to-one mapping between shells and bus ports).

Listing 6 shows the output of the bus allocation. The allocation of bus ports is made by a greedy heuristic that iterates over the applications, based on the number of use-cases they span. The intuition behind this ordering is that the ports used by an application cannot be used by any other application that runs concurrently. Hence, the more use-cases an application spans, the more restrictions are imposed on the not yet allocated applications. After the bus allocation, each connection from Listing 2 is split into a *request channel* and a *response channel* between two pairs of streaming ports. In case of a memory-mapped protocol, these streaming ports are located on the shells. For a streaming protocol, on the other hand, the ports are those of the IP itself. After performing the bus-port allocation, the succeeding steps of the allocation flow looks at requirements in terms of channels between streaming ports. As seen in Listing 6, each channel has a data rate (in Mbyte/s) that includes the effects of serialisation in the protocol shells. In addition to the user-specified connections the allocation flow also assigns resources to the control connections (line 30 and onwards). Listing 6 also shows the arbiter settings of the initiator bus.

After the bus ports are assigned to connections, the second step (`netwalloc`) decides the *spatial mapping* [37, 58] of streaming ports to NIs, i.e. where in the *given network topology* to connect the IPs, and also determines the channels' paths and time slots. The mapping of IPs to NIs is obviously only possible if the resource allocation is done at design time (since it changes the interconnect hardware). If the allocation is done at compile or run time, the given mapping is left unchanged. The allocation selects paths through the router network and allocates an appropriate set of time slots on the selected paths such that the throughput and latency requirements are met. As discussed in [30], we exploit the close correspondence between the two spatial allocation problems and let the path-selection algorithm take the mapping decisions by incorporating the streaming ports (shells and streaming ports of IPs) in the path-selection problem formulation. Similarly, we include the slot allocation in the path pruning to guarantee that a traversed path is able to satisfy a channel's requirements. The resulting architecture is shown in Listing 7, showing the newly added NI ports and the connects between the shells and NIs. Note that the input and output queues of the NIs at this point have a default size, here 255 words each. The corresponding network resource allocation is shown in Listing 8, with the additions compared to Listing 6 (paths and slots) highlighted.

The allocation together with the dimensioning is central to the functional scalability of the interconnect as these steps are responsible for instantiating and then finding appropriate resources to satisfy the application requirements. As we shall see in Sects. 4 and 5 the tools are indeed able to satisfy a growing number of requirements with a constant cost/performance ratio. However, despite the use of heuristics, the core of the path selection is A*Prune which is known to have exponential growth in run time for certain problem instances [46].

```
<allocations>
  <!-- Allocations for user applications -->
  <allocation application='storage'>
    <connection id='0'>
      <channel type='request'>
        <from ip='U2_s1_shell' port='psi' />
        <to ip='mem_ddr_p1_bus_pt0_shell' port='pst' />
        <parameter id='drate' type='double' value='109.375' />
      </channel>
      <channel type='response'>
        <from ip='mem_ddr_p1_bus_pt0_shell' port='psi' />
        <to ip='U2_s1_shell' port='pst' />
        <parameter id='drate' type='double' value='150' />
      </channel>
      <initbus bus='mem_ddr_p1_bus' port='pt0'>
        <parameter id='numerator' type='int' value='3' />
        <parameter id='denominator' type='int' value='51' />
        <parameter id='initialCredits' type='int' value='51' />
        <parameter id='priority' type='int' value='0' />
      </initbus>
    </connection>
    ...
  </allocation>
  ...

  <!-- Allocations for the control application -->
  <allocation application='control'>
    <connection id='0'>
      <channel type='request'>
        <from ip='NIx0y0n0_cfgbus_rmt_shell' port='psi' />
        <to ip='NIx0y0n1_cfgbus_pt_shell' port='pst' />
        <parameter id='drate' type='double' value='6e-06' />
      </channel>
      <channel type='response'>
        <from ip='NIx0y0n1_cfgbus_pt_shell' port='psi' />
        <to ip='NIx0y0n0_cfgbus_rmt_shell' port='pst' />
        <parameter id='drate' type='double' value='1e-06' />
      </channel>
    </connection>
    ...
  </allocation>
</allocations>
```

Listing 6 Allocation of bus ports

Note that this does not impair the functional scalability, looking only at the outcome and not the run-time. The complexity is reduced by using a path-selection algorithm based on Dijkstra, as proposed in [30], but this on the other hand compromises the quality of the solution and might cause the allocation to fail. For contemporary systems, with hundreds of IPs and connections, the proposed algorithms are sufficient, but other approaches might be necessary as the problem size grows.

After the final allocation step, every connection is allocated an NI on the initiator and target side of the network and the resulting allocation of slots and paths adheres to the channel's throughput and latency requirements. The interconnect hardware and software can thus be instantiated. However, the allocation algorithm takes a network-centric approach and excludes the effects the buses and IPs (and also flow control between these components) have on the end-to-end temporal behaviour. We return to this in Sect. 3.4 where we also

```
<architecture id='85500_inspired'>
  <!-- Everything from Listing 5 -->
  ...

  <!-- NIs -->
  <ip id='NIx0y0n0' type='NI'>
     ...
    <port id='U2_s1_shell_ps_in' type='Target' protocol='FIFO_AE'>
      <parameter id='iqsz' type='int' value='255' />
      <parameter id='portid' type='int' value='1' />
    </port>
    <port id='U2_s1_shell_ps_out' type='Initiator' protocol='FIFO_AE'>
      <parameter id='oqsz' type='int' value='255' />
      <parameter id='portid' type='int' value='1' />
    </port>
    <port id='U2_s2_shell_ps_in' type='Target' protocol='FIFO_AE'>
      <parameter id='iqsz' type='int' value='255' />
      <parameter id='portid' type='int' value='2' />
    </port>
    <port id='U2_s2_shell_ps_out' type='Initiator' protocol='FIFO_AE'>
      <parameter id='oqsz' type='int' value='255' />
      <parameter id='portid' type='int' value='2' />
    </port>
  </ip>
  ...

  <!-- Connects -->
  <connect>
     <endpoint ip='vliw_instr_shell' port='psi' />
     <endpoint ip='NIx0y0n1' port='vliw_instr_shell_ps_in' />
  </connect>
  <connect>
     <endpoint ip='U2_s1_shell' port='psi' />
     <endpoint ip='NIx1y1n1' port='U2_s1_shell_ps_in' />
  </connect>
  ...
</architecture>
```

Listing 7 Architecture with NI mapping

demonstrate how to incorporate the end points (shells, buses, IPs, etc.) and dimension the NI buffers using dataflow analysis techniques.

3.3 Instantiation

The instantiation, corresponding to Step 3 in Fig. 2, takes its starting point in a complete specification of the architecture and resource allocation, and addresses the challenges involved in turning the specification into a hardware and software realisation. Turning high-level specifications into HDL and C code is a tedious and error prone job. The instantiation flow greatly simplifies the task by fully automating this process. We see examples of the automation in later sections where *entire system instances are created in a matter of hours*.

To instantiate the hardware, the appropriate blocks must be created and interconnected. Many blocks are parametrisable and require the appropriate values to be set. In addition to the instantiation of individual library modules, such as buses and routers, these modules must also be connected according to the architecture description. On the network and interconnect level we thus have to generate instance-specific hardware. All of this is done

```
<allocations>
  <!-- Allocations for user applications -->
  <allocation application='storage'>
    <connection id='0'>
      <channel type='request' slots='0' path='3 2'>
        <from ip='U2_s1_shell' port='psi' />
        <to ip='mem_ddr_p1_bus_pt0_shell' port='pst' />
        <parameter id='drate' type='double' value='109.375' />
      </channel>
      <channel type='response' slots='0 1' path='4 2'>
        <from ip='mem_ddr_p1_bus_pt0_shell' port='psi' />
        <to ip='U2_s1_shell' port='pst' />
        <parameter id='drate' type='double' value='150' />
      </channel>
      <initbus bus='mem_ddr_p1_bus' port='pt0'>
        <parameter id='numerator' type='int' value='3' />
        <parameter id='denominator' type='int' value='51' />
        <parameter id='initialCredits' type='int' value='51' />
        <parameter id='priority' type='int' value='0' />
      </initbus>
    </connection>
    ...
  </allocation>
  ...

  <!-- Allocations for the control application -->
  <allocation application='control'>
    <connection id='0'>
      <channel type='request' slots='0' path='3'>
        <from ip='NIx0y0n0_cfgbus_rmt_shell' port='psi' />
        <to ip='NIx0y0n1_cfgbus_pt_shell' port='pst' />
        <parameter id='drate' type='double' value='6e-06' />
      </channel>
      <channel type='response' slots='0' path='2'>
        <from ip='NIx0y0n1_cfgbus_pt_shell' port='psi' />
        <to ip='NIx0y0n0_cfgbus_rmt_shell' port='pst' />
        <parameter id='drate' type='double' value='1e-06' />
      </channel>
    </connection>
  </allocation>
</allocations>
```

Listing 8 Allocation of network paths and time slots

by a tool (`vhdlinst`) that also generates a set of auxiliary files that are required by the ASIC/FPGA tools for e.g. HDL elaboration, simulation and RTL synthesis.

The hardware is of no use without the appropriate (host) software. The resource allocations are thus translated into source code (`cinst`) and linked with the appropriate run-time libraries that enable a host processor to open and close connections [28]. At run-time, the host then instantiates these allocations as a result of trigger events [56]. The run-time instantiation, in its current form, limits the interconnect scalability as it uses a centralised host. However, the proposed run-time libraries do not inherently limit the scalability, and it is possible to distribute the work across multiple hosts.

In addition to the instantiation tools, the design flow also includes an area-estimation tool that quickly shows the total area requirements of the interconnect based on the area models described in [29]. The important observation about the silicon area is that the interconnect is dominated by the NIs, that in turn are dominated by their buffers. Buffer sizing is therefore of utmost importance, as discussed next.

3.4 Verification

The verification constitutes Step 4 in Fig. 2 and serves as a robust *design-rule checker* based on the given architecture and allocation XML files, which could have been generated, provided or edited by hand.

Even in the case where the files are automatically generated by the flow, the allocation of communication requirements only covers the network and excludes the effects of end-to-end flow control. When NI ports are added as part of the dimensioning and allocation, all buffers are given a default size, and until now we have assumed that the buffers are sufficiently large. If this is not the case, the application performance suffers, and conversely, if the buffers are unnecessarily large they waste power and area [11, 21, 35, 65]. In addition to the network, also the clock domain crossings, shells, atomisers and buses affect the performance and must be considered. Furthermore, the communication requirements are merely a result of higher-level application requirements, and it remains to verify that the applications have the expected functional and temporal behaviour when they are executed on the system instance, i.e. taking the interdependencies between the applications, the IPs, and the interconnect into account. Thus, for an existing interconnect instance, we must be able to *determine the temporal behaviour* of the applications mapped to it, given fixed NI buffer sizes. On the other hand, if we are designing an interconnect specifically for a set of applications, then it is desirable to *determine sufficiently large* NI buffers.

The verification (`buffdim`) is performed using dataflow models of the interconnect [35] and the IPs. The dataflow models bound the combined behaviour of the interconnect architecture and resource allocation, even for data-dependent applications [77]. Currently our design flow assumes a homogeneous analysis of all applications, using dataflow analysis. However, as discussed in Sect. 2, it is possible to use other buffer-sizing approaches where appropriate, e.g. stochastic traffic models [13, 68]. This diversity enables the designer to capture a wide range of application behaviours and requirements, and is made possible by the composability of our interconnect [33]. Listing 9 shows the resulting architecture specification, highlighting the changes compared to Listing 7. After the buffer dimensioning, the interconnect hardware and software can be instantiated anew. The verification thus completes the interconnect design flow, and we now continue to design interconnects for our two case studies.

4 Digital TV

We now put the proposed design flow to use with an example inspired by NXP's PNX85500, which is a complete one-chip digital TV, aimed at the cost-sensitive midrange market. The PNX85500 delivers multi-standard audio decoding and multi-standard analogue and digital video decoding. The front-end video processing functions, such as DVB-T/DVB-C channel decoding, MPEG-2/MPEG-4/H.264 decoding, analog video decoding and HDMI reception, are combined with advanced backend video picture improvements. One of the key differentiators is the video enhancement algorithms, e.g. Motion Accurate Picture Processing (MAPP2), Halo reduced Frame Rate Conversion and LCD Motion Blur Reduction. The MAPP2 technology provides state-of-the-art motion artifact reduction with movie judder cancellation, motion sharpness and vivid colour management. High flat panel screen

```
<architecture id='85500_inspired'>
  <!-- Everything from Listing 7 -->
  ...

  <!-- NIs -->
  <ip id='NIx0y0n0' type='NI'>
    ...
    <port id='U2_s1_shell_ps_in' type='Target' protocol='FIFO_AE'>
      <parameter id='iqsz' type='int' value='67' />
      <parameter id='portid' type='int' value='1' />
    </port>
    <port id='U2_s1_shell_ps_out' type='Initiator' protocol='FIFO_AE'>
      <parameter id='oqsz' type='int' value='65' />
      <parameter id='portid' type='int' value='1' />
    </port>
    <port id='U2_s2_shell_ps_in' type='Target' protocol='FIFO_AE'>
      <parameter id='iqsz' type='int' value='67' />
      <parameter id='portid' type='int' value='2' />
    </port>
    <port id='U2_s2_shell_ps_out' type='Initiator' protocol='FIFO_AE'>
      <parameter id='oqsz' type='int' value='66' />
      <parameter id='portid' type='int' value='2' />
    </port>
  </ip>
  ...
</architecture>
```

Listing 9 Architecture with minimal NI buffer sizes

resolutions and refresh rates are supported, including 1366 × 768 and 1920 × 1080 at 100 Hz/120 Hz. This is coupled with multiplexing of multiple high-definition streams, multiple video outputs and multiple audio outputs. The SoC supports a rich set of I/O standards (e.g. PCI, USB, Ethernet, UART) and opens many possibilities for new TV experiences with IPTV and Video On Demand (VOD).

We consider the processor cores and IP blocks given, and a schematic architecture is shown in Fig. 3 [41]. As shown in the figure, the SoC contains multiple programmable processors with caches: a MIPS for control, three TriMedia processors for audio/video, one MIPS DSP, and a 8051 processor for power management and control. The TriMedia VLIWs are used for computation (and communication) intensive image processing, e.g. Halo Reduced Frame Rate Conversion, with on-the-fly compression of video traffic to and from the off-chip SDRAM. In addition to the processors, the SoC contains a large number of function-specific hardware IP cores. Those accelerator IPs are used for e.g. audio and video decoding, video scaling and enhancement, graphics, and I/O (LVDS, HDMI, Ethernet, USB, Flash, SPI, PCI and I2C). Finally, the SoC has a range of peripherals for clocking, debug, timers, etc. As seen in the block diagram, the SoC is built around a physically centralised memory architecture. Two central off-chip memory controllers connect with two large external SDRAMs, with a 16-bit and 32-bit wide data path respectively. The SoC contains 11 chiplets and five major frequency domains, from 27 MHz (many IPs) to 200 MHz (majority of IPs) to over 450 MHz (processors). The complete XML description of the IP architecture occupies roughly 250 lines, with a subset shown in Listing 1.

A major design challenge is the on-chip communication, with 110 memory-mapped ports distributed across the IPs (the majority read-only or write-only), contributing to almost 100 logical connections. Predictability is a design constraint. A majority of the connections have relaxed latency requirements (>1000 ns), as the function-specific hardware IP cores make use of prefetching read accesses and posted write accesses. The deep pipelining makes these

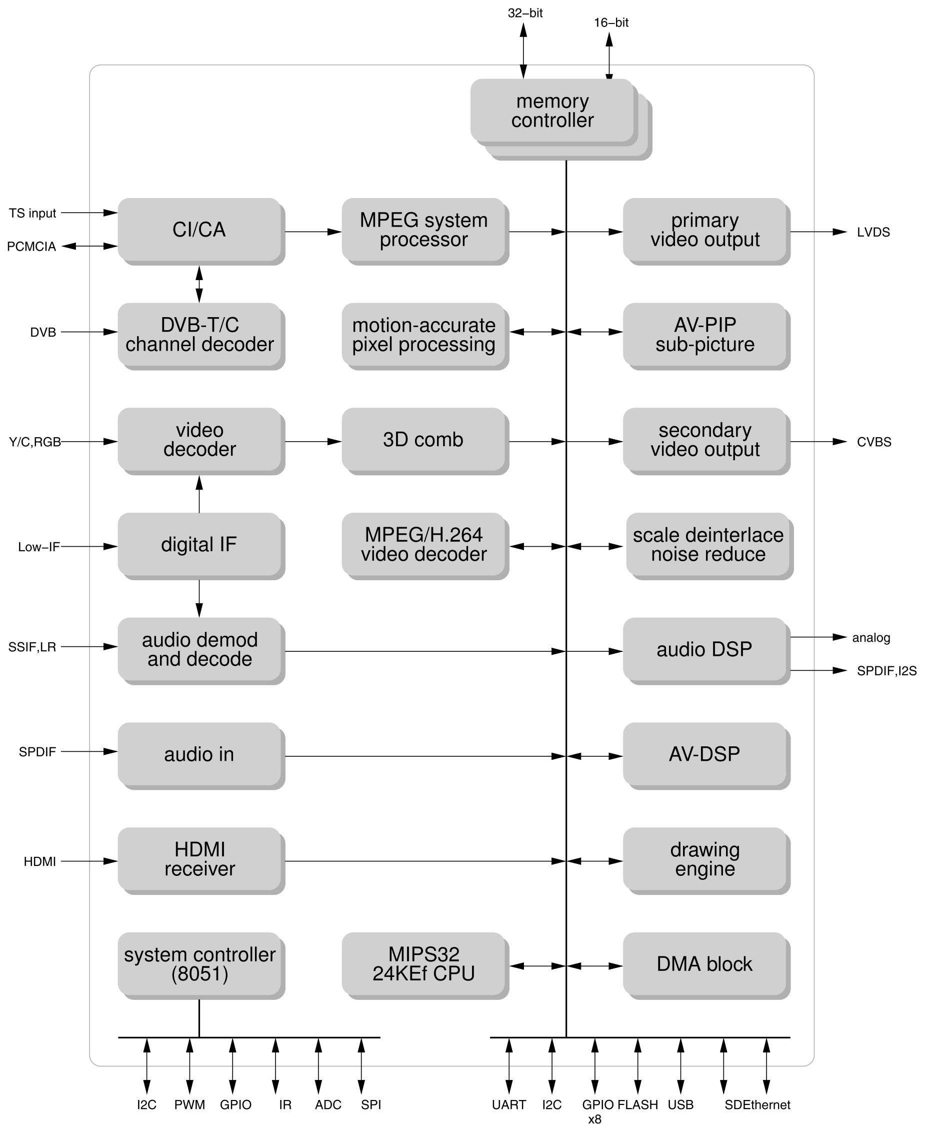


Fig. 3 Digital TV SoC block diagram [41]

connections latency tolerant. Processor instruction fetches (from cache misses), on the other hand, have very low latency requirements ($<$100 ns) and have relatively small transaction sizes, but require a limited throughput. Some accelerators, e.g. the display controller are also latency constrained. With the most critical low-latency connections implemented through direct connections to the memory controller, and several low-throughput peripheral connections merged, this translates to 53 physical ports, and 45 logical interconnections for the on-chip interconnect. A part of the 260 line specification is shown in Listing 2.

Figure 4 shows the distribution of throughput requirements for the 45 connections, with a total of 4402 Mbyte/s (1461 Mbyte/s write, 2941 Mbyte/s read). Note that this is the

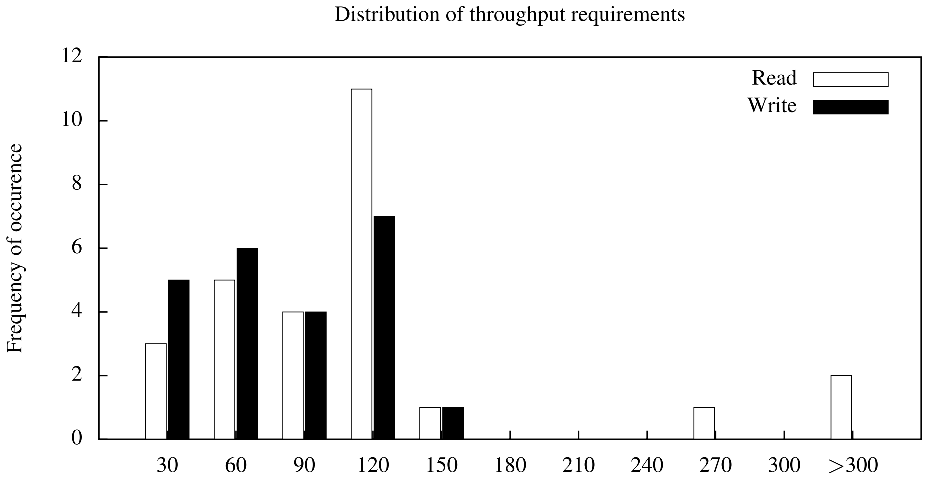


Fig. 4 Write and read throughput distribution (number of connections vs. throughput in Mbyte/s)

data throughput for the memory-mapped communication, not accounting for commands and addresses, byte masks, etc. Burst sizes vary from 64 bytes through 128 and 256 bytes, with the majority of the IPs using the latter (with a mean of 240 bytes). Including the additional address and command signals results in a raw throughput of 4560 Mbyte/s to be delivered by the network.

It is worth noting that this particular case study does not make use of the reconfigurability offered by the interconnect. The reason is that the specification is determined without assuming any such functionality (which is unique for NoCs). All connections are thus assumed to be concurrent, and a re-design with reconfigurability in mind could benefit from mutually exclusive applications and connections.

4.1 Experimental results

As discussed in Sect. 3.1, topology selection is not an automated part of the design flow. The tools do, however, aid in translating a high-level topology description into an XML specification. For this example we chose a 2 by 3 mesh network (six routers) with two NIs connected to all routers but one (11 NIs in total). The router arity is thus kept at five for a single router and four for the remaining ones. This topology offers one NI for each chiplet. The network is clocked at 533 MHz, which is the highest IP clock frequency.

The latency requirement, as discussed in Sect. 3.2, for low-latency connections is specified to 50 ns (not including the memory scheduler and controller). Given the requirements of the connections and the physical topology, a resource allocation for the single use-case is found with a slot table size of 24. The resulting allocation XML file occupies just over 1000 lines and the corresponding architecture description is almost 4000 lines, all automatically generated by the design flow *in a few minutes*. As the final part of the dimensioning, the NI buffers are sized using the approach from Sect. 3.4, and the distribution of the NI queue sizes is shown in Fig. 5. The large burst sizes lead to increased buffering in the NIs, and a total of 2249 words (of 37 bits) in output queues and 2232 words in input queues (3060 on the initiator side and 1621 on the target side). In total the NI buffers store roughly 24 kbyte, and the buffers have a big influence on the area requirements which we evaluate next.

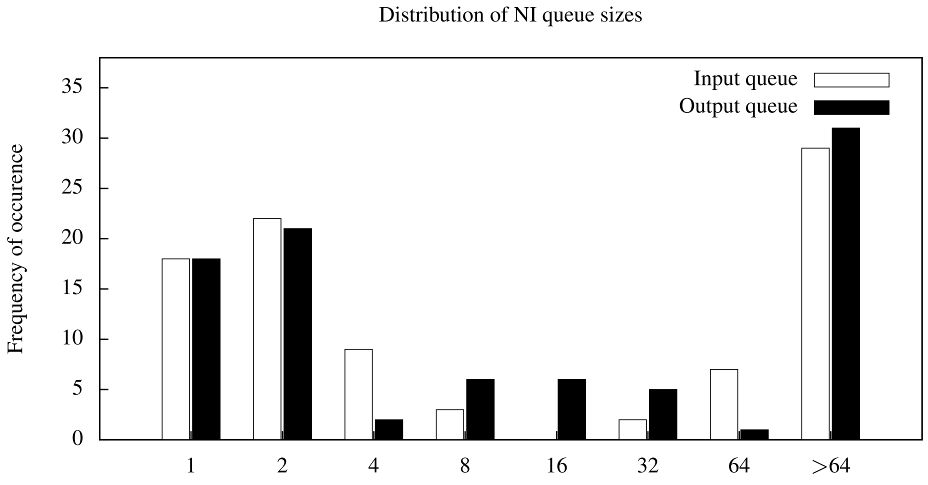


Fig. 5 NI queue size distribution in 37-bit words

Table 1 Area requirements

Module	Instances	Area (mm^2)	Discrepancy
Router	6	0.08	+10%
NI	11	0.41	−10%
Shell	54	0.23	−8%
Bus	6	0.08	+9%

Using the same synthesis setup as described in [29], i.e. Cadence Ambit with Philips 90 nm Low-Power technology libraries, the synthesised interconnect occupies a cell area of 5.5 mm^2 with 4.7 mm^2 (85%) in NI buffers. The large fraction in buffers is due to the register-based FIFO implementation, and with FIFOs based on dual-ported SRAMs the area is reduced to roughly 2.6 mm^2 with 1.7 mm^2 (65%) in buffers. Further reduction is possible with dedicated FIFOs [76], which results in a total area of 1.6 mm^2 with 0.7 mm^2 (44%) in buffers. A break down of the remaining area is shown in Table 1, including a comparison with area estimation models based on the synthesis of the individual blocks in [29]. As seen in the table, the area estimation is within 10% of the result reported by the synthesis tool. Note that the area reported is cell area only, thus before place and route, and clock-tree insertion.

To get further insight into the area requirements and the power consumption of the interconnect instance, we synthesise the design using Synopsys Design Compiler Ultra in topographical mode. The synthesis uses TSMC's low-power 65 nm libraries (9 track with a raw gate density of 0.85 Mgate/mm^2) employing mixed Voltage Threshold (VT) synthesis with both high- and low-VT libraries. The synthesis is performed on a flattened design with multi-stage clock gating enabled, using integrated clock gates from the technology library. The power minimisation is driven by activity traces captured in RTL simulation using traffic generators to mimic the IP behaviour.

The total area after synthesis is estimated at 2.5 mm^2 (9781 cells). Once again the FIFO implementation used is the register-based FIFO, leading to a dominant part of the area spent in buffering. The area of the buffers is, however, reduced by the clock gating as many multiplexers in the FIFOs can be replaced. A total of 7500 clock gating elements are inserted, resulting in 97% of the 251588 registers in the design being gated. At a global (worst-case) operating voltage of 1.08 V the cell power is estimated at 79.5 mW and the net switching power at 43.3 mW. The total dynamic power, under worst-case conditions, is 122.8 mW and the leakage power is estimated at 3.2 mW (or roughly 2.5% of the total power consumption). Roughly 22% of the dynamic power is spent in the clock tree. We can conclude that the low-power libraries, in combination with mixed-VT synthesis and leads to an implementation where leakage-minimisation techniques like power gating, which is costly in level shifters, *are not worthwhile*.

Next, we evaluate the behaviour of the generated interconnect hardware and software by simulating an instance of the entire system using traffic generators to mimic the IP behaviour. Instrumentation is (automatically) inserted in SystemC and VHDL testbenches respectively to observe the utilisation of the different modules, the latency of individual packets in the network, and the end-to-end latency and throughput between the IPs. The cycle-accurate SystemC simulation executes with a speed of roughly 50 kcycles/s on a modern Linux machine (1 ms of simulated time in less than 20 s). The corresponding VHDL testbench is roughly 200 times slower, with 0.2 kcycles/s. Simulation of the synthesised netlist is again a factor 100 slower, reducing the speed to roughly a cycle per second.

First, looking at the critical low-latency connections (specified to 50 ns as previously mentioned) they consistently have an average network latency of 19 ns and a maximum across all the connections of 42 ns. For a specific low-latency connection, the end-to-end latency for a 128 byte read transaction, including the memory controller and memory access, is 280 ns on average and 308 ns for the observed worst case (corresponding to 190 processor cycles at 533 MHz). All the 45 connections have an observed throughput and latency that *meets the user-specified requirements*, and this behaviour is consistent in both SystemC and VHDL simulation. Comparing SystemC, behavioural VHDL and netlist simulation we see slight differences in the statistical traffic profile. The reason for the discrepancies is the use of different traffic generators for the SystemC and VHDL testbenches. The results, however, looking at the end-to-end latency and throughput, are within $\pm 3\%$.

Next, we look at the utilisation of the interconnect, as observed during simulation. The TDM-based arbitration of the network requires low-latency connections to reserve a larger number of appropriately spaced time slots. As a consequence, these slots are often unused. The utilisation does hence not reflect the reserved slots, but rather the slots that are actually used at run time. We first look at the ingress and egress links of the network, i.e. links that connect an NI to a router. Since the NI is comparatively more expensive, the chosen topology aims to minimise the number of NIs. The ingress and egress links are therefore more likely to have a high utilisation compared to the average over the entire network. For the TV SoC, the average NI link utilisation is 49%, with a minimum of 27% and a maximum of 64%. This is to be compared with an overall network link utilisation of 22%. The link with the lowest utilisation is only used 12% of the time. The relatively low utilisation of the network stresses the importance of using clock gating as part of the logic synthesis. The register-level and module-level clock gating ensures that the power consumption is proportional to the usage of the gates rather than the sheer number of gates. As already mentioned, the total interconnect power consumption, under worst-case conditions, is estimated at 125 mW.

4.2 Scalability analysis

Having evaluated our proposed interconnect in the context of the contemporary digital TV SoC, we now continue to look at how the interconnect scales with the requirements, and how

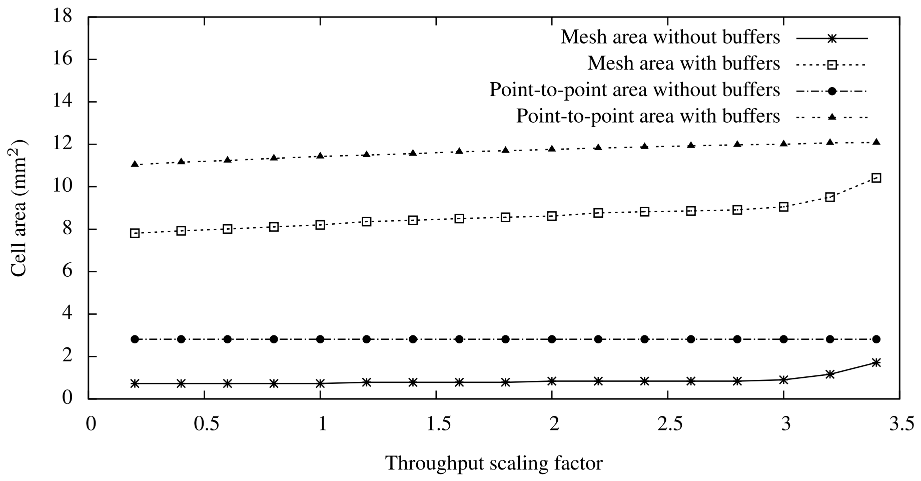


Fig. 6 Scalability of throughput requirements

suitable it is for future generation multimedia applications. As discussed in [41], the number of latency-critical connections has stayed low and fairly constant during the past six generations of TV Socs in the Nexperia platform. The total bandwidth, however, and the number of latency-tolerant function-specific hardware IPs grows over time. These key trends are a result of stable functional subsystems being moved to latency-tolerant implementations as the functionality matures. The latter is a necessary trend to fit more functionality on a single SoC and still be able to serve the latency-critical connections within acceptable bounds. The on- and off-chip throughput is scaling with the generations, but the latency (in absolute time) is fairly constant, leading to growing memory access latencies (measured in processor cycles). With a growing number of latency-critical connections the communication pattern would be *inherently non-scalable*.

Our first scaling experiment fixes the IPs and their 45 connections, and only scales the throughput requirements (leaving the latency untouched). For each design point, we re-run the entire interconnect design flow using the new (scaled) requirements. For each point, we evaluate two different network topologies. First, a minimum mesh, determined by a simple loop that increases the dimensions until a feasible resource allocation (one that satisfies all requirements) is found. For simplicity, the number of NIs is fixed at three per router. Second, a point-to-point topology where each connection is mapped to two unique NIs interconnected by a router (to allow for instantiation of the resource allocation during run-time). A maximum of 32 TDM slots is allowed in both cases. Once a resource allocation is found, we determine sufficiently large buffer sizes (also verifying the performance bounds using dataflow analysis) and estimate the required area using the models from [29]. Thus, for every point we determine two complete hardware and software interconnect instances that are guaranteed to satisfy all the requirements.

Figure 6 shows the area, both with and without buffers, for the two topologies as the throughput requirements are scaled from 0.2 to 3.4. The area estimates once again assume a 90 nm process and register-based FIFOs and are thus overly conservative. However, as we are interested in the scaling the absolute numbers are not critical. What is more interesting is the trend as the scaling factor is increased. For scaling factors below three, the total area is growing linearly for both topologies, with the area of the mesh being roughly

25% smaller than the point-to-point topology (75% smaller when excluding the buffers). Although not visible in the figure, the mesh area excluding the buffers is increasing in a stair case as the number of routers and NIs grow. Increased throughput leads to different mapping and routing decisions as the connections are balanced on a larger interconnect. As expected, the point-to-point topology area is constant when the buffers are not considered. With scaling factors above three, the mesh area is increasing faster, approaching the area of the point-to-point topology, and after 3.4 no feasible resource allocation is found for either of the two topologies. This is due to the inherent sharing of the IP ports, i.e. the memory controllers, where the scaled requirement eventually saturate the port capacity. Thus, the point of contention is pushed to the IPs ports and *there exist no possible topologies and resource allocations*, for any NoC architecture or design flow.

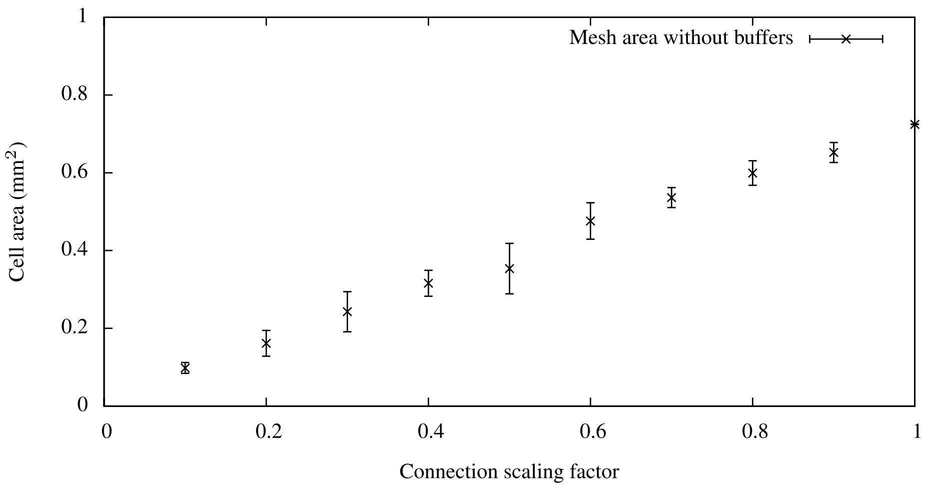
We have seen that the interconnect scales linearly with the throughput requirements when the number of connections is kept constant. Now we look at how the interconnect scales with the number of latency-tolerant connections. Rather than extrapolating a next-generation TV SoC, we randomly select a subset of the connections of the original example, and do so for fractions ranging from 0 to 1 in increments of 0.1. For each design point (belonging to a specific fraction) we repeat the random selection 10 times. As a result, for each point we re-run the entire design flow 10 times with (potentially) different subsets of the communication specification. For each run we determine the minimum mesh as before. After finding a feasible resource allocation the buffers are sized and the interconnect area estimated for each one of the 100 instances.

Figure 7 shows the area estimates with and without buffers, with each point showing the average of the 10 runs together with the standard deviation. Once again, the absolute numbers are not particularly important as they depend heavily on the chosen technology node and buffer implementation. What is important is that both Figs. 7(a) and 7(b), show a clear linear trend. This suggests that the interconnect architecture and design flow is providing functional scalability and thus able to accommodate a growing number of connections with a roughly constant cost/performance ratio.

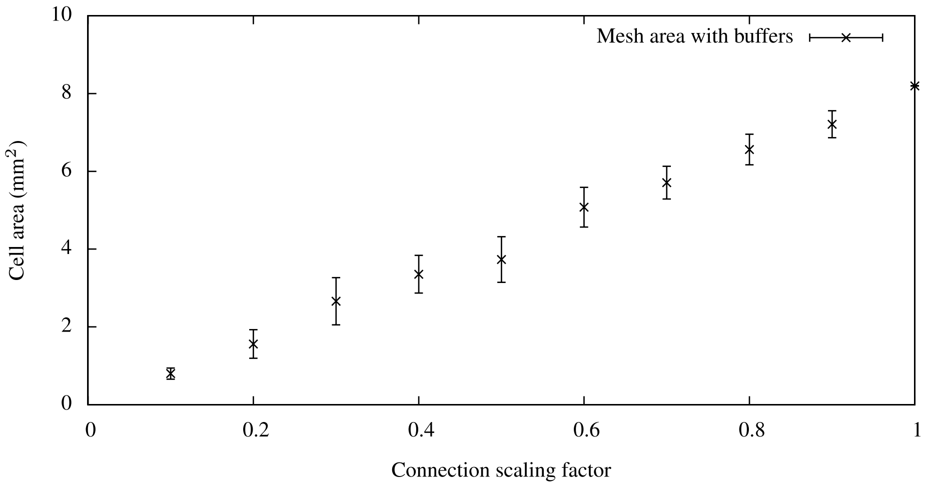
5 Automotive radio

Our second example is a software-defined radio SoC, aimed at the automotive industry. Although mobility, connectivity, security and safety drive the automotive industry, there is an accelerated adoption of consumer features in cars, with home entertainment also in your car. The main applications include high quality, real-time, interactive audio and video, with standards like DAB, DVB-H, DVB-T, T-DMB, HD Radio etc. While delivering a true multimedia experience, the SoC must also meet automotive quality demands. Due to the nature of the SoC, there is an essential need for a powerful and flexible multi-standard, multi-channel software-defined radio, with efficient channel decoder processing.

Similar to the digital TV SoC, we consider the processor cores and IP blocks given. The example is inspired by a SoC that contains three programmable processors with caches: an ARM Cortex for control, and two Embedded Vector Processors [73] (EVP). The EVPs constitute the core of the software-defined radio. In addition to the processors, the SoC contains a large accelerator subsystem. The accelerators are involved in signal-processing and I/O (several DACs and ADCs, DMA, and USB). In contrast to the digital TV SoC, this system is not making use of a physically centralised memory architecture, as it has many on-chip (more than 10) SRAMs in addition to the external SDRAM. As we shall see, the result is *less inherent contention* in the communication patterns. The SoC contains 7 chiplets and



(a) Area without buffers



(b) Area with buffers

Fig. 7 Scaling the number of connections

9 clock domains, ranging from 100 MHz to 350 MHz with the majority of the IPs running at either 100 MHz or 200 MHz.

Communication is central to the SoC with 68 IP ports using both memory-mapped and streaming communications with a total of 52 logical connections. Many of the IPs do not use posted/pipelined transactions and their communication is therefore latency-critical. In contrast to the digital TV it is thus not only cache traffic that is latency critical. Due to the real-time nature of the application, predictability is crucial for all on-chip communication. After merging a few low-throughput connections to peripherals, the on-chip interconnect must accommodate 43 ports (distributed across 12 IPs), with 32 logical interconnections. Roughly half of these connections have latency requirements below 100 ns.

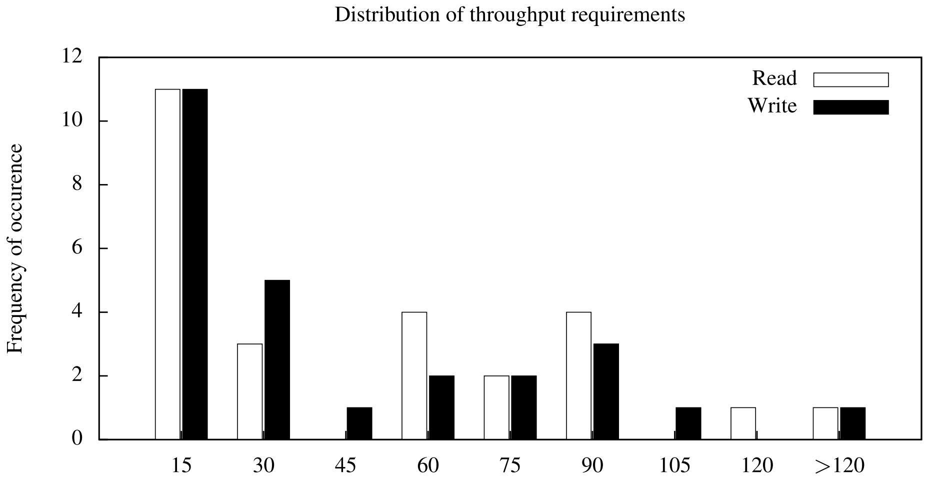


Fig. 8 Write and read throughput distribution (number of connections vs. throughput in MByte/s)

Figure 8 shows the distribution of throughput requirements, with a total of 2151 Mbyte/s (997 Mbyte/s write, 1154 Mbyte/s read). Including command and address this amounts to 4183 Mbyte/s to be delivered by the network. Note that this is almost the double due to the small burst sizes, varying between 4 and 128 bytes (with a mean of 31 bytes). Compare this to the digital TV where we almost exclusively have large bursts, making the raw throughput requirement of the two case studies very similar. Although the throughput requirements are not much different, we shall see that the large difference in burst size has a tremendous impact of the NI buffers, and consequently the interconnect area and power.

Similar to the TV SoC, this case study does not exploit the support for multiple use-cases.

5.1 Experimental results

For the automotive SoC, we use a 3 by 3 mesh (nine routers) with two NIs connected to the corner routers, one NI for the remaining peripheral routers, and no NI for the central router (12 NIs in total). The router arity is thereby kept low and the topology is still large enough to enable the IPs to be placed around the interconnect, grouped according to chiplets and clock domains. For this case study, the network is clocked at 400 MHz which is double the frequency of most IPs.

Given the requirements, an interconnect instance is generated with a slot table size of 21 and the buffers are sized using the proposed design flow. The distribution of NI queue sizes is shown in Fig. 9, with a total of 442 words in input queues, and 587 words in output queues (453 on the target side and 576 on the initiator side). The total capacity is roughly 5 kbyte. Compared to the TV SoC this is roughly *four times less buffering per connection*, largely due to the big difference in average burst size.

The difference in buffer size also has a big impact on the interconnect area. Using the same 90 nm libraries and the same setup as for the TV SoC, the total cell area for the automotive SoC interconnect is 2.13 mm^2 with 1.41 mm^2 (66%) in register-based NI buffers. With dual-ported SRAMS this is reduced to 1.22 mm^2 with 0.5 mm^2 (40%) in buffers, or as little as 0.9 mm^2 with 0.22 mm^2 (23%) in buffers when implemented using dedicated FIFOs [76]. An area break down is shown in Table 2, including both the results from synthesis and the high-level area-estimation models. Just as for the TV SoC, the area estimation is

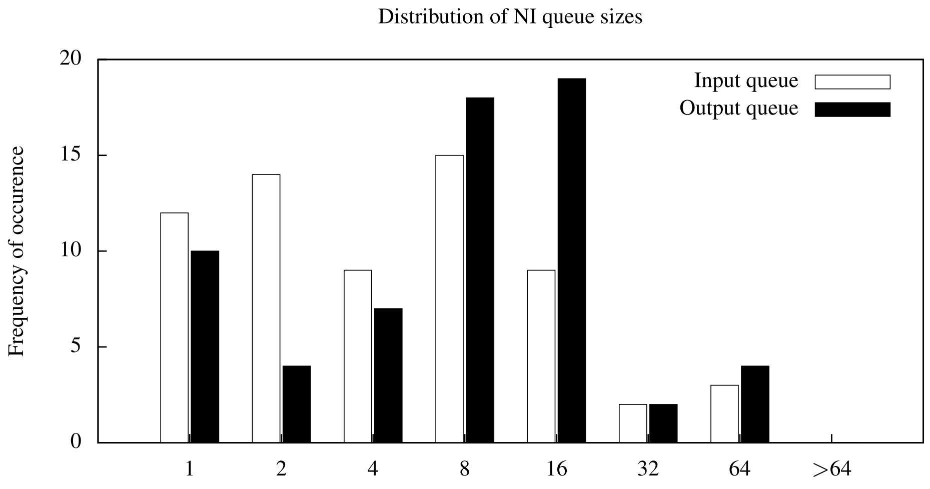


Fig. 9 NI queue size distribution in 37-bit words

Table 2 Area requirements

Module	Instances	Area (mm^2)	Discrepancy
Router	9	0.13	+10%
NI	12	0.32	−8%
Shell	43	0.18	0%
Bus	8	0.07	−4%

within 10%, suggesting the models can be used reliably for early design-space exploration and evaluation.

We synthesise also this interconnect using Synopsys Design Compiler Ultra, aiming to minimise the power consumption. The details are the same as described in Sect. 4. The total area after synthesis is estimated at 0.78 mm^2 (4784 cells) and 90% of the 76474 registers are clock gated. At worst-case conditions, the cell power is estimated at 41.2 mW and the net switching power at 15.4 mW, resulting in a total dynamic power consumption of only 57 mW. The leakage is reported to be less than 1 mW, once again demonstrating the abilities of the low-power libraries and mixed-VT synthesis. Roughly 27% of the total power is spent in the clock tree.

Similar to the TV SoC, we perform SystemC and HDL simulations on the instantiated interconnect. Once again, all throughput and latency requirements are satisfied, and this behaviour is consistent for all the different instantiations. Looking at the utilisation during simulation, the automotive SoC has an average NI link utilisation of 30% with a minimum of 7% and a maximum of 75%. The low utilisation of the NI links is due to the very strict latency requirements which limits the amount of sharing. The overall network link utilisation is also low, only 20%. As previously discussed, the relatively low utilisation highlights the importance of clock gating, as used in our experiments.

5.2 Scalability analysis

We continue our evaluation with two different scaling experiments, similar to what is described for the TV SoC in Sect. 4. First, scaling the throughput requirements between 0.2

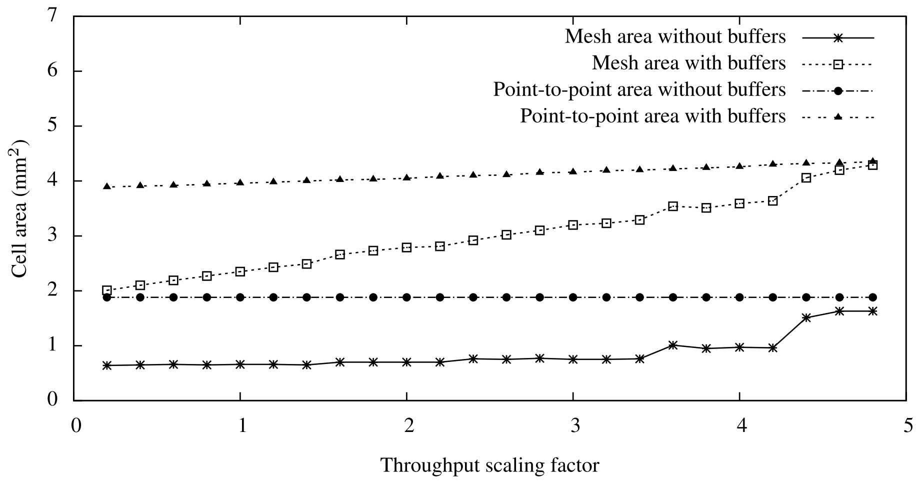


Fig. 10 Scalability of throughput requirements

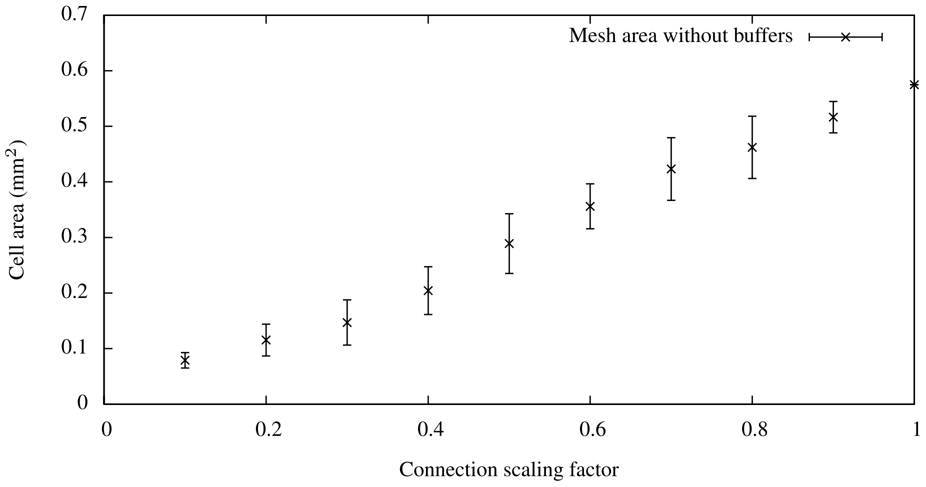
and 4.8, we compare the area of a minimum mesh and a point-to-point topology. Figure 10 shows the result, both with and without buffers. The heuristics used in the mesh sizing results in an area that is growing in a stair case, which is now visible as the total area is less dependent on the buffers. The total area is growing linearly for both topologies until they saturate at a scaling factor of 4.8 where the requirements can no longer be allocated. At this point the mesh area is almost equivalent to the point-to-point topology, using a 5 by 4 mesh with 3 NIs per router. Comparing the results with the TV SoC we clearly see the consequences of having a more distributed communication pattern (more memories) as the throughput can be scaled further (4.8 compared to 3.4). Once again this factor is an inherent limit caused by the sharing of the IP ports and not influenced by the choice of a NoC architecture or design flow.

Next we evaluate the scalability with respect to the number of connections. In contrast to the TV SoC we now include all connections (latency critical and latency tolerant) in the selection. Again we look at 10 different design points and run the entire design flow 10 times per point. Figure 11 shows the area with and without buffers. Although there is a considerable variation in the results, the linear trend is clear. We have thus shown that also for this example, with a larger number of latency-critical connections, but a more distributed communication pattern, the design flow offers functional scalability.

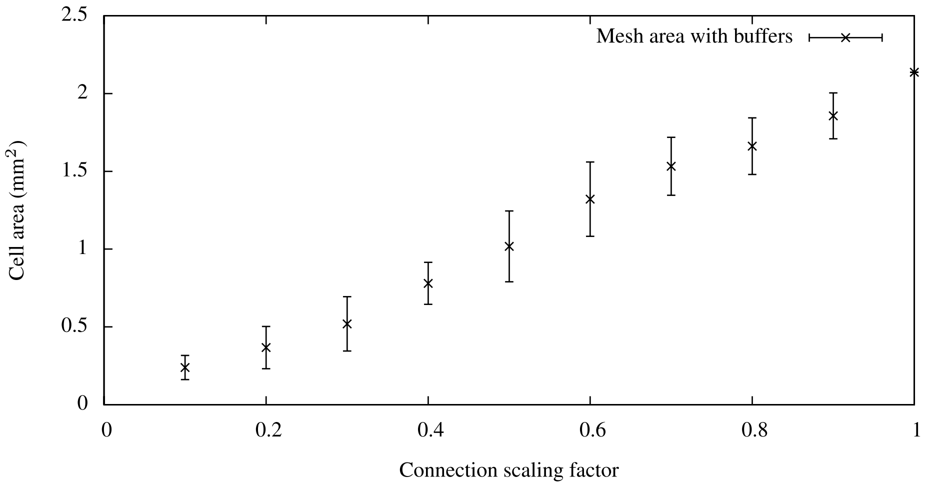
Comparing the results of the two examples we can conclude that the area and power is very much influenced by the temporal requirements and the burst size used by the IPs. However, despite the differences in requirements and behaviours, the proposed design flow is able to scale the two examples, both in terms of requirements and number of connections, for several hundreds of complete system instances.

6 Conclusions

Systems on Chip (SoC) integrate growing number of applications, often with real-time requirements. This calls for an on-chip interconnect that is able to provide throughput and latency bounds to an increasing number of logical connections. Doing so requires physical,



(a) Area without buffers



(b) Area with buffers

Fig. 11 Scaling the number of connections

architectural as well as functional scalability. While many Networks on Chip (NoC) address the physical and architectural scalability, the functional scalability depends heavily on the interconnect design flow which is often overlooked.

In this work we describe a fully operational design flow used to generate application specific NoC instances for multiple real-time applications. We show how to go from requirement specification to a complete hardware and software instance, using two large-scale industrial case studies. We shown the large impact burst sizes have on the interconnect area and power consumption, and quantify the influence latency-critical connections have on the network utilisation. The relatively low utilisation (around 25%) stresses the importance of clock gating in the logic synthesis to minimise dynamic power. We show that low-power mixed-voltage-threshold libraries lead to leakage power being less than 3% in a 65 nm technology. We evaluate the scalability of the interconnect using the two case studies as our starting point, and show that the proposed interconnect is able to accommodate a growing number of requirements with a constant cost/performance ratio.

References

1. AXI (2003) AMBA AXI protocol specification. ARM Limited
2. Bartic T, Desmet D, Mignolet JY, Marescaux T, Verkest D, Vernalde S, Lauwereins R, Miller J, Robert F (2004) Network-on-chip for reconfigurable systems: from high-level design down to implementation. In: Proc FPL
3. Beigne E, Clermidy F, Vivet P, Clouard A, Renaudin M (2005) An asynchronous NOC architecture providing low latency service and its multi-level design framework. In: Proc ASYNC
4. Beigné E, Clermidy F, Miermont S, Vivet P (2008) Dynamic voltage and frequency scaling architecture for units integration within a GALS NoC. In: Proc NOCS
5. Benini L (2006) Application specific NoC design. In: Proc DATE
6. Benini L, de Micheli G (2002) Networks on chips: a new SoC paradigm. IEEE Comput 35(1):70–80
7. Beraha R, Isask’har WI, Kolodny A (2010) Leveraging application-level requirements in the design of a NoC for a 4G SoC. In: Proc DATE
8. Bertozzi D, Jalabert A, Murali S, Tamhankar R, Stergiou S, Benini L, Micheli GD (2005) NoC synthesis flow for customized domain specific multiprocessor systems-on-chip. IEEE Trans Parallel Distrib Syst 16(2):113–129
9. Bjerregaard T, Sparsø J (2005) A router architecture for connection-oriented service guarantees in the MANGO clockless network-on-chip. In: Proc DATE
10. Bjerregaard T, Sparsø J (2005) A scheduling discipline for latency and bandwidth guarantees in asynchronous network-on-chip. In: Proc ASYNC
11. Bjerregaard T, Mahadevan S, Grøndahl Olsen R, Sparsø J (2005) An OCP compliant network adapter for GALS-based SoC design using the MANGO network-on-chip. In: Proc SOC
12. Bjerregaard T, Stensgaard M, Sparsø J (2007) A scalable, timing-safe, network-on-chip architecture with an integrated clock distribution method. In: Proc DATE
13. Bogdan P, Kas M, Marculescu R, Mutlu O (2010) QuaLe: a quantum-leap inspired model for non-stationary analysis of noc traffic in chip multi-processors. In: Proc NOCS
14. Bolotin E, Cidon I, Ginosar R, Kolodny A (2003) QNoC: QoS architecture and design process for network on chip. J Syst Archit 50(2–3):105–128
15. Buttazo GC (1977) Hard real-time computing systems: predictable scheduling algorithms and applications. Kluwer, Dordrecht
16. Chen J, Jone W, Wang J, Lu HI, Chen T (1999) Segmented bus design for low-power systems. IEEE Trans Very Large Scale Integr 7(1):25–29
17. Clermidy F, Bernard C, Lemaire R, Martin J, Miro-Panades I, Thonnart Y, Vivet P, Wehn N (2010) A 477 mw NoC-based digital baseband for MIMO 4G SDR. In: Proc ISSCC
18. Dally WJ, Towles B (2001) Route packets, not wires: on-chip interconnection networks. In: Proc DAC
19. DTL (2002) Device Transaction Level (DTL) Protocol Specification. Version 2.2. Philips Semiconductors
20. Eichenberger A, OBrien J, OBrien K, Wu P, Chen T, Oden P, Prener D, Shepherd J, So B, Sura Z et al (2006) Using advanced compiler technology to exploit the performance of the cell broadband engine architecture. IBM Syst J 45(1):59–84
21. Gangwal O, Rădulescu A, Goossens K, Pestana S, Rijpkema E (2005) Building predictable systems on chip: an analysis of guaranteed communication in the Æthereal network on chip. In: Dynamic and robust streaming in and between connected consumer-electronics devices. Kluwer, Dordrecht
22. Genko N, Atienza D, Micheli GD, Mendias J, Hermida R, Catthoor F (2005) A complete network-on-chip emulation framework. In: Proc DATE
23. González Pestana S et al (2004) Cost-performance trade-offs in networks on chip: a simulation-based approach. In: Proc DATE

24. Goossens K, Gangwal OP, Röver J, Niranjan AP (2004) Interconnect and memory organization in SOCs for advanced set-top boxes and TV—evolution, analysis, and trends. In: Nurmi J, Tenhunen H, Isoaho J, Jantsch A (eds) Interconnect-centric design for advanced SoC and NoC. Kluwer, Dordrecht, pp 399–423, Chap 15
25. Goossens K, Dielissen J, Gangwal OP, González Pestana S, Rădulescu A, Rijpkema E (2005) A design flow for application-specific networks on chip with guaranteed performance to accelerate SOC design and verification. In: Proc DATE
26. Goossens K, Dielissen J, Rădulescu A (2005) The Æthereal network on chip: concepts, architectures, and implementations. IEEE Des Test Comput 22(5):21–31
27. Halfhill TR (2006) Ambric's new parallel processor. Microprocessor Report
28. Hansson A, Goossens K (2007) Trade-offs in the configuration of a network on chip for multiple use-cases. In: Proc NOCS
29. Hansson A, Goossens K (2009) An on-chip interconnect and protocol stack for multiple communication paradigms and programming models. In: Proc CODES+ISSS
30. Hansson A, Goossens K, Rădulescu A (2005) A unified approach to constrained mapping and routing on network-on-chip architectures. In: Proc CODES+ISSS
31. Hansson A, Coenen M, Goossens K (2007) Undisrupted quality-of-service during reconfiguration of multiple applications in networks on chip. In: Proc DATE
32. Hansson A, Goossens K, Rădulescu A (2007) Avoiding message-dependent deadlock in network-based systems on chip. VLSI Des 2007:1–10
33. Hansson A, Goossens K, Bekooij M, Huisken J (2009) Compsoc: a template for composable and predictable multi-processor System on Chips. ACM Trans Des Autom Electron Syst 14(1):1–24
34. Hansson A, Subburaman M, Goossens K (2009) Aelite: a flit-synchronous network on chip with composable and predictable services. In: Proc DATE
35. Hansson A, Wiggers M, Moonen A, Goossens K, Bekooij M (2009) Enabling application-level performance guarantees in network-based Systems on Chip by applying dataflow analysis. IET Computers and Design Techniques
36. Hoskote Y, Vangal S, Singh A, Borkar N, Borkar S (2007) A 5-GHz mesh interconnect for a teraflops processor. IEEE MICRO 27(5):51–61
37. Hu J, Mărculescu R (2003) Exploiting the routing flexibility for energy/performance aware mapping of regular NoC architectures. In: Proc DATE
38. Jalbert A et al (2004) ×pipesCompiler: a tool for instantiating application specific networks on chip. In: Proc DATE
39. Jantsch A (2006) Models of computation for networks on chip. In: Proc ACSD
40. Kavaldjiev N (2006) A run-time reconfigurable network-on-chip for streaming dsp applications. PhD thesis, University of Twente
41. Kollig P, Osborne C, Henriksson T (2009) Heterogeneous multi-core platform for consumer multimedia applications. In: Proc DATE
42. Krstić M, Grass E, Gürkaynak F, Vivet P (2007) Globally asynchronous, locally synchronous circuits: overview and outlook. IEEE Des Test Comput 24(5):430–441
43. Lee H, Chang N, Ogras U, Marculescu R (2007) On-chip communication architecture exploration: a quantitative evaluation of point-to-point, bus, and network-on-chip approaches. ACM Trans Des Autom Electron Syst 12(3):1–20
44. Leijten J, van Meerbergen J, Timmer A, Jess J (2000) Prophid: a platform-based design method. J Des Autom Embed Syst 6(1):5–37
45. Liang J, Swaminathan S, Tessier R (2000) aSOC: a scalable, single-chip communications architecture. In: Proc PACT
46. Liu G, Ramakrishnan KG (2001) A*Prune: an algorithm for finding K shortest paths subject to multiple constraints. In: Proc INFOCOM
47. Mangano D, Locatelli R, Scandurra A, Pistritto C, Coppola M, Fanucci L, Vitullo F, Zandri D (2006) Skew insensitive physical links for network on chip. In: Proc NANONET
48. Marculescu R, Ogras U, Peh LS, Jerger N, Hoskote Y (2009) Outstanding research problems in NoC design: system, microarchitecture, and circuit perspectives. IEEE Trans Comput-Aided Des Integr Circuits Syst 28(1):3–21
49. Marescaux T, Mignolet J, Bartic A, Moffat W, Verkest D, Vernalde S, Lauwereins R (2003) Networks on chip as hardware components of an OS for reconfigurable systems. In: Proc FPL
50. Millberg M, Nilsson E, Thid R, Jantsch A (2004) Guaranteed bandwidth using looped containers in temporally disjoint networks within the Nostrum network on chip. In: Proc DATE
51. Moerman K (2007) Embedded vector processor is one way to tune software-defined radios. In: EE Times
52. Moraes F, Calazans N, Mello A, Möller L, Ost L (2004) HERMES: an infrastructure for low area overhead packet-switching networks on chip. Integr VLSI J 38(1):69–93

53. Moreira O, Valente F, Bekooij M (2007) Scheduling multiple independent hard-real-time jobs on a heterogeneous multiprocessor. In: Proc EMSOFT
54. Murali S, De Micheli G (2005) An application-specific design methodology for STbus crossbar generation. In: Proc DATE
55. Nandi A, Marculescu R (2001) System-level power/performance analysis for embedded systems design. In: Proc DAC
56. Nollet V, Marescaux T, Avasare P, Mignolet JY (2005) Centralized run-time resource management in a network-on-chip containing reconfigurable hardware tiles. In: Proc DATE
57. OCP (2007) OCP Specification 2.2. OCP International Partnership
58. Ogras UY, Hu J, Marculescu R (2005) Key research problems in NoC design: a holistic perspective. In: Proc CODES+ISSS
59. Panades I, Greiner A, Sheibanyrad A (2006) A low cost network-on-chip with guaranteed service well suited to the GALS approach. In: Proc NANONET
60. Paukovits C, Kopetz H (2008) Concepts of switching in the time-triggered network-on-chip. In: Proc RTCSA
61. Penning de Vries R (2008) Ic innovations in automotive. In: Plenary session international conference on solid-state and integrated-circuit technology
62. PIBus (1994) PI-Bus Standard OMI 324. Siemens AG, ver. 0.3d edn
63. Pinto A et al (2003) Efficient synthesis of networks on chip. In: Proc int'l conference on computer design (ICCD)
64. Pullini A, Angiolini F, Murali S, Atienza D, De Micheli G, Benini L (2007) Bringing NoCs to 65 nm. IEEE MICRO 27(5):75–85
65. Rădulescu A, Dielissen J, Goossens K, Rijpkema E, Wielage P (2005) An efficient on-chip network interface offering guaranteed services, shared-memory abstraction, and flexible network programming. IEEE Trans CAD Integrated Circuits Syst 4–17
66. Rostislav D, Vishnyakov V, Friedman E, Ginosar R (2005) An asynchronous router for multiple service levels networks on chip. In: Proc ASYNC
67. Rutten M, Pol EJ, van Eijndhoven J, Walters K, Essink G (2005) Dynamic reconfiguration of streaming graphs on a heterogeneous multiprocessor architecture. In: IS&T/SPIE electron imag, vol 5683
68. Scherrer A, Fraboulet A, Risset T (2006) Automatic phase detection for stochastic on-chip traffic generation. In: Proc CODES+ISSS
69. SonicsMX (2005) SonicsMX Datasheet. Sonics, Inc. Available on www.sonicsinc.com
70. Stergiou S, Angiolini F, Carta S, Raffo L, Bertozzi D, de Micheli G (2005) ×pipes Lite: a synthesis oriented design library for networks on chips. In: Proc DATE
71. Stoica I, Zhang H (1999) Providing guaranteed services without per flow management. In: Proc SIGCOMM
72. Tota S, Casu M, Roch M, Macchiarulo L, Zamboni M (2009) A case study for NoC-based homogeneous MPSoC architectures. IEEE Trans Very Large Scale Integr 17(3):384–388
73. van Berkel K, Heinle F, Meuwissen P, Moerman K, Weiss M (2005) Vector processing as an enabler for software-defined radio in handheld devices. EURASIP J Appl Signal Process 2005:2613–2625
74. Weber WD, Chou J, Swarbrick I, Wingard D (2005) A quality-of-service mechanism for interconnection networks in system-on-chips. In: Proc DATE
75. Wentzlaff D, Griffin P, Hoffmann H, Bao L, Edwards B, Ramey C, Mattina M, Miao CC, Brown JF, Agarwal A (2007) On-chip interconnection architecture of the tile processor. IEEE MICRO 27(5):15–31
76. Wielage P, Marinissen E, Altheimer M, Wouters C (2007) Design and DfT of a high-speed area-efficient embedded asynchronous FIFO. In: Proc DATE
77. Wiggers MH, Bekooij MJ, Smit GJ (2008) Buffer capacity computation for throughput constrained streaming applications with data-dependent inter-task communication. In: Proc RTAS
78. Wiklund D, Liu D (2003) SoCBUS: switched network on chip for hard real time embedded systems. In: Proc IPDPS
79. Wingard D, Kurosawa A (1998) Integration architecture for system-on-a-chip design. In: Proc CICC
80. Wolkotte P, Smit G, Rauwerda G, Smit L (2005) An energy-efficient reconfigurable circuit-switched network-on-chip. In: Proc IPDPS