This appendix discusses the NoC analysis, optimization and evaluation tools used while preparing this book. We first present the NoC performance analysis, architecture optimization and voltage-frequency island synthesis tools. Then, we present the simulator used to evaluate these tools. Finally, we provide a brief overview of the NoC prototypes implemented to demonstrate and evaluate our major contributions.

A.1 NoC Analysis and Optimization Tools

A.1.1 NoC Performance Analysis Tool

The NoC performance analysis tool is coded using C++. The tool uses a simple command line interface to setup different parameters. These parameters are:

- Network architecture and routing algorithm: The architectures is accepted using the “architecture-config” format. This format specifies the number of routers, their interconnection and the routing algorithm. A configuration file can be generated in two ways. The first way is to generate a configuration file given a standard mesh network of any size and a standard deterministic routing algorithm, such as XY routing. We developed a separate utility for this purpose. The second option is to use our tool for the NoC performance optimization via long-range link insertion. This format is also supported by the worm_sim simulator [11] which is used to evaluate the performance of our analysis tool.
- Number of virtual channels
- Depth of the input buffers in the router
- Target application: the tool accepts the application in the “traffic-config” format. In this format, each line in the input file contains a 3-tuple, which
specifies the source IP, the destination IP and the packet rate from the source to the destination. This format is also supported by worm_sim simulator [11] and it can be generated from “APCG format (see Sect. 4.3) via worm-sim.

A.1.2 NoC Architecture Customization via Long-Range Links

The long-range link insertion tool is also developed using C++. This tool accepts the application description in the same way as the performance analysis tool described in Appendix A.1.1. In addition to this, the size of the initial mesh network and the default routing algorithms are taken as command line inputs. Finally, the tool takes the maximum total length of the long-range links that can be added to the initial mesh network. The output is an configuration file in the “architecture-config” format. As mentioned before, this format specifies the number of routers, their interconnection and the routing algorithm. The output file can be used for the NoC performance analysis tool and worm_sim simulator for further evaluation.

A.2 Simulator Support

A.2.1 Worm_sim NoC Simulator

We utilized a cycle-accurate NoC simulator, called worm_sim to evaluate the techniques presented in this book. Worm_sim was developed from scratch in C++ using a standard template library by Jingcao Hu [11]. The initial version of worm_sim is capable of simulating mesh and torus topologies under various routing algorithms. The user controllable performance parameters include channel buffer size, routing engine delay, crossbar arbitration delay, etc. Worm_sim can simulate the system under standard traffic patterns, such as uniform, transpose, and hotspot traffic patterns, as well as application-specific traffic, specified by application configuration files or traces. Besides reporting average packet latencies, worm_sim also reports communication energy consumption using the Ebit model [13] and the Orion power model library [10].

- Besides the standard mesh and torus topologies, the current version accepts mesh or torus topologies with long-range links. Furthermore, any deterministic routing algorithm is accepted in the form of a routing table. This enables us to simulate arbitrary topologies with any deterministic routing.1

---

1 Link insertion is done explicitly using long-range links, while link removal is mimicked by never using certain links.
The flow control technique presented in Chap. 7 is implemented in *worm_sim*. Besides the average packet latency reported by the initial version, the extended version more detailed performance reports such as average and maximum number of packet in the network at a given time, the average delay experience at each router and the input buffer utilizations.

- We have developed two new mechanisms to generate traffic in *worm_sim*. First, we implemented ON–OFF traffic sources presented in Sect. 7.4 in *worm_sim*. Besides this, we have developed a tool that generates finite state machines (FSM) to describe the behavior of each traffic generator. This way, the simulator can capture the control and data dependencies in the target application and generate more realistic traffic patterns than random traffic.

### A.3 On-Chip Router Prototype

On-chip routers are at the heart of NoC designs. Therefore, we designed an output buffered on-chip router which is shared across all prototypes developed in this dissertation. Due to its moderate buffer requirements, our design (whose simplified block diagram is shown in Fig. A.1) implements wormhole flow control. The router consists of four pipeline stages; hence, it takes four cycles to route the header flit. Then, the remaining flits of the packet, which can vary from 1 flit to 255 flits, follow the header in a pipelined fashion. The depth and width of the output buffers are parameterized. The packets in the network are divided into 16-bit flits, since the width of the channels is 16 bits. The router supports deterministic routing, the routing strategy being implemented as a lookup table for flexibility reasons. Finally, based on the network topology and their location in the network, the routers may have different number of ports and area, as summarized in Table A.1.

![A simplified block diagram of the on-chip router](image)
The router was implemented using Verilog HDL. Standard FIFOs from Xilinx IP library are used to implement the output buffers, while the remaining modules are custom designs. To test the functionality of the routers, a 4 × 4 mesh network is instantiated. Then, random number generators are attached to each router to generate random traffic. Finally, the resulting network is simulated using ModelSim to verify that all the packets reach successfully their destinations. More details of this prototype can be found in [6, 8].

A.4 NoC with Application-Specific Long-Range Links

To further demonstrate the effectiveness of the long-range link insertion methodology, we present an FPGA prototype using a Xilinx Virtex™-II XC2V4000 device. We first connected the on-chip routers to compose a 4 × 4 mesh network using Verilog HDL. After the operation of the network is tested by hardware simulation performed using ModelSim, application-specific long-range links for hotspot traffic pattern and various benchmarks from E3S suite [2] are determined. To implement the network with long-range links, we replaced the routers used in the original mesh network with 6-port routers whenever necessary, and inserted the long-range links. This process is accomplished in less than a day by modifying the top level Verilog module describing the mesh network. Similarly, the testbench used for the mesh network is reused to test the functionality of the network with long-range links. Finally, the designs are synthesized, implemented and downloaded to Xilinx Virtex-II FPGA using Xilinx ISE Foundation.

Inserting long-range links requires more resources, hence, increases the area of the design. Therefore, we analyze the size of the individual routers in a 4 × 4 mesh network and its customized version with long-range links under the constraint of $s(l) = 12$ (Table A.1). We observe that moving from 3-to-4, 4-to-5, and 5-to-6 ports increases the slice utilization by 85, 93 and 106 slices, respectively. Moreover, we observe that the total number of slices used by a 4 × 4 network implementing transpose traffic rises about 7.0 %, as summarized in Table A.1. More details of this prototype can be found in [8].
We observe that the improvements in the average message latency and network throughput measured using the FPGA prototype are consistent with the simulation results. The latency comparison under transpose traffic is plotted in Fig. A.2, while further experimental results can be found in [8]. This basically validates our simulation results and offers a solid basis for the newly proposed approach.

We also performed accurate energy measurements on the FPGA prototype using the cycle-accurate power measurement tool develop by Lee et al. [5]. The experiments showed minimal impact on the energy consumption and validated the theoretical expectations and simulation results in Sect. 6.6 [8].

### A.5 Implementation Overhead of Flow-Control Algorithm

In order to accurately evaluate the area overhead, we implemented the proposed flow control in Verilog HDL and synthesized the design using Synopsys Design Compiler. The equivalent gate count of the proposed flow controller is found as 1093 gates.

We also integrated the proposed flow controller into an existing router and developed an FPGA prototype based on a Xilinx XC2V3000 platform. The basic NoC router without the proposed flow controller is based on the prototype presented in Appendix A.3. The router implements the basic link-level ON/OFF flow control. The input buffers of the router have 16 flit depth and 16 bit width. The router implements wormhole flow control with deterministic table-based routing.

The router takes four cycles to process the header flit (that is, to receive, make a routing decision, traverse the crossbar switch and place it to the desired outgoing link). After that, the remaining flits simply follow the header in a pipelined fashion. The time it takes to route packets is not affected by the flow controller,
since the computation of the availabilities are performed concurrently with routing.

The proposed controller takes up 80 slices in the target FPGA; this corresponds to about 18% increase in the number of resources used by the router. It is also important to evaluate the overhead of the router in a real design. For instance, the overhead of the proposed controller is about 0.8% for the MPEG-2 encoder presented in [6]. In general, the overhead of our controller is estimated be about 1% of the total chip area.

A.6 Validation of VFI-Based NoC via Prototyping

This section presents a FPGA prototype based on Virtex2Pro Xilinx FPGA platform [9] for NoCs with multiple VFIs, and the dynamic frequency control architecture.

A.6.1 Design of NoCs with Multiple VFIs

A typical router in an NoC consists of a FIFO and an output controller (OC) for each port, and an arbiter to channel the traffic between the ports, as depicted in Fig. A.3. To connect a node in a VFI with another node residing in a different VFI, all data and control signals need to be converted from one frequency/voltage domain to another. For this purpose, we implemented mixed-clock/mixed-voltage interfaces using FIFOs, which are natural candidates for converting the signals from one VFI to the another, as shown in Fig. A.3.

Fig. A.3 Illustration of the interface between two different voltage-frequency domains VFI1 and VFI2
To support the simulation results, we implement a GALS-based NoC with a $4 \times 4$ mesh topology using Verilog HDL. Block RAM-based mixed-clock FIFOs from the Xilinx library are used in routers to transfer data between different clock domains. Our design can be partitioned into as many as 16 VFIs. In our implementation, the signal conversion, both in terms of clock and voltage domains, occurs at FIFO interfaces. In this particular design, the Delay Locked Loops (DLLs) from the Xilinx FPGA device are used to generate the individual clock signals. However, since multiple voltage levels are not readily available for FPGA platforms, our prototype does not support voltage level conversion.

For experimental purposes, this implementation is configured with 16 islands and simulated using the auto-industry benchmark from E3S [2]. We first verify that no packets are lost in the VFI interfaces. After that, we compute the total energy consumption corresponding to single VFI and 2-VFI implementations, as shown in Sect. 8.5.1. To compute the energy consumption values, we utilize the energy characterization of the on-chip routers reported in [6]. The total energy consumption for single VFI operating at 1V is found as 109 nJ. On the other hand, the total energy consumption of the 2-VFI partitioning found using the proposed approach is 21.2 nJ. Hence, we observe about 81% reduction in the energy consumption. The energy consumption results obtained using the FPGA prototype are different from that measured by simulation in Table 8.1 due to the differences in the target platform and implementation details. Nevertheless, we note that according to the simulation results, the relative improvement in the energy consumption for the same benchmark is 80%, which is very close to the result obtained using the actual prototype.

### A.6.2 Dynamic Frequency Control Architecture

The NoC prototype presented in the previous section consists of multiple VFIs operating at different clock frequencies. However, it does not support dynamic frequency scaling. The architecture we present in this section illustrates the dynamic frequency control technique presented in Sect. 8.4.

The dynamic frequency control architecture with three different frequency islands is depicted in Fig. A.4. Delay Locked Loops (DLLs) available on the Xilinx FPGA are used to generate four basic clock signals that are not multiples of each other (see second column in Table A.2). These four clock signals are further divided by the clock control module to generate the clock signals for the islands by the clock control module according to the utilization of the interface queues under control (FIFO 1 and FIFO 2, in Fig. A.4).

The clock control module is activated periodically, once for every control interval $T$. The dynamic frequency scaling algorithm (e.g., the state-space feedback controller described in Sect. 8.4) is implemented in the clock control module. Our current implementation is based on PicoBlaze microprocessor [12] for flexibility reasons. It could be also implemented using dedicated hardware [1].
We also note that only a finite set of different clock frequencies can be derived from the basic clocks. For example, the current implementation can derive 22 different clock signals using the four basic clock signals, as summarized in Table A.2. Clocks of individual VFIs are selected from these 22 clock frequencies by the clock control module.

![Clock Control Diagram]

**Fig. A.4** Dynamic frequency control architecture. Clock DLLs generate four basic clocks (20, 17.5, 15, 12.5 MHz). The clock control module whose implementation is depicted on the right-hand side of the figure is capable of deriving 22 distinct clocks from these basic clocks, as shown in Table A.2. Clocks of individual VFIs are selected from these 22 clock frequencies by the clock control module.

**Table A.2** The twenty-two output clocks that can be generated by our current FPGA implementation

<table>
<thead>
<tr>
<th>Output clocks (MHz)</th>
<th>20</th>
<th>20</th>
<th>10</th>
<th>5</th>
<th>2.5</th>
<th>1.25</th>
<th>0.625</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic input clocks (MHz)</td>
<td>20</td>
<td>17.5</td>
<td>8.75</td>
<td>4.375</td>
<td>2.188</td>
<td>1.094</td>
<td>0.547</td>
</tr>
<tr>
<td>15</td>
<td>15</td>
<td>7.5</td>
<td>3.75</td>
<td>1.875</td>
<td>0.938</td>
<td>0.478</td>
<td></td>
</tr>
<tr>
<td>12.5</td>
<td>12.5</td>
<td>6.25</td>
<td>3.125</td>
<td>1.563</td>
<td>0.781</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

We also note that only a finite set of different clock frequencies can be derived from the basic clocks. For example, the current implementation can derive 22 different clock signals using the four basic clock signals, as summarized in Table A.2. Therefore, the clock control module selects one of these clock frequencies with the help of search frequency and frequency table modules, as depicted in Fig. A.4.

The dynamic frequency controller depicted in Fig. A.4 utilizes 474 4-input look-up tables (LUTs) in Xilinx Virtex-II Pro XC2VP30, which implies a small area overhead. For example, we divided the NoC-based MPEG-2 encoder presented in [4] into three VFIs and used the proposed control architecture to control the individual clock frequencies. This increase the device utilization from 16,966 LUTs to 19,161 LUTs resulting in about 13% overhead. Even when there are no workload variations, our measurements using the Xpower tool from Xilinx show that using the 3VFI architecture decreases the power consumption from 277 to 259 mW. Therefore, the proposed architecture is expected to provide significant savings for multimedia traffic which is typically characterized by large workload variations. Finally, the current implementation achieves a maximum frequency of
122 MHz in the target FPGA. As a result, it can be employed as a test bed for evaluating the effectiveness of DFS (Dynamic Frequency Scaling) algorithms on FPGA prototypes; this can further help projecting the energy savings when voltage scaling is also performed on the actual implementation.
Appendix B
Experiments Using the Single-Chip Cloud Computer (SCC) Platform
With Contributions from Paul Bogdan and Radu David, Carnegie Mellon University

Single-chip Cloud Computer (SCC) platform which is a research multi-core platform built by Intel [4]. The chip contains 24 tiles, each with two processing cores, interconnected by a $4 \times 6$ mesh NoC. There are six voltage and frequency islands (VFIs), organized in groups of four tiles. For this platform, voltage can be adjusted per island, while frequency can be adjusted on each tile individually.

The 48 cores of the SCC platform run a special Linux distribution that uses the RCCE library for inter-core communication. Each core runs a separate instance, so this can be regarded as a distributed platform that has separate kernels on each processor; the interaction between cores is done exclusively through message passing. Each core is capable of running its own application stored in the system shared DDR3 memory, while synchronization and communication among the cores is performed using the RCCE API [7]. It allows programmers to create barriers for synchronization and communication purposes, to pass messages among the cores or access the core timer.

To ensure a fast communication among cores, a dedicated memory buffer is used. Each tile has its own such message passing buffer (MPB), consisting of 16 KB of SRAM, for a total of 384 KB. This address space is memory mapped on each core and the RCCE functions use it as a message passing interface. Power management capability is also provided by the RCCE API through functions that can be used to modify the voltage and frequency of the tiles.

B.1 Driver Application

We employ the Sobel algorithm, a popular image processing algorithm, which is at the basis of most video processing approaches [3]. For our implementation, the application can be organized such that a ‘parent’ core loads a full $1024 \times 1024$ image from memory, proceeds to analyze it and then sends 16 blocks of data that contain $256 \times 256$ pixels, each from the image to the sixteen children cores. This partitioning offers a convenient workload for 16 cores as shown in Fig. B.1.
response, the children cores perform edge detection and send the resulting image to the parent core. Finally, the parent core reassembles the processed image and writes it back to memory. A description of the application partitioning and the mapping of the application on the SCC platform is shown in Fig. B.1.

At the finest level of granularity, the Sobel algorithm involves computing a $3 \times 3$ pixel convolution to get a value for the edge intensity of a pixel. This mask is applied to the entire image to obtain the edge map, as shown in Fig. B.2.

Significant variation in the communication volume is obtained by identifying regions of the image where there is not much pixel variation. These areas show no sign of edges and thus edge detection doesn’t have to be performed. The parent core performs this check and only sends data for the regions that contain significant features variation.

Fig. B.1 Mapping of Sobel image processing algorithm on the SCC. Each SCC tile contains two cores, meaning that the image processing application runs on nine different tiles.

Fig. B.2 Partitioning and results shown for running the Sobel edge detection algorithm. The idea of workload variation based on identifying featureless and crowded areas of an image is also shown.
B.2 Implementation of the Dynamic Power Manager on SCC

To properly monitor the buffer occupancy at run-time an occupancy variable keeps track of the filling level of the each core MPB, and also measures the time between two consecutive send and two consecutive receive operations. At the end of the communication round, the MPB occupancy values are transmitted to the control core.

Each voltage island contains four tiles or eight processors. The API allows for only one of the eight cores, called a Power Domain Master, to adjust the voltages of the island it controls. Our application is running on 17+1 cores that span three voltage islands, hence three different MPBs are monitored while the application is running. The queue usage of each MPB is sent to the control core in an asynchronous fashion, meaning that a polling strategy is also implemented. The control core checks for new MPB occupancies continuously and also updates the global timer for the next instance of the control interval. At every control interval, the controller computes the next frequency level based on the given occupancy reference, the feedback control matrix and the current MPB occupancies. The resulting frequencies are then sent back to the three domain master cores which finally select the closest frequency divider to the resulting frequency.

B.3 Experiments with Intel Single-Chip Cloud Computer Prototype

We use the MBPs in the tiles of SCC to track the activity level, i.e., as the system state defined in Eq. 8.12. In other words, the system reacts to the variation in MPB occupancy and modify the frequency and voltage accordingly. To test this, we analyzed the output of the application for a particular video frame. The results in Fig. B.3 clearly show the power dissipation changes which are well correlated with the variations in buffer occupancy levels. This particular frame shows how

![Fig. B.3 Example of the control algorithm behavior (i.e., power and MPB occupancy) while-running an individual frame. Area a has less features than area b; thus the MPB occupancy varies differently and the power controller reacts by reducing the frequency and voltage of the islands, significantly reducing the overall power consumption of the system](image-url)
the top region, with fewer features, requires less data to be communicated and thus smaller values of occupancy are observed, while the bottom region, which requires all of the pixels to be sent to children cores, generates a higher occupancy values which determines an increase in the level of power used.

Figure B.4 shows the results from a run of our program on 10 separate frames with varying complexity. It can be clearly seen how images with fewer features have a lower power envelope, while more complex images use more power. To perform an in-depth power and performance test, the program was run for 100 consecutive frames from a movie and power measurements were recorded. Dynamic power management enabled 38% power reduction compared to the static voltage and frequency implementation.

It is important to note that due to limitations in the implementation of the control algorithm, this particular prototype uses only 8 of the 48 cores, while the rest of the cores are kept idle and at the lowest frequency and voltage levels. This means that for an application that would take advantage of more of the 48 cores, even better results can be obtained since the workload variation will be even higher.

References

Index

A
Algorithms, 11, 13–16, 40, 76, 105–107, 116, 144, 158
Analytical models, 9, 15
Application mapping, 7, 10, 11, 41, 52, 65, 68, 69, 92
Application scheduling, 11
Architecture customization, 158
Average packet latency, 21, 22, 50–54, 64–69, 76, 82, 92–99, 113, 124–132, 159, 161

B
Bandwidth, 3, 12, 18, 19, 21–23, 42–45, 52, 54, 63, 90, 105, 107
Benchmark, 84, 93, 97–101, 141, 147, 149, 150, 163

C
Clocks, 21, 137, 164
Clustering coefficient, 84, 85
Communication infrastructure, 3, 6, 7, 17, 39, 41
Communication architecture, 1, 3, 4, 5, 18, 33, 34, 61, 111, 155
Communication paradigm, 6, 7, 12, 41, 136
Congestion control, 14, 15, 105, 108
Contention probability, 18, 52

D
Design automation, 141
Design methodology, 1, 6, 46, 101, 136
Dynamic power management, 147, 152, 170

E
Energy and power consumption, 1, 2, 5, 10–12, 15–19, 21–23, 34, 36, 37, 135, 136, 138–144, 147–149, 164, 169

F
Fault tolerance, 5, 12, 16, 17, 21, 40, 80, 92, 107
Feedback control, 136, 137, 144–147, 150, 151, 169
Finite buffers, 52
FPGA, 23, 34, 37, 46, 91, 92, 101, 124, 131, 137, 151, 160–165
FPGA prototyping, 22, 46, 162
Free packet delay, 81, 85

G
Globally asynchronous locally synchronous (GALS), 1, 15, 21, 135, 137, 163

I
Input/output and data communications, 39
Index

L
Latency, 3, 5, 11, 12, 14, 15, 17–22, 40–43, 50–54, 63, 64, 66–69, 72, 73, 76, 80, 81, 88, 92, 94–99, 101, 107, 112, 113, 124–131, 137, 140, 159, 161
Low power, 15, 18, 19, 21, 155

M
Mathematical model, 9
Microarchitecture, 13, 16, 60
Mixed FIFO, 136, 140, 141, 150, 162, 163
MPEG-2, 23, 33–35, 37, 109, 110, 162–164
Multicore design, 155
Multimedia applications, 7, 33
Multiprocessor systems-on-chip (MPSoCs), 11, 15, 151

N
Network-on-chip, 1, 2, 34, 39
Network synthesis, 3–5, 20, 41, 42, 157
Network topology, 5, 17, 35, 42, 43, 71, 81, 108, 109, 159
NoC design, 3–6, 11, 14, 15, 17, 19, 22, 36, 37, 42, 43, 46, 69, 73, 135, 152, 155, 159
NoC validation, 4, 5, 21

O
On-chip communication, 1, 2, 18, 22, 34
ON/OFF traffic model, 110
Optimization, 3, 4, 9, 12, 16, 19, 49–52, 64, 65, 68, 69, 73, 76, 81, 82, 84, 99, 135, 143, 144, 153, 155, 157

P
Packet injection rate, 65, 67, 78, 81, 105, 106, 127, 132
Performance analysis, 5, 19, 20, 22, 46, 49–53, 57, 63, 66, 71, 72, 157, 158
Performance evaluation, 5, 22, 23, 35
Performance optimization, 157
Phase transition, 77, 82, 93

Q
Quality of service (QoS), 6, 12, 14, 15, 23, 42, 92, 137

R
Runtime energy management, 11, 15, 16

S
Scalability, 2, 3, 9, 17, 20, 23, 33, 36, 37, 94, 130
Service time, 2, 50–52, 54, 56, 59–68
Simulation, 5, 21, 22, 49, 50, 67–72, 81, 82, 93, 96, 100, 124, 125, 132, 150, 160, 161, 163
Small-world networks, 6, 76, 84
Systems-on-chip (SoCs), 1, 2, 16, 20

T
Thermal management, 15, 16, 152
Topology synthesis, 18, 20, 42
Traffic modeling, 4, 9, 10, 109, 110

V
Virtual channels, 57, 58
Voltage-frequency islands, 21, 135, 138, 148–152, 162, 163