Università degli Studi di Bologna

FACULTY OF ENGINEERING

Doctoral Course in Electronics, Computer Science and Telecommunications

INTERCONNECTION SYSTEMS FOR HIGHLY INTEGRATED COMPUTATION DEVICES

PhD Dissertation by:

FEDERICO ANGIOLINI

Advisor:

Prof. LUCA BENINI

Co-advisor:

Prof. BRUNO RICCÒ

Doctoral Course Coordinator:

Prof. PAOLO BASSI

ING-INF/01

Academic Year 2006/2007 – XX Cycle

Scaling Everest was, by far,
the most amazing and transformative experience of my life.
Unfortunately, this is a thesis on context-free grammars.

-Jonathan Blum

Abstract

The sustained demand for faster, more powerful chips has been met by the availability of chip manufacturing processes allowing for the integration of increasing numbers of computation units onto a single die. The resulting outcome, especially in the embedded domain, has often been called SYSTEM-ON-CHIP (SOC) or MULTI-PROCESSOR SYSTEM-ON-CHIP (MPSOC).

MPSoC design brings to the foreground a large number of challenges, one of the most prominent of which is the design of the chip interconnection. With a number of on-chip blocks presently ranging in the tens, and quickly approaching the hundreds, the novel issue of how to best provide on-chip communication resources is clearly felt.

NETWORKS-ON-CHIPS (NOCS) are the most comprehensive and scalable answer to this design concern. By bringing large-scale networking concepts to the on-chip domain, they guarantee a structured answer to present and future communication requirements. The point-to-point connection and packet switching paradigms they involve are also of great help in minimizing wiring overhead and physical routing issues.

However, as with any technology of recent inception, NoC design is still an evolving discipline. Several main areas of interest require deep investigation for NoCs to become viable solutions:

- The design of the NoC architecture needs to strike the best trade-off among performance, features and the tight area and power constraints of the on-chip domain.
- Simulation and verification infrastructure must be put in place to explore, validate and optimize the NoC performance.
- NoCs offer a huge design space, thanks to their extreme customizability in terms of topology and architectural parameters. Design tools are needed to prune this space and pick the best solutions.

- Even more so given their global, distributed nature, it is essential to evaluate the physical implementation of NoCs to evaluate their suitability for next-generation designs and their area and power costs.

This dissertation focuses on all of the above points, by describing a NoC architectural implementation called ×pipes; a NoC simulation environment within a cycle-accurate MPSoC emulator called MPARM; a NoC design flow consisting of a front-end tool for optimal NoC instantiation, called SunFloor, and a set of back-end facilities for the study of NoC physical implementations.

This dissertation proves the viability of NoCs for current and upcoming designs, by outlining their advantages (along with a few tradeoffs) and by providing a full NoC implementation framework. It also presents some examples of additional extensions of NoCs, allowing *e.g.* for increased fault tolerance, and outlines where NoCs may find further application scenarios, such as in stacked chips.

Keywords

Network-on-Chip (NoC)
System-on-Chip (SoC)
Multi-Processor System-on-Chip (MPSoC)
On-Chip Interconnect Fabric
Design Flow
Physical Design
Fault Tolerance
Cycle-Accurate Simulation
3D Chip
Flow Control

Acknowledgments

First of all, I would like to thank my advisors over these years - Prof. Luca Benini, Prof. Giovanni De Micheli and Prof. Bruno Riccò - for their guidance and support. They have allowed me to get in contact with exciting research opportunities, with many brilliant people, and with world-class institutions. ~~Behind their backs~~Thanks to them, I have also had the chance to travel to many places in the world, from North America to Australia. (In a world of constrained budgets, we are still trying to figure out how I managed exactly).

I would also like to thank my family, for coping with my absences and, even worse, with my presences. I guess - in fact, I am pretty sure - it may have been a bumpy ride for them. Maybe it got a bit better when they began to realize that the chances of me coming back home in time for dinner were just... well, low.

Of course this dissertation would never have been written without the contribution of many people. To let the reader know who to blame, I just want to name a few. A special mention to my funny officemate Srini, for continuously promoting the "Rule of Ten": suggesting ways to get within 10% of my same experimental results, in about 10× less time. And then follow David, Paolo, Salvatore, Davide, Antonio, Shankar, Igor, Tommaso, who have all been great to work with. Even despite some painful defeats at Risk in the after hours.

A special thanks to all the astounding people in Bologna, Cagliari, Torino, Lausanne, Copenhagen and Madrid that I have had the chance to meet. Probably, in not so many other jobs I would have had the same chances of making so many good friends on the workplace.

And now, on to new exciting discoveries... :-)

Contents

Abstract i

Keywords iii

Acknowledgments v

Contents vii

List of Figures xiii

List of Tables xvii

List of Algorithms xix

Listings xxi

1 Introduction 1
- 1.1 Background . . . 3
- 1.2 Networks-on-Chips: Opportunities and Challenges . . . 6
- 1.3 Related Work . . . 9
  - 1.3.1 MPSoC Interconnects . . . 9
  - 1.3.2 Simulation and Traffic Generation . . . 15
  - 1.3.3 Topology Design . . . 20
  - 1.3.4 Fault Tolerance in NoCs . . . 22
  - 1.3.5 NoCs for 3D Chips . . . 24
- 1.4 Contributions of This Dissertation . . . 26
- 1.5 Dissertation Outline . . . 28

2 MPSoC Interconnect Evolution 31
- 2.1 Motivation and Key Challenges . . . 31
- 2.2 Bus Architectures . . . 32

2.2.1 AMBA 2.0 AHB . . . 32
2.2.2 STBus . . . 34
2.2.3 AMBA 3.0 AXI . . . 35
2.3 Bus Performance Analysis . . . 36
2.3.1 Impact of Protocols on Interconnect Scalability . . . 37
2.3.2 Topology Effect on Interconnect Performance . . . 42
2.4 Conclusions . . . 45

3 Simulation and Traffic Generation 47
3.1 Motivation and Key Challenges . . . 47
3.2 SystemC Platform Simulation . . . 49
3.3 Integration of Advanced Platform Cores . . . 51
3.3.1 The LISATek Design Platform . . . 53
3.3.2 The LISATek Simulation Interface . . . 53
3.3.3 L1 Memory Placement Strategies . . . 56
3.3.4 Core-Associated Devices . . . 58
3.3.5 The Resulting System Architecture . . . 59
3.3.6 Experiments and Case Studies . . . 59
3.4 Reactive Traffic Generation . . . 63
3.4.1 Application Reactiveness in MPSoC Environments . . . 66
3.4.2 RIPE Model and Implementation . . . 71
3.4.3 Using RIPE Programs . . . 76
3.4.4 RIPE as a Simulation Aid . . . 78
3.4.5 Validation Results . . . 83
3.4.6 RIPE as a Design Tool . . . 88
3.5 Conclusions . . . 91

4 The ×pipes NoC Architecture 95
4.1 Motivation and Key Challenges . . . 95
4.2 General Concepts . . . 96
4.3 The ×pipes Building Blocks . . . 98
4.3.1 Network Interfaces . . . 98
4.3.2 Switches . . . 100
4.3.3 Links . . . 101
4.4 NoC Flow Control Protocols . . . 101
4.4.1 Retransmission-Based Flow Control Protocol . . . 105
4.4.2 Credit-Based Flow Control Protocol . . . 106
4.4.3 Timing-Error-Tolerant Flow Control Protocol . . . 106
4.4.4 Experiments on Alternative Flow Controls . . . 108
4.5 Conclusions . . . 113

5 NoC Design Flow Front-End: Topology Design 115
5.1 Motivation and Key Challenges . . . 115
5.1.1 Custom Topology Design and Floorplan Awareness . 115
5.1.2 Deadlock Freedom . . . 117
5.1.3 Target Operating Frequency . . . 120
5.1.4 Proposed Solution . . . 120
5.2 Required Input Models . . . 121
5.3 Topology Design Algorithm . . . 123
5.4 Topology Instantiation . . . 131
5.5 Experiments and Case Studies . . . 133
5.5.1 Experiments on MPSoC Benchmarks . . . 133
5.5.2 Case Study: A Layout-Level Comparisons . . . 136
5.5.3 Message-Dependent Deadlock Removal . . . 138
5.5.4 Compensation for Congestion Effects . . . 141
5.6 Conclusions . . . 142

6 NoC Design Flow Back-End: Physical Implementation 145
6.1 Motivation and Key Challenges . . . 145
6.2 A Synthesis Flow . . . 147
6.2.1 A Traditional View of the Back-End Design Flow . . 147
6.2.2 The ×pipes Back-End Infrastructure . . . 148
6.2.3 Placement-Aware Logic Synthesis . . . 151
6.3 Cross-Benchmarking: NoCs Against Buses . . . 152
6.3.1 The Fabrics Under Test . . . 153
6.3.2 The Test Applications . . . 155
6.3.3 Fabric Synthesis . . . 158
6.3.4 Cross-Benchmarking Results . . . 160
6.4 NoC Area and Power Modeling . . . 172
6.4.1 Proposed Modeling Methodology . . . 172
6.4.2 Parameters of Interest . . . 173
6.4.3 Area and Power Models . . . 176
6.4.4 Choice of a Relevant Training Set . . . 181
6.4.5 Fitting Model Coefficients . . . 182
6.4.6 Experimental Results . . . 184
6.5 Bringing NoCs to 65nm . . . 192
6.5.1 65nm Technology Libraries: Degrees of Freedom . . . 192
6.5.2 Link Delay and Power . . . 192
6.5.3 Wire Routability Issues in NoCs . . . 196
6.5.4 Design Space in 65nm Technology . . . 197
6.5.5 Tradeoffs in the Design of Large Switches . . . 198
6.5.6 Clock Tree Insertion . . . 200

6.5.7 Leakage Power . . . 201
6.5.8 Test Design: a Multimedia Benchmark . . . 201
6.6 Conclusions . . . 203

7 NoC Traffic Handling: Fault Tolerance, Performance, Power 207
7.1 Motivation and Key Challenges . . . 207
7.1.1 Case Study: MPEG4 Video Texture Coder . . . 209
7.1.2 Assumptions on Underlying System . . . 211
7.1.3 Proposed Hardware Extensions . . . 212
7.2 Run-Time Fault Tolerant NoC-Based Schemes . . . 214
7.2.1 Permanent Error Recovery Support . . . 214
7.2.2 Intermittent Error Recovery Support . . . 217
7.3 Experimental Results . . . 217
7.3.1 Performance Studies . . . 219
7.3.2 Architectural Exploration of NoC Features . . . 222
7.3.3 Effects of Varying Percentages of Critical Data . . . 222
7.3.4 Synthesis Results . . . 224
7.4 Additional Applications of the Proposed Methodology . . . 224
7.5 Conclusions . . . 225

8 Looking Forward: NoCs for 3D Chips 227
8.1 Motivation and Key Challenges . . . 227
8.2 Physical Modeling of Vertical TSVs . . . 229
8.3 Integration of TSVs within NoC Switches . . . 235
8.4 Implementation of TSV-Based NoCs . . . 237
8.5 Architecture of a Mesochronous Synchronizer for 3D NoCs . 240
8.5.1 Circuit Description . . . 241
8.5.2 Timing Margins of the Proposed Circuit . . . 243
8.5.3 Adding Support for Backwards Flow Control . . . 245
8.6 Experimental Results on Synchronizer Implementation . . . 249
8.6.1 Example Mesochronous Link Implementation . . . 249
8.6.2 Timing Properties of the Synchronizer . . . 251
8.6.3 Silicon Cost of Proposed Synchronizer . . . 252
8.7 Conclusions . . . 252

9 Conclusions and Future Work 255
9.1 Summary of Contributions . . . 255
9.2 Conclusions . . . 256
9.3 Future Work . . . 256

A Author's Relevant Publications 259

Bibliography 263

List of Figures

1.1 Infineon design. 2
1.2 Shared bus limitations. 4
1.3 Evolution of shared buses. 5
1.4 Conceptual view of a Network-on-Chip. 6
1.5 The complete proposed NoC design flow. 27

2.1 Partial STBus crossbars. 34
2.2 Burst interleaving waveforms. 38
2.3 Benchmark execution times, varying cache sizes. 39
2.4 Bus performance metrics, 256B caches. 41
2.5 Transaction completion latency with 256B caches. 42
2.6 Performance metrics for five different buses. 43
2.7 Performance with varying caches and wait states. 45

3.1 The MPARM SystemC virtual platform. 49
3.2 LISATek-based ASIP design flow. 54
3.3 Possible placements of the L1 memory. 57
3.4 The scheme of a *processor tile*. 58
3.5 A system simulation screenshot. 60
3.6 Performance *vs.* interconnect congestion. 61
3.7 Performance *vs.* cache size. 61
3.8 A 3-Slot VLIW FFT Processor. 62
3.9 Bus latency of a mixed ARM + FFT platform. 63
3.10 Polling behavior in the DES benchmark. 64
3.11 Possible usage scenarios of RIPE. 65
3.12 Timelines for relevant MPSoC application scenarios. 67
3.13 Application flow of **pipe**. 71
3.14 Trace to RIPE Program Flow. 79
3.15 Example of conversion from MPARM trace to RIPE program. 80
3.16 RIPE Accuracy Validation Test. 84
3.17 RIPE vs. native ARM speedup. 85

3.18 Traffic profiles according to traffic injections. 89
3.19 Performance of the four synchronization patterns under test. 92

4.1 ×pipes initiator and target NIs. 99
4.2 ×pipes initiator NI block diagram. 99
4.3 ×pipes switch block diagram. 101
4.4 ×pipes pipelined link block diagram. 102
4.5 T-Error protocol implementation. 107
4.6 T-Error concept waveforms. 107
4.7 The star-like topology under test. 109
4.8 Link congestion under increasing traffic. 110
4.9 Communication latency, congested pipelined links. 111
4.10 Benchmark runtime under increasing congestion, long links. 112
4.11 T-Error performance under varying error probabilities. . . . 113

5.1 The proposed NoC design flow: front-end. 116
5.2 Deadlocks and deadlock-free architectures. 119
5.3 Filter application. 122
5.4 Algorithm examples. 126
5.5 A simple NoC topology. 131
5.6 VOPD Application. 135
5.7 Image processing: hop delay comparison. 135
5.8 Hand-designed and automatically-generated topologies. . . 137
5.9 IMP Application. 139
5.10 Cost of message-dependent deadlock avoidance schemes. . 140
5.11 Number of message types and power consumption. 141
5.12 Compensation of congestion effects. 142

6.1 The proposed NoC design flow: back-end. 146
6.2 A schematic view of a traditional design flow. 148
6.3 The synthesis flow for ×pipes. 149
6.4 Example usage of fences. 150
6.5 Platform fabrics under test. 154
6.6 Communication graphs for the two applications under test. 156
6.7 The physical implementation flow for the fabrics under test. 159
6.8 Execution time of three benchmarks for varying cache sizes. 161
6.9 Latency of different transfers on the interconnects. 163
6.10 Layouts of the test fabrics. 166
6.11 Split report for area and power of three ×pipes topologies. . 170
6.12 Outline of characterization activity. 173
6.13 Area requirements *vs.* target operating frequency. 175

6.14 Dependency of the output buffer area on fw, bd. 177
6.15 Example traffic in a 4x2 switch. 179
6.16 Modeling inaccuracy. 187
6.17 Distribution of the area modeling inaccuracy. 188
6.18 Power modeling inaccuracy for a mesh. 188
6.19 Modeling inaccuracy, layout-level test set. 189
6.20 Modeling inaccuracy, layout-level training and test sets. . . . 190
6.21 Two ×pipes switches in different technology libraries. 193
6.22 Link power consumption, varying lengths and frequencies. . 195
6.23 A 65nm 4x4 ×pipes mesh. 198
6.24 Physical-level metrics for switches of increasing cardinality. 199
6.25 30-core multimedia benchmark with link pipelining. 202

7.1 2D wavelet decomposition of an image. 210
7.2 Target NI extensions to support traffic rerouting. 212
7.3 Handling of packet flows in the system. 215
7.4 The three topologies under test. 218
7.5 Performance cost of adding reliability support. 221
7.6 Dependency on size of the critical data set. 223

8.1 Through-Silicon Vias in SOI and bulk-silicon. 230
8.2 Schematic representation of a bundle of 3D vias. 231
8.3 Capacitance when sweeping via diameter, constant pitch. . . 233
8.4 Capacitance when sweeping via pitch, constant diameter. . . 234
8.5 Layout detail: switch and LEF macros of TSVs. 236
8.6 Frequency with ACK/NACK and STALL/GO in 2D and 3D. 237
8.7 2D 3x2 mesh NoC and possible 3D re-implementation. . . . 238
8.8 Layouts of original 2D mesh and 3D re-implementation. . . 239
8.9 Proposed mesochronous synchronizer circuit. 242
8.10 Proposed synchronization across two layers. 243
8.11 Circuit to generate the `latch_enable` control wire. 245
8.12 Example of the waveforms in the proposed synchronizer. . . 246
8.13 ACK/NACK modified switch block diagram. 248
8.14 STALL/GO modified switch block diagram. 249
8.15 Layout of a 3D chip stack with a mesochronous NoC link. . 250
8.16 Area cost to implement mesochronous synchronization. . . . 253

9.1 Conceptual view of a Network-on-Chip. 257
9.2 The complete proposed NoC design flow. 257

List of Tables

3.1 RIPE instruction set. 73
3.2 RIPE special registers. 74
3.3 RIPE *vs.* native ARM performance on AMBA. 86
3.4 Patterns of interrupt issue. 88

4.1 Flow control protocols at a glance. 104

5.1 Topology Comparisons. 117
5.2 Area and power models in 130nm technology. 122
5.3 SunFloor-generated topologies and standard topologies. . . 134

6.1 Pre- and post-placement achievable frequencies. 166
6.2 Power consumption of the fabrics. 168
6.3 Energy consumption of the fabrics. 168
6.4 Dependencies of P_A. 179
6.5 Dependencies of P_B, P_C. 180
6.6 Dependencies of P_D. 181
6.7 Summary of coefficient dependencies. 181
6.8 Accuracy of the linear *vs.* parabolic models. 191
6.9 Performance/power of meshes in different 65nm libraries. . 197
6.10 Dynamic and leakage power in a 22x22 switch. 201

8.1 Capacitance matrix of TSVs in SOI technology. 232
8.2 Capacitance matrix of TSVs in bulk-silicon technology. . . . 232

List of Algorithms

1 Topology Design Algorithm. 125
2 PATH_COMPUTE(i, SCG, ρ, PTS, θ). 127

Listings

3.1 LISATek communication API prototypes. 55
3.2 RIPE program for the **IO** application. 75
5.1 Example topology description file. 131

CHAPTER 1

Introduction

Silicon vendors are constantly facing pressure to deliver feature-rich, high-performance, low-power, low-cost chips, in as short a time as possible. Luckily, silicon manufacturing techniques have been continuously perfected, following the well-known Moore's Law; this has provided the potential for answering customer demands.

However, along the years, an increasing gap has been observed among the number of available on-chip transistors and the capability of designers to make good use of them. As a consequence, some trends in chip design have become crystal clear:

- An increasing emphasis on modularity, reuse and parallelism is mandatory. Redesign from scratch is too time-consuming. Also, deploying multiple instances of existing computation blocks can be more efficient than developing more powerful blocks. Therefore, libraries of so-called INTELLECTUAL PROPERTY (IP) cores are increasingly becoming the foundation of platform development.

- Also based on the previous item, complexity is nowadays shifting from the development of functional units to the task of system integration. This is exacerbated by the fact that full designs are nowadays almost impossible to characterize in all possible operating conditions, leading to closure, optimization and verification issues.

- Software tools devoted to design automation are key at all levels. This applies to performance characterization, platform assembly and validation, physical implementation, *etc.*. Without efficient tools, the sheer complexity of billion-transistor designs and deep-submicron lithographic processes is impossible to tackle by designer teams of any size.

A typical outcome of these trends are today's MULTI-PROCESSOR SYSTEMS-ON-CHIPS (MPSOCS). These are full-featured chips, composed of a variety of functional blocks, to the point of integrating the foundation of a whole system or device into a single die. MPSoCs are used in a variety of environments, including multimedia gadgets, gaming stations, smartphones, automotive equipment, healthcare devices, industrial machinery, aerospace control units, and many more. MPSoCs are built upon assemblies of IP cores, and rely extensively on COMPUTER AIDED DESIGN (CAD) tooling for initial design space exploration, system optimization, system verification, and physical implementation.

An increasingly critical piece of the MPSoC puzzle is the on-chip interconnection infrastructure. Today, even MPSoCs used in mid-range mobile phones can easily contain tens of IP cores (Figure 1.1 [1]), and new chips with more than a hundred such internal units are appearing for various applications. The trend expressing the number of IP cores that can be integrated on a chip is exponential, roughly doubling every 18 months. How to effectively provide communication resources among such a number of building blocks is clearly a challenge. In fact, it is likely a key factor in determining the success or failure of upcoming MPSoCs will be the ability to efficiently provide the communication backbone into which to seamlessly plug a variety of IP cores.

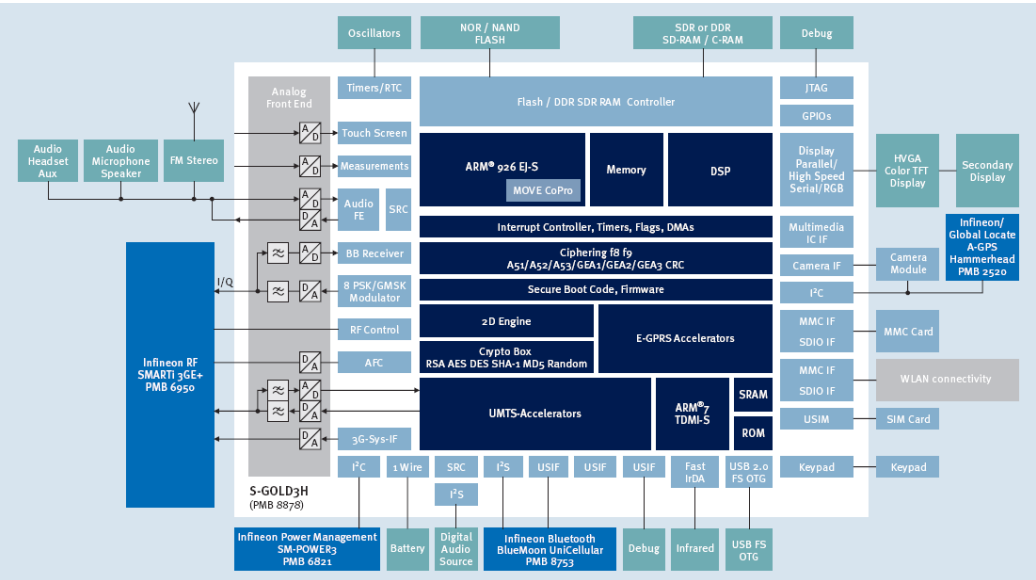


Figure 1.1: Block diagram of the Infineon S-GOLD3H chip for multimedia HSDPA mobile phones.

1.1 Background

Traditionally, MPSoC interconnects have been based on the *shared bus* concept [2]. In other words, a bunch of wires would be laid among all the IP cores in the design. Only one pair of devices would be allowed to communicate over this bus at any point in time; access to the shared medium would be regulated by arbitration, either based on fixed priorities, on time slots, randomly, or on other criteria.

Shared buses have as a main advantage their extreme simplicity, both in conceptual terms and circuit design terms. However, they are completely unsuitable for next-generation MPSoCs, due to two fundamental limitations (Figure 1.2 on the next page):

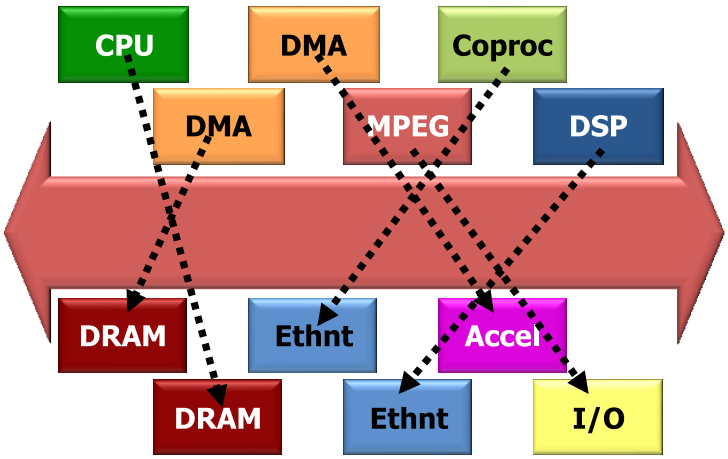
- Their maximum available bandwidth is capped by their shared nature, and this limit can easily be trespassed when the number of attached cores becomes more than a few.

- Their electrical performance degrades dramatically with new lithographic nodes. Since a shared bus is necessarily a structure composed of global wires, as per the INTERNATIONAL TECHNOLOGY ROADMAP FOR SEMICONDUCTORS (ITRS) [3], its propagation delay actually increases with miniaturization. Therefore, with each new chip generation, a shared bus becomes slower in operating frequency, and even slower when compared to the progress in speed achieved by logic blocks. Long wires are also more vulnerable to crosstalk, variability and electrical noise, all of which represent increasingly serious problems in current technologies.

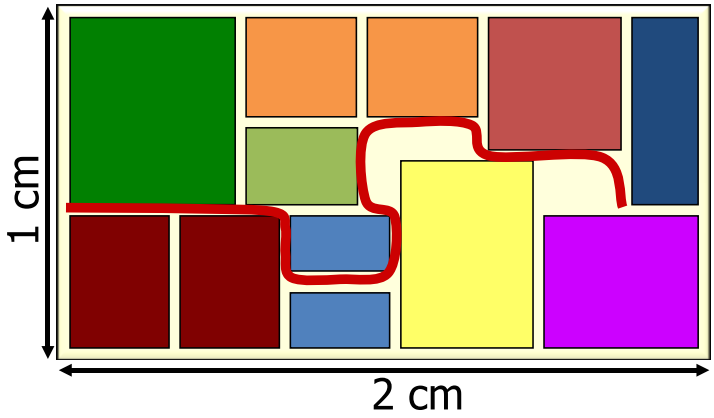
In response to these issues, buses have undergone evolutions [4, 5, 6] in two respects: protocols and topologies (Figure 1.3 on page 5).

Protocol evolution allows for more sophisticated handshakes occurring on the bus, such as multiple outstanding transactions, out-of-order retirement of responses, burst requests, smarter arbitration, *etc.*. These evolutions help in making the best possible use of the limited available bandwidth. While useful in temporarily reducing the extent of bandwidth issues, they still do not provide a long-term solution to the fundamental limitations of buses.

Topology evolutions are a more radical departure from the original shared bus paradigm. The main principle is to deploy multiple buses,



(a) Transactions cannot occur in parallel, even among independent pairs of devices.



(b) Global wiring spanning across the chip has poor electrical performance.

Figure 1.2: Shared bus limitations.

attached to each other by *bridges* or elements called *crossbars* - *i.e.*, devices providing full simultaneous connectivity among all their inputs and all their outputs. The outcome is often called *hierarchical bus* or *multilayer bus*. Hierarchical buses are a much better response to MPSoC design concerns, and in fact most MPSoCs today leverage hierarchical buses.

Even despite these improvements, buses are still a sub-optimal solution for next-generation MPSoCs, due to several factors:

- Hierarchical buses are mostly a manual workaround, by means of which designers try to fix the issues they are presently facing. The development and verification steps have to be performed mostly

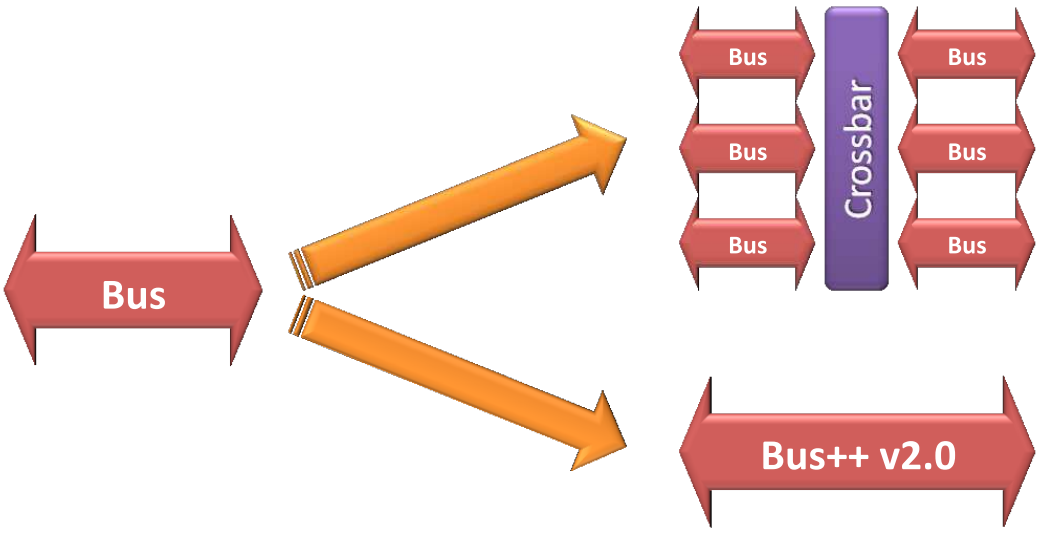


Figure 1.3: Evolution of shared buses towards hierarchical buses and more advanced protocols.

manually, and it is hard to guarantee that the design will scale upon the addition of more IP cores in the next revision of the design. Issues such as deadlock prevention, address mapping, and compliance with performance objectives are among the challenges left to designers.

- From the physical design point of view, hierarchical buses are not much better than shared buses. While allowing for some wire segmentation (wires only have to span regions of the whole system), wires are still normally laid with wide parallelism (typically more than 100 wires for a 32-bit bus and possibly close to 200 for a 64-bit bus), and they still connect multiple entities (a large *fanout*). Together, these issues mean that buses are still electrically inefficient, and difficult to route during physical design.

- Buses normally involve interaction among three agents, usually called *master*, *slave* and *arbiter*, instead of providing a point-to-point, one-to-one handshake. This is unnecessarily making system integration harder.

1.2

Networks-on-Chips: Opportunities and Challenges

A comprehensive solution to on-chip interconnection issues has been proposed in the form of NETWORKS-ON-CHIPS (NOCS) (Figure 1.4).

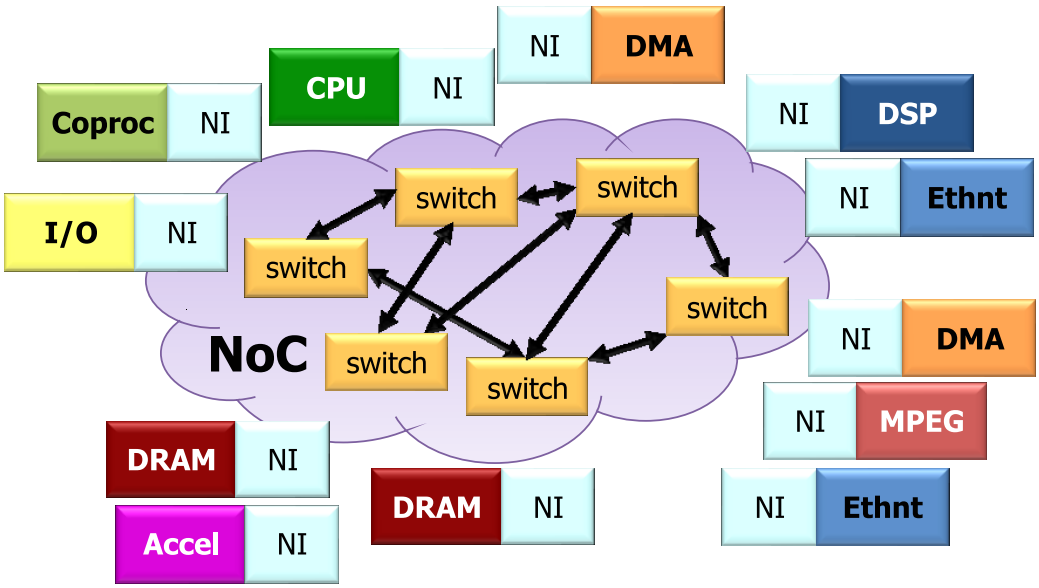


Figure 1.4: Conceptual view of a Network-on-Chip.

NoCs are packet-switching networks, brought to the on-chip level. The rationale is that, since the complexity of on-chip communication is rapidly approaching that of large area systems in terms of actors, it makes sense to reuse some of the solutions devised in the latter space. Therefore, NoCs are based upon topologies of *switches* (also called *routers*) distributing packets around, over point-to-point *links*. Since existing IP cores do not normally communicate by means of packets, NETWORK INTERFACES (NIS) (also called *network adapters*) are in charge of protocol conversion; they convert commands appearing on the pinout of IP cores into packets, and vice versa at the receiving end of a transaction.

NoCs have the potential to bring a large number of advantages to on-chip communication, such as:

- Virtually unlimited architectural scalability. As known from wide area networks, it is easy to comply with higher bandwidth requirements by larger numbers of cores simply by deploying more switches and links.

- Much better electrical performance. All connections are point-to-point. The length of inter-switch links is a design parameter that can be adjusted. The wire parallelism in links can be controlled at will, since packet transmission can be serialized. All these factors imply faster propagation times and total control over crosstalk issues.

- Also due to the possibility of having narrower links than in buses (*e.g.* 20 bits instead of 100), routing concerns are greatly alleviated, and wiring overhead is dramatically reduced. This leads to higher wire utilization and efficiency.

- Faster and easier design closure achievement. Physical design improvements make NoCs, in general, more predictable than buses. Therefore, it is more unlikely that costly respins will be required upon physical design and performance qualification.

- Better performance under load. Since the operating frequency can be higher than in buses, the data width is a parameter, and communication flows can be handled in parallel with suitable NoC topology design, virtually any bandwidth load can be tackled.

- More modular, plug&play-oriented approach to system assembly. IP cores are attached in point-to-point fashion to dedicated NIs; NIs can be specialized for any interface that may be needed, either industry standards such as AMBA AHB or any custom protocol. Potentially any core may be seamlessly attached to a NoC given the proper NI. Computation and communication concerns are clearly decoupled at the NI level.

- Potential for the development of streamlined design flows. While hierarchical buses are often assembled by hand and therefore must be tuned and validated with manual intervention, a network can be designed, optimized and verified by automated means, leading to large savings in design times, and getting a solution closer to optimality.

- A much larger design space. NoCs can be tuned in a variety of parameters (topology, buffering, data widths, arbitrations, routing choices, *etc.*), leading to higher chances of optimally matching design requirements. Being distibuted, modular structures, NoCs can also accommodate differently tuned regions. For example, some portions of a NoC could be tuned statically for lower resource usage and lower performance (*e.g.* by reducing the data width), or could

dynamically adjust their mode of operation (*e.g.* frequency, voltage scaling).

At the same time, NoCs are facing a completely different set of constraints compared to wide area networks. While in the latter environment a switch is implemented with at least one dedicated chip, in a NoC the switch must occupy a tiny fraction of the chip real estate. This means that some of the principles acquired in wide area networking have to be revisited. Some of the challenges lying ahead of NoCs include:

- The tradeoffs among network features, area and power budgets have to be studied from scratch. Policies which are widely accepted in general networking (*e.g.* dynamic packet routing) must be reassessed to evaluate their impact on silicon area.

- Performance requirements are very different in the on-chip domain, also due to the completely different properties of on-chip wiring. Bandwidth milestones are much easier to achieve, since information transfer across on-chip wires is much faster than across long cables. Conversely, latency bounds are much stricter; while milliseconds or even hundreds of milliseconds are acceptable for wide area networks, IP cores on a chip normally require response times of few nanoseconds.

- Contrary to wide area networks, where nodes may often be dynamically connected to and disconnected from the network, in NoCs the set of attached IP cores is obviously fixed. In many applications, it is also relatively easy to statically characterize the traffic profiles of such IP cores. This opens up the possibility of thoroughly customizing NoCs for specific workloads. How to achieve this goal is, however, less clear.

- Design tools for NoCs can be developed, but, as above, how exactly is an open question. The customizability of NoCs, while an asset, is also an issue when it comes to devising tools capable of pruning the design space in search of the optimal solutions. The problem is compounded by the need to take into account both architectural and physical properties; by the need to guarantee design closure; and by the need to validate that the outcome is fully functional, *e.g.* deadlock-free and compliant with performance objectives.

- NoCs are a recent technology, and as such, they are in need of the development of thorough infrastructure. In addition to design tools,

this includes simulators, emulation platforms (such as FIELD PROGRAMMABLE GATE ARRAY (FPGA) boards), and back-end flows for the implementation on both FPGAs and APPLICATION-SPECIFIC INTEGRATED CIRCUITS (ASICs).

1.3 Related Work

1.3.1 MPSoC Interconnects

The continuous time-to-market pressure for consumer embedded devices makes it infeasible to perform a complete redesign each time a new product needs to be developed. Therefore, reuse-intensive platforms such as MPSoCs have become a very attractive solution for the new consumer multimedia embedded market [7]. Several platforms from the major semiconductor vendors (*e.g.* Philips Nexperia [8], TI OMAP [9], ST Nomadik [10]) are available today, exemplifying these paradigms in heterogeneous platforms.

Although MPSoCs promise to significantly improve the processing capabilities and versatility of embedded systems, one major problem in their current and future design is the effectiveness of the interconnection mechanisms between the internal components, as the amount of components grows with each new technological node. Traditional SoC interconnects, as exemplified by AMBA AHB [11], are based upon low-complexity shared buses, in an attempt to minimize area overhead. Such architectures, however, are not able to cope with the heterogeneous and demanding communication requirements of MPSoCs, motivating the need for more scalable designs. The most advanced SoC communication architectures used in industry today represent evolutionary solutions with respect to such shared buses. For instance, the Sonics MicroNetwork [6] is a TDMA-based bus which can easily adapt to the data-word width, burst attributes, interrupt schemes and other critical parameters of the integrated cores, while providing very high bandwidth utilization. Another example is the STBus interconnect [5], a high-performance communication infrastructure that allows to instantiate shared busses as well as more advanced topologies such as partial or full crossbars. Recently, the AXI [4] evolution of AMBA AHB has focused on decoupling communication and computation concerns as much as possible, by providing only specifications for

the core/interconnect interface, and leaving ample freedom to interconnect designers. In this sense, a similar role is played by the OPEN CORE PROTOCOL (OCP) [12] interface specification, which is meant to serve as a standard, composable interfacing specification for next-generation MPSoCs, and which is already in use today by several vendors [13].

In addition to increasing bandwidth and performance demands, a new set of problems is coming from the physical side. As the semiconductor industry reaches deep submicron technologies [14], power density and process variations become critical design concerns for embedded systems as well; thus, predictability in the design of on-chip interconnects is becoming as important as the provided bandwidth. Further, well-known trends in wire propagation delay scaling [3] point out a very concrete problem in the layout of global wires, making the further development of buses very problematic. Hence, new paradigms and methodologies that can design efficient, power-effective and reliable interconnects for MPSoCs are a must nowadays.

Networks-on-Chips

NETWORKS-ON-CHIP (NOCS) have been suggested as a promising solution to the aforementioned scalability problem of forthcoming MPSoCs [15, 16, 17, 18, 19]. NoCs build on top of the latest evolutions of bus architectures in terms of advanced protocols and topology design, and, by bringing packet-based communication paradigms to the on-chip domain, they address many of the upcoming issues of interconnect fabric design better than buses [20]. In terms of performance scalability, in NoCs the available bandwidth can be boosted simply by increasing the number of links and switches, therefore guaranteeing support for designs of extreme complexity, as wide-area networks testify. Furthermore, compared to irregular, bridge-based assemblies of clusters of processing elements, NoCs also help in tackling design complexity and verification issues [17, 21].

One of the earliest contributions in this area is the Maia [22] heterogeneous signal processing architecture, based on a hierarchical mesh network. In [23], the authors sketch the architecture of a VLSI multi-computer using 2009 technology, where a chip with 64 processor-memory tiles is envisioned. Communication is based on packet switching.

Most early NoC proposals are packet switched and exhibit regular structure. The NOSTRUM network [24] adopts a mesh based approach. The SCALABLE PROGRAMMABLE INTEGRATED NETWORK (SPIN) [25] is another regular, fat-tree-based network architecture. The Linköping SoCBUS [26] is a two-dimensional mesh network which uses a packet

connected circuit to set up routes through the network. In [27] the use of the *octagon* communication topology for network processors is presented. Moreover, the implementation of a *star-connected* on-chip network supporting *plesiochronous communication* among system components is described in [28]. These NoCs provide scalable network fabrics for homogeneous system (*e.g.* symmetric chip-multiprocessors), but they do not allow arbitrary heterogeneous topology instantiation. Unfortunately, many real-world MPSoCs exhibit a large degree of heterogeneity, both in communication requirements (clock frequency, data width, injected bandwidth) and in physical terms (size and positioning of the IP cores on the floorplan).

Significant steps in the direction of instantiation-time flexibility have been taken in the Æthereal [29] NoC design framework, which aims at providing a complete infrastructure for developing heterogeneous NoCs with end-to-end quality of service guarantees. The network supports GUARANTEED THROUGHPUT (GT) for real time applications and BEST EFFORT (BE) traffic for timing unconstrained applications. Æthereal's NI is highly configurable (it supports several session-layer standards, a variable number of ports, *etc.*), but the switch architecture is quite rigid. Furthermore, from the implementation viewpoint, both the NI and the switch make use of custom hard-macro FIFO buffers. These structures are not synthesizeable in standard cell flows and they must be manually re-tuned when migrating to new technologies.

Support for heterogeneous architectures requires highly configurable network building blocks, customizable at instantiation time for a specific application domain. For instance, the Proteo [30] NoC consists of a small library of predefined, parameterized components that allow the implementation of a large range of different topologies, protocols and configurations.

In this work, we propose the ×pipes NoC, which pushes the configurability approach to the limit, by instantiating an application-specific NoC from a library of synthesizeable soft macros (network interfaces, switches and links). The components are highly parameterizable and provide reliable and latency-insensitive operation, while minimizing both latency and implementation cost.

Physical Design and Cross-Benchmarking

A major advantage of NoCs is that the interconnect structure and wiring complexity can be fully controlled by matching network topology with physical constraints. When the interconnect is structured, the number of timing violations that may occur during physical design (floorplanning

and wire routing) is minimized. Such design predictability is critical for today's MPSoCs to achieve timing closure. It leads to faster design cycles, to a reduction in the number of design re-spins and to a faster time-to-market.

As the wire delay to gate delay ratio is increasing with each technological generation, having shorter wires is even more important for future MPSoCs. This property of NoCs means that, for a proper assessment, it is crucial to take into account the physical implementation phase. The synthesis flow of NoCs has been explored by several groups. Layouts are presented in [29, 31], a test chip is shown in [32], and an FPGA target is provided for [33].

In this work, to take into account as many key effects as possible, we establish a flow that takes our NoC topologies down to placed&routed layouts. This flow allows us to derive final frequency, area and power figures for the NoC blocks in order to perform complete studies of different overall NoC interconnects. In our analyses and studies of on-chip interconnects we cover NoCs implemented with the proposed design flow using three different technology libraries (130, 90 and 65nm), such that we can provide conclusions for a very representative part of the design spectrum.

One key topic which has not yet been extensively covered is studying how NoCs compare to more traditional interconnects. In [34], an analytical methodology is illustrated to compare NoCs of arbitrary topology (a shared bus and a crossbar are provided as examples) also taking into account area, frequency and power metrics. However, some assumptions of this work (such as the relative cost of wiring *vs.* logic) do not seem to be fully confirmed when considering actual fabrics. In [35], a synthesis-aware flow is presented to characterize the Hermes NoC; PI Bus is used as a benchmark for performance metrics, but not for area and power analysis. Further, PI Bus is not representative of current, widely used high-performance interconnects.

In this work, we tackle this shortcoming by providing a complete cross-benchmarking experiment at the 130nm technology node, showing performance, area, power, energy and predictability figures. Moreover, we extend our analysis by providing NoC scaling results down to the 65nm node, and verifying the impact of routability, leakage, clock tre distribution, *etc.* effects.

A very interesting study on the impact of technology scaling on the energy efficiency of standard topologies (such as meshes, tori and their variants) has been presented in [36]. The current work differs from this research in two ways: first, we consider the design of platform-specific

NoC topologies and architectures. Second, we use a complete design flow that is integrated with standard industrial tool chains to perform accurate physical implementations of the NoCs.

Flow Control Protocols

The simplest flow control mechanisms are bufferless. Memoryless switches are employed in [37]: in case of congestion, packets are emitted in a non-ideal direction, also called deflective routing. The introduction of guaranteed bandwidth in [38] was made possible by loop containers and temporally disjoint networks.

Providing QoS by establishing circuits between communicating nodes requires some buffering resources. A novel hybrid circuit switching with packet based setup is reported in [26], which needs minimum buffering resources, capable of holding just a request packet. A circuit switched NoC using time division multiplexing is reported in [39].

In buffered flow control, NoC performance is tightly related to the amount of buffering resources implemented. A methodology to size the FIFOs in an interconnect channel containing one or more FIFOs in series as a function of system parameters (data production and consumption rate, burstiness, *etc.*) is reported in [40], pointing out the impact on performance.

Credit based flow control was described in [41], for use in ATM networks, and in [42] for use in interconnection networks. It is applied on a hop-by-hop basis. The upstream node keeps a count of the number of free flit buffers at the destination node. Credit based flow control is used in [43, 44]. It is also employed in the asynchronous multi-service level QNoC router in [45].

Flow control in the SoCIN NoC architecture is based on the handshake concept [33]. When a sender puts data on the link, it activates the related VALID signal. When the receiver is ready to consume the validated data, it activates the corresponding ACK signal. Both handshake and credit based flow control are supported in the revised SoCIN architecture called ParIs [46].

The router in [47] handles both best effort (BE) and guaranteed throughput (GT) traffic. The GT router relies on a time division multiplexing mechanism. Slot tables in the routers divide up bandwidth per link and switch data to the correct output at each time slot. Credit based flow control is used in the BE router at the link level, but also for end-to-end flow control in the network interfaces [44].

The link control mechanism of NoCGEN uses a request, grant and ready handshake to enable flow control on point-to-point links [48].

Finally, ON/OFF flow control can greatly reduce the amount of upstream signaling [42]. The upstream internal state is a single control bit that represents whether the node is permitted to send (ON) or not (OFF). A feedback signal is sent upstream only when it is necessary to change this state, for instance when the number of free downstream buffers falls below a certain threshold.

In this work, we present a comparative analysis on three representative flow control schemes that we implemented on ×pipes: ACK/NACK (retransmission-based), STALL/GO (similar to ON/OFF) and T-Error [49] (based on STALL/GO, but with additional logic to compensate for timing-related errors - which can be used to either increase the tolerance to faults or to voluntarily overclock or overstretch the links; a frequency boost of around 50% can be achieved while introducing a much smaller overhead for the compensation of the resulting violations). All three protocols are extended and studied in presence of pipelined links, *i.e.* already accounting for propagation delay phenomena which are expected to be dominant in deep submicron technologies [3].

NoC Area and Power Modeling

Based on our back-end physical implementation flow, we construct area and power models for NoC components. This modeling activity is crucial not only to better understand the potential for optimization in the ×pipes NoC, but also to drive our topology generation flows.

Power models and simulators for processors and memories have been proposed in an extremely large body of research [50, 51]. Interconnects have also become the focus of research [52], due to their increasing role in the hardware budget of recent and future systems; for example, the on-chip network of the MIT RAW chip multiprocessor is taking 36% of the chip power budget on average [53].

Some models of NoC hardware cost have already been proposed in previous literature. Results in [54] are derived from a mix of experiments on template circuits and from technology trends, and are specifically aimed at wide applicability. Therefore, even though they have been used for design space exploration [55] and in association with high-level traffic injection models [56], they do not guarantee maximum accuracy within an architecture-specific CAD flow. The main advantage of these techniques is flexibility and fast deployment. We see them as complementary to our approach, especially for initial exploration when the NoC com-

ponent library is not available yet.

The approach in [57], on the other hand, attempts to build a cycle-accurate power model of a target router instance. However, several major points differentiate our approach. First, we build a model which is parametric not only on traffic-related events, but also on the architectural knobs of the design. Second, we include an area model in the exploration. Third, our model can be more readily adopted within a CAD mapping flow; this is both because we express the model as a function of architectural parameters, and because we provide a high-level dependence on traffic variables, instead of a cycle-by-cycle one. Fourth, we strive to make our approach as applicable as possible in real-world conditions, including the hard-to-model peculiarities of the behaviour of synthesis tools when aiming for maximum frequency operation, and placement and routing issues. Fifth, we propose a fast characterization mechanism, by means of which model coefficients can be quickly derived with a minimal amount of synthesis runs.

In [58], a framework for NoC exploration is presented; the framework includes a power modeling flow. The power model features very limited dependence on architectural parameters and does not seem to account for the configuration knobs of synthesis tools. No area model is provided.

In [59], a bit energy modeling flow is proposed to compare different switch fabrics in IP network routers. The approach is focused on the cost for transmitting bits from input to output ports, and while bit pattern-accurate, it is only focused on comparing router topologies against each other. The authors of [60] propose a model based on transistor count, while in [61], which is focused on FPGAs, switch cardinality is the main parameter. None of these models is meant for simultaneously accurate, parametric and fast representation of power consumption, *i.e.* suitable for design space exploration within a CAD environment.

1.3.2 Simulation and Traffic Generation

Several works have described performance evaluation environments for interconnects. For example, a cycle-precise simulator speeding up performance of the well-known SystemC simulation engine is described in [62], and used in [18] to compare two communication architectures (SPIN micronetwork and PI-Bus). Transaction level models for the AMBA architecture are described in [63]. A modeling framework for communication architectures with accompanying simulation tools is presented in [64], and is based on a hierarchical class library, whereby new communication architectures can be developed based on reusable components. A C++ model-

ing library developed on top of SystemC is at the core of IPSIM design environment, which separates IP modules into behaviour and communication components [65].

However, the enabling technology for communication optimization is system-level performance analysis. This is because, in order to optimize interconnect performance, not only the accuracy of the interconnect model should be high; in fact, the realism of the injected traffic is equally crucial. Therefore, analysis frameworks are an emerging research area. Several approaches have been proposed:

- The entire system can be simulated using models of the components and their communication at different levels of abstraction [66, 67].
- Static system performance estimation techniques, including models of the communication time between system components. Time estimates are usually either optimistic (ignoring dynamic effects such as bus contention) [68] or pessimistic (assumption of worst case scenario) [69].
- Mixed approaches, deriving set of traces from an initial cosimulation of the system (assuming abstract data transfers), and forwarding them to an analysis tool that, for a specified communication architecture, comes up with system performance estimates [70].

The second and third approach target communication architecture space exploration early in the design stage. The first one is necessary whenever high-accuracy system level simulations have to be performed. This is the case of comparisons between a restricted set of architectural options, wherein dynamic effects, for instance bus-contention-related latency, or the dependency of performance on application generated traffic, can make the difference.

When simulation falls short, formal approaches can be applied to the MPSoC domain, offering systematic verification based on well-defined models [71]. A synchronous, finite state machine based method for modeling communication aspects of SoCs is presented in [72]. A performance model to abstract a general class of reconfigurable SoC architectures is described in [73].

A limitation of the above mentioned approaches is that the performance of the communication architectures is always derived under non-realistic workloads. Traditionally, parameterized statistic traffic generators are used [74, 61, 75]; in spite of their generality, they prevent designers from assessing performance in presence of real-life workloads and make it

difficult to account for dynamic effects such as bus contention. The work in [76] goes in this direction.

In this dissertation, we present MPARM, a complete virtual platform environment [77, 78, 79] which enables cycle-accurate exploration of the MPSoC design space. One of the salient features of MPARM is that, in addition to supporting alternative interconnect models (including buses and NoCs), it also allows fot a range of other design variables in the environment. For example, MPARM allows for picking a variety of IP cores, memory hierarchies, and software stacks, allowing designers to optimize the interconnect design based on realistic traffic injection.

Flexible Modeling of IP Cores

In order for virtual platforms to be useful, they need to have two properties: a wide portfolio of alternative models to plug in the system, and as much openness as possible in terms of model configuration and modification. Unfortunately, academic platforms are typically lacking in completeness, as they are often built with limited resources to test some specific MPSoC aspect. The SimpleScalar [80] framework stands out for its feature set, but is essentially a single-processor model with an unclear scalability path towards multiprocessor systems. Many other projects exist [81, 82, 83, 84], but their scope seems currently to be too limited for full MPSoC exploration. On the other hand, industrial platforms are typically lacking in openness. Synopsys CoCentric System Studio [85], CoWare ConvergenSC [86], the ARM RealView MaxSim [87] and others [88, 89] spring to mind as some of the best known industrial environments. They all share a plug-and-play approach of licensed IP blocks whose models are provided in encrypted form. As a result, if the internal architecture is to be investigated and optimized (*e.g.* with the addition of custom instructions or data lanes), alternative open blocks must be written by hand, taking much of the appeal of IP portfolios away.

A subset of the MPSoC design space is covered by existing ASIP design tools. Tools available today can be roughly categorized into three categories, ARCHITECTURE DESCRIPTION LANGUAGE (ADL) driven, template architecture based, and predefined component library based. Within the first category, there are tools like EXPRESSION [90], LISATek [91], archC [92] or CHESS [93]. However, little information is publicly available about their usage in a heterogeneous MPSoC simulation environment, especially in an open one. Unlike the ADL driven approach, a partially configurable processor is used as template by the tools in the second category, where Tensilica [94] is a popular representative. ASIPMeister [95] has a

predefined library of processor micro-architecture components and falls into the third category.

A contribution we bring in this work is to integrate the LISATek framework within MPARM, by plugging its core models within the MPARM platform. This brings together the best of both worlds. MPARM gains the ability to instantiate a large variety of freely modifiable IP cores, inclusive of state-of-the-art, industrial-strength tooling for configuration and debugging. On the other hand, LISATek can now enjoy a platform framework wehereby all the other degrees of freedom of the MPSoC can be explored.

Traffic Generation

Cycle-accurate system simulation suffers from two disadvantages. First, it is time-consuming to code cycle-accurate models of IP cores; these models may actually only become available very late in the development process of MPSoCs, making them useless for exploration and optimization. Second, cycle-accurate models are intrinsically slow, making design space exploration a lengthy task. As a workaround, IP emulation devices such as TRAFFIC GENERATORS (TGS) have been traditionally employed. Several approaches and models have been proposed.

In [96], a stochastic TG model is used for the interconnect exploration; the IP behavior is statistically represented by means of uniform, Gaussian, or Poisson distributions. A similar approach in [97] uses random and semi-deterministic distributions. The IP model used for NoC optimization in [98] takes into account the nature of MPSoC traffic such as real-time, short-data access, bursty, *etc.*, however the injection rate is governed by statistical methods. In [99], an extra dimension of "self-similarity" is added to the stochastic model which is argued to assist in precise characterization of multimedia traffic by examining the correlations in traffic traces at the macroblock-level. Despite the refinements, the inherent probabilistic nature of the statistical approaches makes it less accurate, as each TG injects traffic in complete isolation from every other. As surveyed in [100], such stochastic models are widely popular for analysis of macro-networks, *e.g.* the Internet, that exhibit such behaviour; unfortunately, this paradigm is unlikely to hold in an MPSoC environment. To overcome the speed limitation of simulation-based approaches, FPGA-based emulation platforms have also been proposed [101]. However, these approaches again leverage stochastic or trace-driven model to generate traffic, which, as addressed before, are not sufficiently accurate for MPSoC performance optimization.

A modeling technique which adds functional accuracy and causality is TRANSACTION-LEVEL MODELING (TLM), which has been widely used for MPSoC design [102, 103, 104, 105, 106, 107]. In [105, 106], TLM has been used for bus architecture exploration. The communication is modeled as read and write transactions towards the bus. Depending on the required accuracy of the simulation results, timing information such as bus arbitration delay is annotated within the bus model. In [106] an additional layer called CYCLE COUNT ACCURATE AT TRANSACTION BOUNDARY (CCATB) is presented. Here, the transactions are issued at the same cycle as that observed in BUS CYCLE ACCURATE (BCA) models. Intra-transaction visibility is traded off for a simulation speed gain. An average speedup of 1.55x is reported. While modeling the entire system at TLM, both [105] and [106] present a methodology for preserving accuracy with gain in simulation speed. Such models are efficient in capturing regular communication behaviour, but the fundamental problem of capturing system unpredictability in the presence of OS and interrupts is not addressed.

In this work, we propose RIPE, a reactive traffic generation framework, capable of not just replacing IP cores, but in fact capable of emulating the behaviour of real applications running on such IP cores. This is achieved by means of a simple instruction set processor, which can be programmed to inject traffic also based on environmental conditions, such as synchronization conditions and interrupts. Only in this way it becomes possible to abstract the behaviour of complex parallel applications for MPSoCs, and to faithfully recreate it on any interconnect platform. RIPE can operate in two ways, either by manually writing programs to emulate any desired behaviour, or by extracting traces and behaviours from simulations in different contexts.

A transformation methodology of high-level simulation traces with cycle-true information from the target architecture, *e.g.* memory distribution and communication details, is presented in [108] and [96]. In [108], based on accurate information, different rules are specified for inserting and ordering synchronization events in the output execution trace, while in [96] a trace-based communication graph is adjusted with interconnect-specific details, such as connection setup time, burst size, *etc.*. The RIPE approach, in contrast, is to identify synchronization events based on system information and abstract them for communication refinement. We believe that the RIPE model and the approach presented in [108] are complementary in addressing trace-based MPSoC analysis from functional to cycle-true abstraction.

In [109], a commercial TLM-based reactive workload generation framework is presented that is somewhat similar to our RIPE approach, wherein

users can configure traffic patterns for handling synchronization and inter-IP events. Primitives for timing-dependent behaviour are provided, so that the user can trigger actions which do not depend on application flows but on simulation time. The RIPE approach however supports multi-threading, which is required for interrupt-driven OS context switches, and traffic generation at multiple levels of abstraction, including in a cycle- and bit-true environment.

Other commercial efforts also exist, including the OpenVERA [110] language and toolchain, that model concurrency and synchronization. However, our approach is focused on maximum accuracy of results, while OpenVERA is mostly focused on the verification issue, providing a flow from higher abstraction levels to RTL.

1.3.3 Topology Design

Early works on NoC topology design assumed that using regular topologies, such as meshes, like in macro-networks, would lead to regular and predictable layouts [111, 21]. While this may be true for designs with homogeneous processing cores and memories, it is not true for most MPSoCs as they are typically composed of heterogeneous cores in terms of area and communication requirements. A regular, tile-based floorplan, as in standard topologies [21], would result in poor performance, with large power and area overheads. Moreover, for most state-of-the-art MPSoCs (like Philips Nexperia [8] or TI OMAP [9]) the system is designed with static (or semi-static) mapping of tasks to processors and hardware cores, and hence the communication traffic characteristics of the MPSoC can be obtained statically. Thus, an application-specific NoC with a custom topology, which satisfies the design objectives and constraints, can be envisioned, and is critical to have efficient on-chip interconnects for MPSoCs.

A large body of research works exists in synthesizing and generating bus-based systems [112, 113]. Floorplan-aware point-to-point link design and bus design methodologies are presented in [114, 113]. While some of the design issues in the NoCs are similar to bus based systems (such as link width sizing), a large number of issues such as finding the number of required switches, sizing the switches, finding routes for packets, *etc.* are new in NoCs.

Methods to collect and analyze traffic information that can be fed as input to the bus and NoC design processes have been presented in [115, 113]. Mappings of cores onto standard NoC topologies have been explored in [111, 116, 117, 118]. In [117], a unified approach to mapping, routing

and resource reservation has been presented. However, the work does not explore topology design process. The NoC design process for supporting multiple applications has been presented in [119].

Important research in macro-networks has considered the topology generation problem [120]. As the traffic patterns on these networks are difficult to predict, most approaches are tree-based (like spanning or Steiner trees) and only ensure connectivity with node degree constraints [120]. Hence, these techniques cannot be directly extended to address the NoC synthesis problem. Application-specific custom topology design has been explored earlier in [121, 122, 123, 24]. The works from [121, 122] do not consider the floorplanning information during the topology design process. In [124], a physical planner is used during topology design to reduce power consumption on wires. However, the work does not consider the area and power consumption of switches in the design. Also, the number and size of network partitions are manually fed. In [123], a slicing tree based floorplanner is used during the topology design process. This work assumes that the switches are located at the corners of the cores and it does not consider the network components (switches, network interfaces) during the floorplanning process. Also, deadlock free routing, which is critical for custom NoC designs is not supported in the work. Moreover, a complete design space exploration, from architectural parameter setting to simulation is not presented.

In this work, we introduce a complete topology design framework capable of solving the shortcomings of previous approaches. We design and leverage a tool, called SunFloor, which is able to design customized topologies and to map cores to them, while optimizing a choice of latency and power metrics and respecting constraints on area, power and performance. SunFloor leverages, or produces at the user's choice, floorplanning information when generating topologies; this guarantees that key aspects, such as wire lengths, will be accounted for, both by making sure that directly communicating entites are placed close to each other on the floorplan, and by providing link pipelining on long wires. For proper operation, SunFloor leverages upon area and power models of the NoC components, that we capture for each technological back-end.

Deadlock Freedom

The use of turn models to avoid deadlocks in mesh and torus networks has been presented in [125]. There has been a large body of work focused on developing routing-dependent deadlock-free operation for interconnection networks [126, 125, 127, 128, 129]. Several other works exist in the area

of recovering from deadlocks in networks [130, 131]. Routing-dependent deadlock avoidance strategies have been presented for meshes [111] and custom NoC topologies [132, 117, 133].

Several works on application specific NoCs [111, 122, 123], however, do not address the crucial issue of message-level deadlock avoidance, which is critical for proper system operation. The deployment of logically separated networks to avoid message-dependent deadlocks has been utilized in several industrial multi-processors, such as [134, 135, 136, 137]. The use of physically separated networks to remove message-dependent deadlocks is also used in many designs, such as [138, 139]. In [19], message-level deadlock freedom is achieved by a different mechanism than using logically or physically separated networks. That work utilizes an end-to-end flow control scheme, which ensures that messages are sent from the sender only when the receiver has enough buffering resources to store them. This is coupled together with a network design that uses time division multiplexing to divide the network resources among the various communicating elements, providing guaranteed throughput to connections. This leads to a buffering free network for such connections and removal of message-level deadlocks. The deadlock avoidance mechanism using such protocol is presented in [140]. However, it is important to notice that not many NoC design provide facilities for end-to-end flow control, limiting the applicability of this technique.

In this work, we present techniques to design NoCs which are free from both routing-dependent and message-level deadlock issues, while incurring a minimum overhead penalty to do so. Our methodology does not rely on expensive hardware support; instead, deadlocks are avoided at the topology generation level. Routing-dependent deadlocks are avoided by forbidding certain turns [129] for packet routes, while message-level deadlocks are avoided by simply not mapping conflicting message types over the same links.

1.3.4 Fault Tolerance in NoCs

Regarding fault tolerance mechanisms at the micro-architectural level, reliability work on soft errors is presented in [141]. Redundant components can be used to increase processor lifetime and system reliability [142]. At the system level, dynamic fault-tolerance management [143] is shown to improve system reliability in embedded systems. Different metrics are proposed to estimate the effect of soft failures with particular attention to energy efficiency, computation performance and battery lifetime tradeoffs. An interesting approach to simultaneously achieving SoC reliability and

high efficiency is explored by [144] and [145]. There, the SoC is aggressively configured to comply with *typical case* constraints, thus delivering high performance and low power; in worst case conditions, which rarely occur, errors appear, but are transparently corrected either by a built-in checker or by timing error-tolerant circuitry.

The test and repair of SoCs, and more specifically of their memories, which are a critical component in this respect, has been extensively explored [146, 147]. To provide reliable operation, the use of SINGLE-ERROR-CORRECTING-MULTIPLE-ERROR-DETECTING (SEC-MED) codes is already integrated in many on-chip memories [148]. Recent studies show that different program behavior patterns can be identified, and can be used to generate various custom error correction mechanisms for different memory portions [149]. A very important research area is represented by the development of memory cores with built-in self-test logic and spare storage resources [150, 151, 152]. While all these approaches have been demonstrated to be robust, they necessarily come at an area cost.

In this work, we propose a novel approach to improving the fault tolerance of MPSoCs, by leveraging the communication backbone to enable redundancy policies at minimal cost. Our approach is based on the main idea of deploying backup devices (namely, memories) somewhere in the chip; NoCs allow for doing this with a minimum of additional complexity but with maximum flexibility. NoCs can also transparently handle the backup functionality and the switchover mechanism upon actual failures, while incurring minimal overhead to support these additional features.

In order to propose this scheme, we rely on previous works on fault tolerance for NoCs themselves. For example, the authors of [153] describe a way of designing reliable NoC links by comparing them to radio channels, while link architectures can be tuned [49] to tolerate timing errors of up to 50% of the reference clock period. Data retransmission schemes are well known in wide area networks and have been applied to NoCs, for example in [154]; coupled with error detection circuitry, they allow for soft error recovery upon NoC links.

Separate units (such as for example pre-existing microcontrollers [155]) have also been deployed in SoCs to supervise the status of on-chip memories. With respect to these techniques, we choose to leverage a built-in support in the underlying SoC communication infrastructure to minimize the silicon overhead. Additional advantages of this choice are complete transparency to the software designer and the avoidance of any performance disruption upon fault occurrences. By leveraging NoCs as the communication backbone, our approach also guarantees maximum scalability.

1.3.5 NoCs for 3D Chips

Chip stacking is emerging as a way to sustain the increasing demand for on-chip functionality and performance, which is paired with a push towards package miniaturization and modularity. A number of technologies for 3D chip manufacturing have been explored in recent years, including transistor stacking [156], die-on-wafer stacking [157], wafer stacking [158], chip stacking [159]. Wafer stacking approaches represent one of the most promising avenues for the implementation of high-performance yet inexpensive (multiple 3D chips can be processed in a single pass) three-dimensional ICs. Wafer stacking relies on THROUGH-SILICON VIAS (TSVS) [160] for vertical connectivity, guaranteeing low parasitics (*i.e.* low latency and power) and, if needed, extremely high densities of vertical wires (*i.e.* high bandwidth). Tezzaron Semiconductor Corporation [161] and IBM Technologies [162] are active players in this field; the major differences between their processes are in wafer bonding methodologies and TSV formation. The former resorts to via formation followed by high-temperature wafer bonding, so that electrical connectivity and bonding strength are guaranteed by thermocompression. The latter uses oxide fusion bonding at room temperature, allowing a very high precision alignment, while vias are formed after the wafers have been bonded together. Post-silicon nano-scale 3D interconnections have also been recently investigated [163], but large scale availability of these technologies in the near future is uncertain.

As a consequence of this fast development, and since NoCs are already emerging as the interconnection paradigm for planar chips, research has recently been undertaken on 3D NoCs. For example, in [164, 165] alternative ways of interconnecting 3D chips are contrasted; namely, the authors focus on several variants of 3D meshes, stacked meshes, stacked tori, *etc.*. The main focus of the authors is on topologies and on performance metrics, while the physical implementation is not studied in depth. In [166], the authors propose a dimension decomposition scheme to optimize the cost of 3D NoC switches, and present some area and frequency figures derived from a physical implementation. The fundamental assumption of their work is that a regular, homogeneous NoC is the best solution for a 3D design, and therefore the next logical step is to reduce the cost of each required building block. However, we believe that, for such complex designs as stacked 3D chips, which are likely to mix logic layers with memory layers and even more uncommon functionality, heterogeneity will likely be significant, especially along the vertical axis.

In this work, we present results aimed at solving those shortcomings

and enabling efficient, heterogeneous 3D NoCs. First, we build accurate models of the parasitics involved in vertical links for 3D stacked chips. In this respect, our work is orthogonal and complementary to the ones mentioned above; to the best of our knowledge, no previous work fully characterizes the vertical interconnections for use in NoCs, especially with respect to physical implementation and timing requirements. Next, we propose a more general approach, where the designer is allowed to choose among planar and vertical communication on a switch-by-switch basis, without any topological constraint. We also show a prototype, partially automated design flow to enable the design of such heterogeneous topologies, showing example layouts.

We subsequently focus on system-level integration issues. In 3D chips, the distribution of clock signals is likely to incur major skew issues [167, 168]. This means that fully synchronous paradigms, such as that natively used by ×pipes, are unworkable. A large body of research exists on asynchronous NoC design styles. For example, the CHAIN network [169] is completely based on clockless circuit design techniques. Other asynchronous NoC libraries include MANGO [170], ASPIN [171] and NEXUS [172]. ANOC [173] is based on a Quasi-Delay-Insensitive circuit design. Specific network building blocks are presented for example in [45] and asynchronous link design is tackled in [174]. The main goals of asynchronous NoCs have traditionally been lower power consumption than synchronous alternatives, increased tolerance to delay variability, and reduced electromagnetic emissions [175]. Despite all the research efforts, however, the actual physical implementations of asynchronous NoCs [176, 177] are few and limited in complexity (few millions of gates, 130nm technology). This is often attributed to the current lack of fully mature synthesis toolchains, simulation environments and testing infrastructures, hindering industrial implementations. Suitable component libraries are also very difficult to build and characterize.

GLOBALLY ASYNCHRONOUS LOCALLY SYNCHRONOUS (GALS) approaches do not disrupt as much the existing design flows. GALS systems [178, 179, 180] attach together a number of synchronous building blocks, and provide asynchronous facilities for the inter-block communication. While some of the tool maturity issues mentioned above still hold, the encapsulation of mixed-clock concerns within well-defined boundaries, which can be validated separately, provides a more conservative, and possibly more promising, solution to the interconnection issue. Several ways to synchronize clock domains at the boundaries exist, such as interleaving pipeline registers, using dual-clock FIFOs, adding programmable delays [181], deploying synchronous-to-asynchronous wrap-

pers [178]. Although some of these solutions (for instance, dual-clock FIFOs) are very flexible, allowing for arbitrary clock frequencies in the sender and receiver domain, they all have one or more drawbacks, ranging from robustness to implementation complexity, from high latency to large area overhead. Some solutions have instead been specifically tuned only for the relatively simpler problem of mesochronous signaling, and have therefore been focused on low complexity and ease of implementation in existing tool flows. Two recent papers [182, 183] both suggest to implement the boundary interface with a source-synchronous design style, and propose some form of ping-pong buffering to counter timing and metastability concerns.

In this work, we improve on these papers by studying such synchronizers inside of a NoC layout for a 3D chip, optimizing them for the requirements of 3D NoCs, and considering full duplex communication with flow control.

1.4 Contributions of This Dissertation

This dissertation aims to shed light on the tradeoffs in the design of NoCs, and to push forward the bar of state-of-the-art NoCs. Its main contributions are:

- The development of a REGISTER TRANSFER LEVEL (RTL) NoC component library, called ×pipes. This library is highly configurable and optimized for high performance and minimum resource usage.
- The integration of the NoC component library in a cycle-accurate simulation and traffic generation environment, called MPARM [79], for validation, characterization and optimization purposes.
- The integration of the NoC component library into a complete design flow, spanning from application requirements to finalized FPGA and ASIC physical implementations (Figure 1.5 on the facing page). This unique design flow is instrumental to proving the viability of the proposed NoC solution.
- The analysis of NoC performance and cost in a variety of environments and conditions. Analyses will be shown evaluating the performance of alternative NoC implementations; comparing NoC perfor-

mance with hierarchical bus performance; and comparing NoC performance across manufacturing technologies. Results include architectural simulations and back-end analysis on operating frequency, area and power consumption.

- The extension of NoCs with facilities able to guarantee better fault tolerance. This is a key feature in today's environment, featuring increasing variability and uncertainties in chip manufacturing.

- Finally, as an outlook on possible future applications of NoCs, the description of an initial NoC implementation for upcoming stacked ("3D") chips.

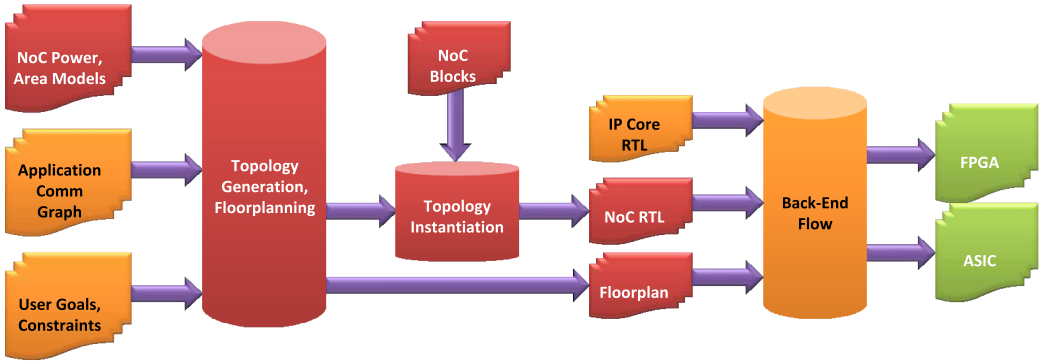


Figure 1.5: The complete proposed NoC design flow.

This dissertation presents work jointly carried on by the author and by several co-authors. While efforts will be made to focus mainly on the areas in which the author was primarily involved, it is actually impossible to completely decouple the contributions. Further, describing the complete framework in which the author's research was performed is helpful for a much better understanding of the goals and the achievements of this effort. Appropriate credit to major co-authors will be provided across this dissertation. A list of the papers on which this work is based, complete with the names of all the co-authors, is supplied in Appendix A on page 259.

The author would like to stress that NoCs are a relatively recent body of research, with roots dating to 2001, meaning that research opportunities were (and still are) numerous. This is proven by the fact that, during these years, NoCs have enjoyed phenomenal success at the academic

level, with hundreds of publications recorded to date and a dedicated conference since 2007. Therefore, it is materially impossible to provide a comprehensive overview of all the possible architectural degrees of freedom, of all the possible research trends, and of all the relevant related work. This dissertation merely reflects the research choices done by the author and his co-authors based on the information and analyses available to them (which will be substantiated wherever possible across the present dissertation), and based on time constraints.

1.5 Dissertation Outline

In Chapter 2 on page 31 we provide a brief discussion of existing bus protocols as available in two mainstream bus architectures, AMBA AHB [11] and AMBA AXI [11] by ARM Ltd. and STBus by STMicroelectronics [5]. This analysis outlines some of the interconnect evolutions applied to bus architectures to keep them viable as long as possible. However, it also shows how limited and complex the exploitation of the bus performance can be.

Chapter 3 on page 47 deals with the NoC simulation infrastructure we put in place to study in detail, among other things, interconnect performance. This SystemC cycle-accurate virtual platform is called MPARM [79]. This environment is a flexible MPSoC simulator, allowing for a variable number and type of cores to be attached to the interconnection backbone. MPARM also allows for complex memory hierarchies, with a dramatic impact on the interconnection load. MPARM was also extended to integrate extensive facilities for traffic generation; in fact, one of the challenges of interconnect design is the availability of accurate testing environments, based on loads as realistic as possible. The traffic generator we developed fulfills this requirement, providing the infrastructure to extract application behaviour from a simulation in a different environment and then replicate it onto a NoC. The traffic generation is accurate to the transaction level, and models cache effects, latencies and even reactiveness to interrupts.

Chapter 4 on page 95 provides an overview of our NoC architectural implementation, called ×pipes, which was also integrated into MPARM. ×pipes is a versatile synthesizable library of NoC components, mainly switches, NIs and links. ×pipes can be configured in many respects, including topology, switch radix, amount of buffering, data width, and even

flow control and arbitration policies. ×pipes is a fully synchronous library (even though its NIs have a facility to support integer clock division towards the attached core); this choice is motivated by the fact that, in our experiments, validated at the layout level, we could achieve very high operating frequencies without major drawbacks, rendering asynchronous design styles mostly superfluous. ×pipes supports OPEN CORE PROTOCOL (OCP) [12] at the IP core interface, making it possible for the NoC to seamlessly accommodate a variety of IP cores. As an example of architectural design space exploration, we study how different flow control implementations impact the NoC performance and its resource requirements. Three flow control protocols are compared: a retransmission-based one, a credit-based one, and a specialized one, named *T-Error*, designed to tolerate timing errors. These protocols are studied also in presence of link pipelining, *i.e.* in presence of links that need more than one clock cycle for traversal. The alternatives turn out to be best in different environments: maximum fault tolerance, tight resource constraints and maximum performance.

The following two chapters focus on the design flow that we set up to instantiate, optimize, verify, implement and characterize NoCs. Chapter 5 on page 115 discusses the front-end, namely the tool called SunFloor [133]. SunFloor is a topology generation software. Given as an input a communication graph that models the requirements of the target application (endpoints and bandwidth of each traffic flow), a set of area and power models of the NoC implemented in the target technology, and a set of objectives and constraints, SunFloor instantiates the optimal NoC for the given application. The topology is deadlock-free by construction. During its analyses, SunFloor takes into account the chip floorplan, and for example instantiates pipeline stages along links which are too long for single-cycle traversal.

Chapter 6 on page 145 deals with the other half of the design flow, the back-end. It presents the work we did to physically implement NoCs, going through the major steps of synthesis, placement and routing. This part of the flow is crucial to understanding factors such as maximum operating frequency, area occupation and power consumption. We then built NoC area and power models for use by SunFloor. Two main sets of experiments were performed once the flow back-end was in place. The first was a cross-benchmarking effort against bus architectures; it proved that NoCs have tangible advantages, including performance and predictability. A second set of experiments involved NoC implementations in 90nm and 65nm technologies, and returned valuable insight on technology scaling, NoC scaling, and tooling constraints for deep submicron implemen-

tations.

Chapter 7 on page 207 builds upon a basic NoC architecture and extends it. The NoC is enriched with a mechanism for dynamic traffic rerouting. This mechanism allows for workload processing even upon system failures (fault tolerance). It can also be useful in other ways, including load balancing (better performance) and workload processing even in presence of some system components which are turned off (power saving).

In Chapter 8 on page 227, we explore one of the potential future fields of application for NoCs, namely, stacked chips (often called "3D" chips). These devices exhibit many properties which are an ideal match for what NoCs have to offer, including extreme complexity, modularity requirements, and, due to manufacturing limits, scarce availability of wires in the vertical axis. We show some of the foundations required for 3D NoCs, including studies on physical properties, tentative layouts, and solutions for inter-layer clocking issues.

Finally, Chapter 9 on page 255 draws conclusions from the author's doctoral work and proposes directions for future research.

CHAPTER 2

MPSoC Interconnect Evolution

This chapter[1] presents, by means of cycle-accurate simulations, performance studies on bus-based MPSoC interconnects. These analyses allow for a better understanding of on-chip traffic, of the performance of existing buses, and of the evolution (in terms of protocol and topology) that on-chip buses have witnessed.

2.1 Motivation and Key Challenges

As lithographic processes keep improving, the integration of large numbers of IP blocks onto the same silicon die is becoming technically feasible. The communication subsystem of these complex Systems-on-Chip (SoCs) is increasingly critical for system performance, and therefore represents a key component to be investigated during architecture definition and tuning.

Most current designs are based on shared communication resources (buses) due to their low cost. Unfortunately, scalability is limited by serialization for multiple bus access requests. To face evolving communication requirements in MPSoC designs, the industrial response has been to improve interconnect fabric capabilities by adopting new topologies or new protocols.

It is therefore mandatory to analyze the performance of MPSoC based

[1]The author would like to acknowledge contributions by Dr. Mirko Loghi, Prof. Davide Bertozzi, Dr. Francesco Poletti, Martino Ruggiero, Prof. Luca Benini and Dr. Roberto Zafalon.

on traditional and evolved bus interconnection schemes, so as to assess their efficiency and limitations. Crucial questions include latencies, saturation points, scalability and dependency on other system parameters, such as cache size.

We provide analyses to answer these questions, by comparing three alternative bus architectures (AMBA AHB, STBus and AMBA AXI) and several topological variants (from shared bus to full crossbar), under varying traffic conditions. Our study is based on cycle-accurate simulation of bus models, allowing us to pinpoint low-level protocol details responsible for macroscopic performance differences. Furthermore, we inject realistic, functional traffic derived from MPARM (Section 3.2 on page 49) applications instead of fixed execution traces, or statistic traffic generators, or analytical models. In this way, dynamic effects such as interaction among traffic sources can be taken into account. Experimental results demonstrate that subtle protocol mismatches and middleware-induced behavior are indeed responsible for macroscopic performance differences.

2.2 Bus Architectures

We focus our attention on some of the best-known, and most widely used, on-chip bus architectures:

- AMBA 2.0 AHB by ARM Ltd.
- STBus by STMicroelectronics
- AMBA 3.0 AXI by ARM Ltd.

While not exhaustive, we believe that this set of architectures is comprehensive enough to evaluate the current status and trends of on-chip interconnects.

2.2.1 AMBA 2.0 AHB

The ADVANCED MICROCONTROLLER BUS ARCHITECTURE (AMBA) 2.0 [11] interconnect is a well-established fabric architecture for SoC designs, thanks to its moderate silicon footprint. The AMBA 2.0 specification dictates three different architectures with varying levels of complexity and

performance; here, we will refer to ADVANCED HIGH-PERFORMANCE BUS (AHB), the fastest of them.

AMBA AHB leverages upon a straightforward shared bus topology. A single address and control channel is provided; two data links (one for reads, one for writes) are available, but only one of them can be active at any time. This is because the communication protocol is kept simple to minimize area overhead: a single transaction can be pending. An AHB bus joins several IP blocks (acting as AHB masters and slaves), and includes one central arbiter to manage interconnection resource access. A minimal amount of flip-flops is required in the architecture, which is typically bufferless.

The bus resources are owned by a single master at a time; if the targeted slave is slow and inserts wait states before responding, no other transaction can be initiated, neither by the current bus owner nor by any other master. As a result, the utilization of bus bandwidth might be poor. To work around the most serious instances of this issue, AHB provides two mechanisms. The first is called *split/retry* transfer: a high-latency slave can optionally decide to release the bus while preparing its response to a master-initiated transaction. However, this mechanism requires more complex slaves and arbiters. The second alternative is called *early burst termination*: if the arbiter detects that the bus has been busy for too long, it can interrupt a burst transfer in progress and assign the bus to another master with pending bus access request. Both methods are ineffective when the slave latency is of just a few cycles, because the overhead they impose would be worse than just waiting.

AMBA AHB exploits logical pipelining. Transfers are composed of an *address phase* (involving the address and control wires) and of a *data phase* (involving one of the two data buses); the address phase of a new transfer overlaps with the data phase of the previous transfer. This allows to increase throughput while imposing light timing requirements upon the slaves, but also increases latency.

AMBA AHB, however, does not resort to pipelining at the physical level. In other words, the paths for communication among all masters and all slaves are combinational. Therefore, a key performance assumption is that the propagation delay of the interconnect wires will be short. If that is the case, communication will incur the minimum possible latency. However, new technology nodes are leading to faster and faster logic, potentially resulting in faster clock periods, while wire propagation delays are proportionally increasing. If the whole fabric is constrained to slow operation by wire delay, this factor represents a limit to maximum operating frequency.

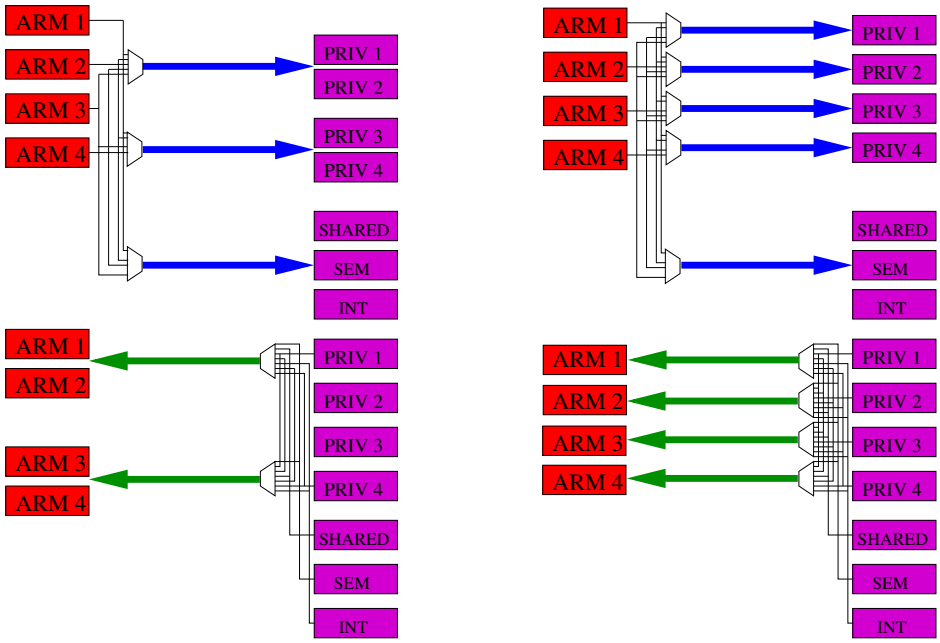


Figure 2.1: Partial STBus crossbar configurations: 3/2, 5/4 request/response channels.

AMBA AHB supports bursts, but it treats them as streams of single transactions; bursts are simply a way of arbitrating just once for multiple transfers, thereby reducing latency. Memories have no way of detecting bursts early, and accordingly make use of prefetching or buffering.

2.2.2 STBus

STBus [5] is a flexible communication architecture developed by STMicroelectronics. Its specifications define three different protocols; the simplest is called *type 1* and supports simple load/store operations, *type 2* adds more complex transfers, pipelining and split transactions, and finally *type 3* adds out-of-order support. Our tests are based on type 3 protocol.

The topology of an STBus interconnect is also very flexible and can range from a simple shared bus, like AMBA AHB, to a full crossbar. We analyze performance obtained from a variety of STBus topologies, from shared bus to full crossbar. Two possible examples of partial crossbars, that we use for our experiments, are presented in Figure 2.1.

STBus features two data communication channels, a request one from initiators (*e.g.*, processors) to targets (*e.g.*, memories and dedicated hardware) and the response one in the opposite direction. This allows an initia-

tor to send a request while a target is sending a response. This overlapping of transfers is a key performance enhancer. As soon as the response channel frees up, the second request can immediately be serviced, thus hiding target wait states behind those of the first transfer. The amount of saved wait states depends on the depth of the prefetch FIFO buffers on the slave side. Additionally, the split channel feature allows for multiple outstanding requests by masters, with support for out-of-order retirement.

STBus features fast arbitration, and this makes it possible to complete single read transfers in just two cycles, versus the three needed by AMBA - one cycle for arbitration/sending addresses and one for receiving data. When inserting a wait state, the minimum latency becomes of three cycles.

2.2.3 AMBA 3.0 AXI

AMBA ADVANCED EXTENSIBLE INTERFACE (AXI) [4] builds upon the concept of point-to-point connection. AMBA AXI does not provide masters and slaves with visibility of the underlying interconnect, instead featuring the concept of *master interfaces* and symmetric *slave interfaces*. This approach, besides allowing for seamless topology scaling, has the advantage of simplifying the handshake logic of attached devices, which only need to manage a point-to-point link. Complex features, like multiple outstanding transactions support (with out-of-order or in-order delivery selectable by means of transaction IDs) and time interleaving of traffic towards different masters on internal data lanes, can be transparently provided within the interconnect fabric.

To provide high scalability and parallelism, five different logical monodirectional channels are provided in AXI interfaces: a read address channel, a write address channel, a read channel, a write channel and a write response channel. Activity on different channels is mostly asynchronous (*e.g.* data for a write can be pushed to the write channel before or after the relevant address is issued to the write address channel), and can be parallelized, allowing for multiple outstanding read and write requests. However, the mapping of channels, as visible by the interfaces, to actual internal communication lanes is decided by the interconnect designer; single resources might be shared by all channels of a certain type in the system, or a variable amount of dedicated signals might be available, up to a full crossbar scheme. The rationale of this split-channel implementation is based upon the observation that usually the required bandwidth for addresses is much lower than that for data (*e.g.* a burst requires a single address but maybe four or eight data transfers). Availability of independently scalable resources might, for example, lead to medium complexity

designs sharing a single internal address channel while providing multiple data read and write channels. In our protocol exploration, to provide a fair comparison, we assume the "shared bus" topology to comprise a single internal lane per each one of the AXI channels.

2.3 Bus Performance Analysis

We test the performance of AMBA AHB, AMBA AXI and STBus within the framework of the MPARM simulation platform (Section 3.2 on page 49). This environment is composed of a configurable number of ARM cores attached to the system interconnect. Traffic workload and pattern can easily be tuned by running different benchmark code on the cores, by scaling the number of system processors, or by changing the amount of processor cache, which leads to different amounts of cache refills. The AMBA AHB and AMBA AXI modeling is based upon SystemC libraries provided within the Synopsys CoCentric/DesignWare [184] suites, while the STBus model is provided by STMicroelectronics.

We test the interconnects with four functional benchmarks running on MPARM (Section 3.2 on page 49). These benchmark perform matrix multiplications, either independently from each other or in pipeline, and with and without an underlying OS (**OS-IND**, **OS-PIP**, **ASM-IND** and **ASM-PIP** respectively). The benchmarks interact differently on the interconnect. **OS-IND** and **ASM-IND** do not contend for any memory device, since each processor operates on a private memory slave, but they have to contend for access to the single bus to which the memories are attached. **OS-PIP** and **ASM-PIP** are more sophisticated, as the processors act in a producer/consumer pipelined fashion, exchanging data through shared memory banks - which become an additional bottleneck. The final interesting difference is in the synchronization mechanism adopted by **OS-PIP** and **ASM-PIP**. The former leverages OS primitives for message passing, and therefore exchanges interrupts to synchronize producers and consumers. **ASM-PIP**, in need of a simpler implementation due to the lack of OS libraries, resorts to polling on a semaphore device; this behaviour establishes a further bottleneck, which is application-dependent more than hardware-dependent.

We measure several statistics:

- Bus usage. We define this as the ratio of data transfers over total

execution time, which expresses the amount of injected traffic and is a metric of congestion.

- Bus efficiency. We define this as the ratio of data transfers over the time during which the interconnect is busy with any transaction (including arbitration intervals, *etc.*).

- Read transaction latency.

We first (Section 2.3.1) explore the impact on performance scalability of the adoption of more advanced protocols, by comparing the three bus architectures all with a shared bus topology. We subsequently (Section 2.3.2 on page 42) introduce the topology variable into the picture, by presenting a study on various STBus topologies (from shared bus to crossbar, including two partial crossbars). In both studies, we present AMBA AHB as a baseline, since it is designed for lower area occupation and thus lower performance.

2.3.1 Impact of Protocols on Interconnect Scalability

In this section, to further show the scope of optimizations at the architectural level even when sticking to plain shared buses, we test the STBus model in two configurations, namely by varying the depth of the FIFOs instantiated at the target side of the interconnect. We benchmark with 1-stage ("STBus") and 4-stage ("STBus (B)") FIFOs.

Figure 2.2 on the following page shows an example of the efficiency improvements made possible by advanced interconnects in the test case of slave devices having two wait states, with three system processors and 4-beat burst transfers. AMBA AHB has to pay two cycles of penalty per transferred piece of data. STBus is able to hide latencies for subsequent transfers behind those of the first one, with an effectiveness which is a function of the available buffering. AMBA AXI is capable of interleaving transfers, by sharing data channel ownership in time. Under conditions of peak load, when transactions always overlap, AMBA AHB is limited to a 33% efficiency (transferred words over elapsed clock cycles), while both STBus and AMBA AXI can theoretically reach a 100% throughput.

To assess interconnect scalability, we choose to run **ASM-IND** on every system processor. This means that, while producing real functional traffic patterns, the test setup is not constrained by bottlenecks due to shared slave devices.

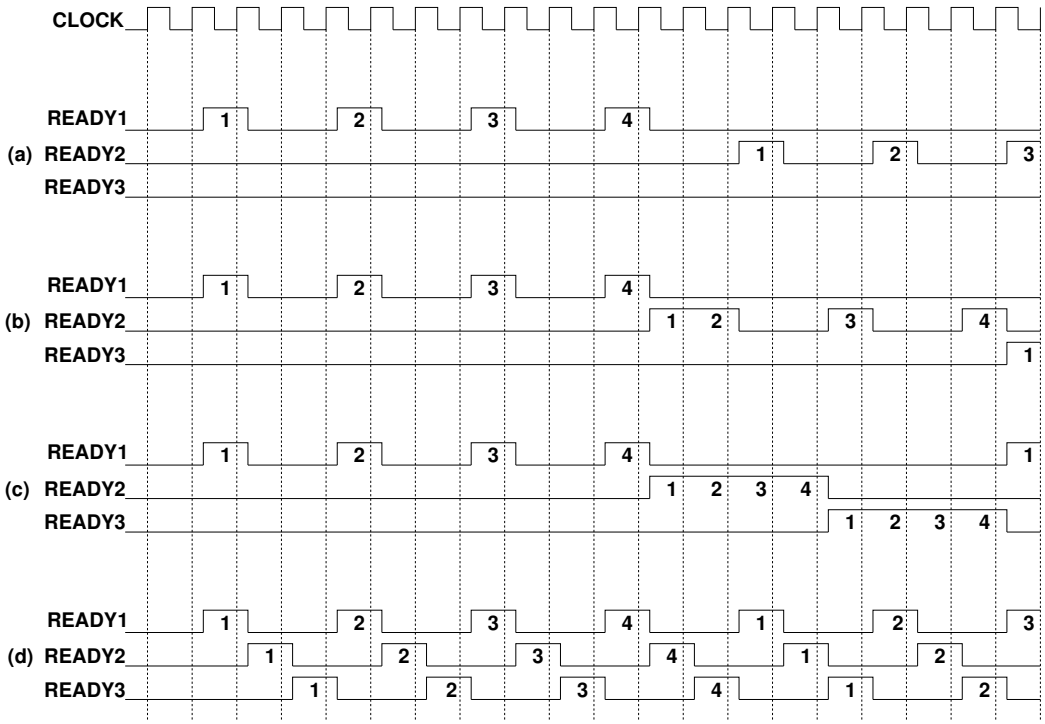
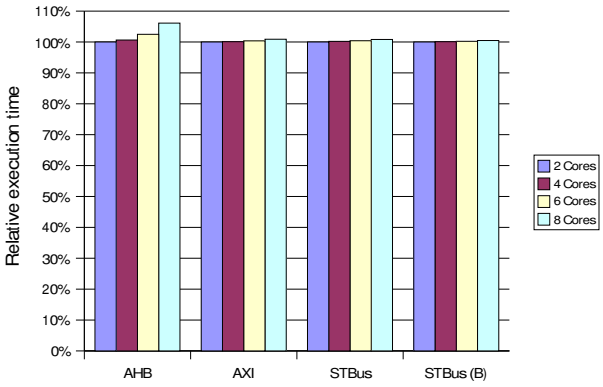
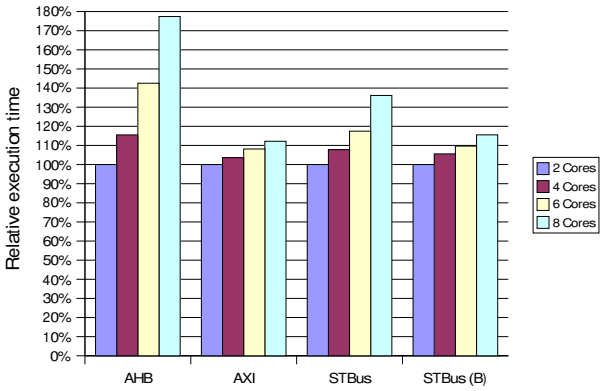


Figure 2.2: Concept waveforms showing burst interleaving for the three interconnects. (a) AMBA AHB; (b) STBus (with minimal buffering); (c) STBus (with more buffering); (d) AMBA AXI.



(a) 1 kB caches.

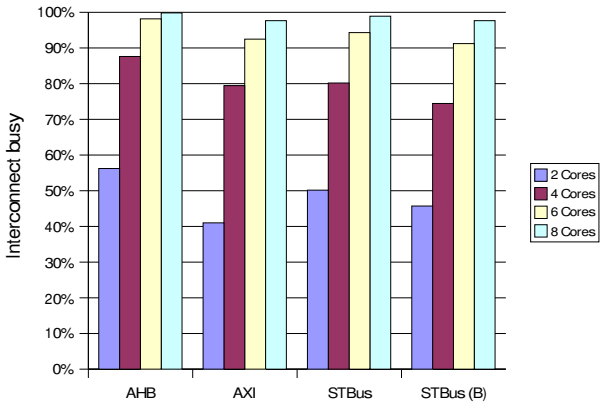


(b) 256 B caches.

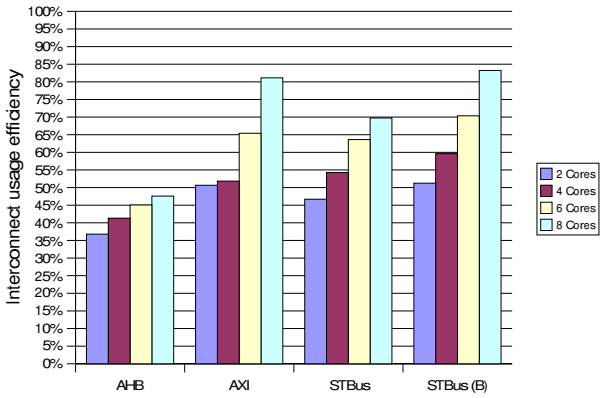
Figure 2.3: Benchmark execution times, varying cache sizes.

Scalability results are shown in Figure 2.3 on the previous page in terms of execution time variation when attaching an increasing amount of system cores to a single shared interconnect. Figure 2.3(a) on the preceding page reports figures for a system with 1 kB caches, while in Figure 2.3(b) on the previous page caches are reduced to 256 bytes, thus causing many more cache misses and greater interconnect congestion. Execution times are normalized against those for a two-processor system, trying to isolate the scalability factor alone. Simulations show that, as long as traffic is relatively light (1 kB caches), all of the interconnects perform very well, with only AHB showing a moderate performance decrease of 6% moving from two to eight running processors. With 256 B caches and many processors, interconnect saturation takes place, as can be seen from Figure 2.4(a) on the facing page, which reports the bus usage time. In such a congested environment, as can be seen in Figure 2.4(b) on the next page, AMBA AXI and STBus (with 4-stage FIFOs) are able to achieve transfer efficiencies of up to 81% and 83% respectively, while AMBA AHB reaches 47% only - near to its maximum theoretical efficiency of 50% (one wait state per data word). The resulting execution times, as Figure 2.3(b) on the preceding page shows, got 77% worse for AMBA AHB when moving from two to eight cores, while AXI and STBus manage to stay within 12% and 15%. The impact of FIFOs in STBus was noticeable, since the interconnect with minimal buffering showed execution times 36% worse than in the two-core setup. This stresses the impact that comparatively low-area-overhead optimizations can sometimes have in complex systems.

According to simulation results, some of the advanced features in AMBA AXI provide highly scalable bandwidth, but at the price of latency in low-contention setups. Figure 2.5 on page 42 shows the minimum and average amount of cycles required to complete a single write and a burst read transaction in STBus and AMBA AXI. STBus has a minimal overhead for transaction initiation, as low as a single cycle if communication resources are free. This is confirmed by figures showing a best-case three-cycle latency for single accesses (initiation, wait state, data transfer) and a nine-cycle latency for 4-beat bursts. AMBA AXI, due to its complex channel management and arbitration, requires more time to initiate and close a transaction: the minimum recorded completion times were six and eleven cycles for single writes and burst reads respectively. As bus traffic increases, completion latencies of AMBA AXI and STBus get more and more similar because the bulk of transaction latency is spent in contention.



(a) Bus usage.



(b) Bus efficiency.

Figure 2.4: Bus performance metrics, 256B caches.

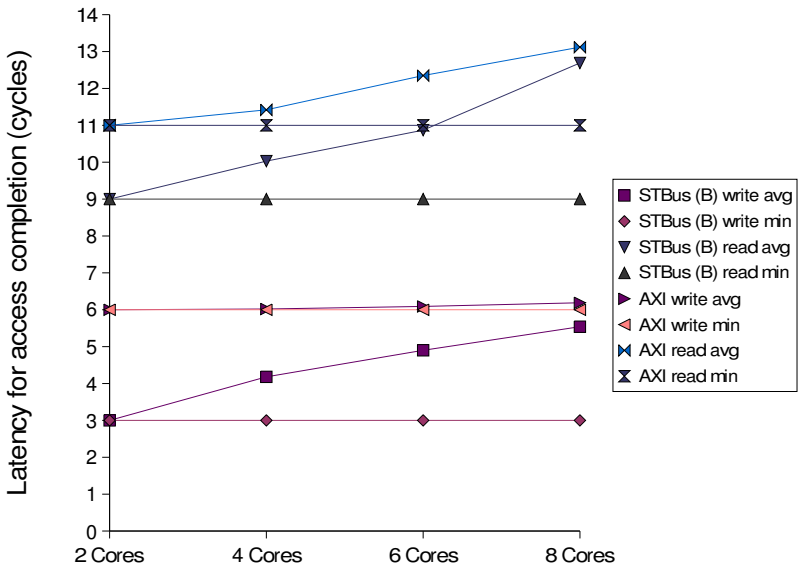


Figure 2.5: Transaction completion latency with 256B caches.

2.3.2 Topology Effect on Interconnect Performance

We proceed to comparing the performance of STBus interconnects arranged in different topologies: shared bus (ST-BUS), full crossbar (ST-FC), two partial crossbars (ST-32 and ST-54, see Figure 2.1 on page 34). We include AMBA AHB as a baseline.

The results are presented in Figure 2.6 on the next page. First, bus congestion results (Figure 2.6(a) on the facing page) show that three benchmarks put relatively light pressure on the system interconnect (around 10%), while **OS-PIP** is much more demanding, due to its larger memory footprint and worse memory locality, which increase the amount of bursts for cache refills. There is no significant difference in congestion metrics among the interconnects, since this value is mostly benchmark-dependent.

Bus efficiency (Figure 2.6(b) on the next page), in contrast, is clearly higher for STBus. Since transfers are composed of one wait state followed by a single piece of data, efficiency could in principle be estimated to be 50%; STBus however is always above that threshold, because, even in its shared bus topology, it has the ability to hide some wait states via its dual request/response channels (Figure 2.2 on page 38). AMBA efficiency instead is always below 50%; this is because of the arbitration overhead. ST-FC, ST-32 and ST-54 are able to boost dramatically bus efficiency, since they allow more transfers in parallel. Since accesses to shared devices (shared

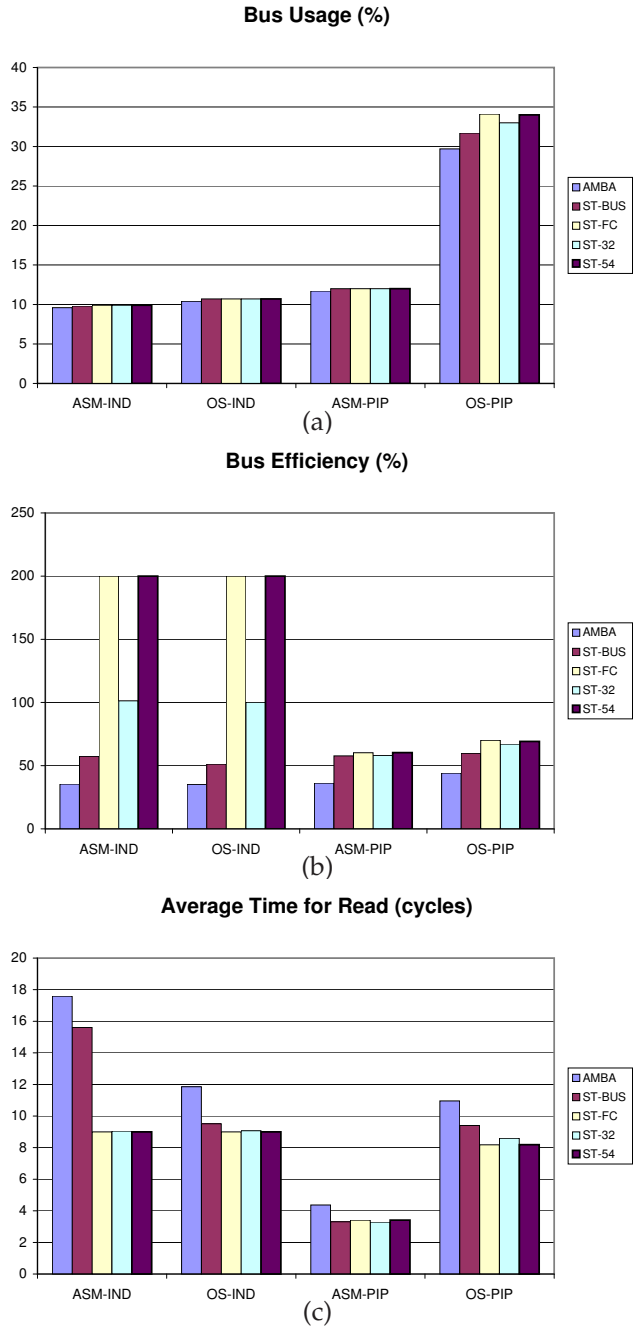


Figure 2.6: Analysis of bus performance metrics for five different bus interconnects. 8 kB caches, slaves with 1 wait state.

memory, sempahores, interrupt module) are serialized anyway, the advantage in **ASM-PIP** and **OS-PIP** is still relatively small; but in **OS-IND** and **ASM-IND**, where all accesses are to private memories, crossbars significantly outperform other schemes. ST-32 achieves 100% efficiency (two memories can be accessed at a time), while ST-54 and ST-FC hit 200% (four memories at a time). It is evident that crossbars behave best when data access is local and no destination conflicts arise.

Finally, Figure 2.6(c) on the previous page shows average completion latencies for read accesses. STBus is faster and exhibits lower latencies. ST-BUS has an edge of one to about two cycles over AMBA, mostly due to arbitration (which ST-BUS always performs one cycle faster), and in part also to the better efficiency seen above. Once more, crossbars show a substantial performance advantage, the only exception being **ASM-PIP**, where ST-BUS performs similarly. This can be explained with the continuous semaphore polling performed by this (and only this) benchmark; while crossbars may have an advantage in private memory accesses, the resulting speedup only gives processors more opportunities to poll the semaphore device, which becomes a bottleneck. Not plotted, we also record worst-case completion latencies of up to 34 cycles for AMBA, while the STBus topologies fare substantially better, especially the crossbars.

We now study the impact of varying cache sizes (and therefore, cache miss traffic) and slave wait states on the interconnects. Figure 2.7 on the facing page shows the total execution time of the **OS-PIP** benchmark, in scenarios having different cache and memory latency settings. Three interesting comparisons can be made by looking at this graph. The first is again an analysis of interconnection performance, this time as a function of different environments. As expected, STBus always exhibits an advantage, in that it cuts execution times from 9% to 35% with respect to AMBA (from 18% to 58% with ST-FC). ST-54 (not graphed) performs almost identically to ST-FC, while ST-32 (not graphed) once again trails behind other crossbars, but is still faster than both buses. When comparing more efficient interconnections to less efficient ones, gains are lowest when the traffic is lightest, *i.e.* with big caches and fast memories, but progressively increase with interconnection congestion.

The second analysis regards performance improvement due to cache size. A 4 kB cache can bring 4% to 26% speed-ups in execution time with respect to a 1 kB cache; with 8 kB, speed-ups range from 20% to 48%. The widest gaps, as expected, can be noticed with relatively slow interconnections and high latency memories.

A third assessment that can be made is about memory latency impact on execution times. Increasing memory wait states from 1 to 4 slows down

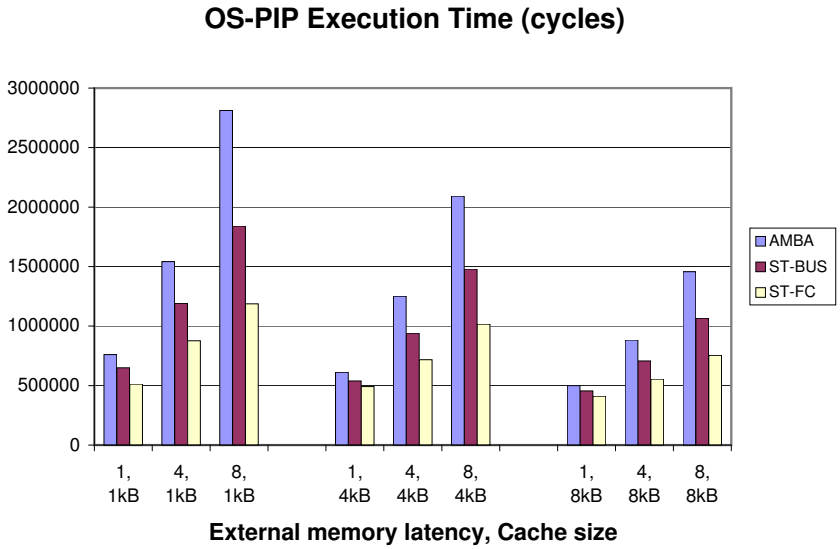


Figure 2.7: Performance of the interconnects as a function of cache size and slave wait states.

execution times from 35% to 104%, and 8 wait states even from 84% to 370%. This once more stresses the importance of fast memories, even though big caches and fast interconnections help somehow.

2.4 Conclusions

We have shown comparative performance evaluations of different bus-based interconnects for MPSoCs. The chosen case studies reflect an evolution which has occurred in industrial practice, namely the shift from plain shared buses to either (i) buses with more advanced protocol features, and/or (ii) topologically-enhanced bus interconnects, featuring components such as crossbars.

Our results, pertaining purely to architectural-level performance estimation, prove that both choices guarantee some performance headroom to plain shared buses - of course, at a silicon cost, either due to more complex protocol handling, to more buffering, or simply to the deployment of additional communication links. Both AMBA AXI and STBus prove able to more efficiently utilize the available bandwidth.

It must be pointed out, however, that protocol improvements alone cannot overcome the intrinsic performance bound due to the shared na-

ture of the interconnect resources. While protocol features can push the saturation boundary further, and get near to a 100% efficiency, traffic loads taking advantage of more parallel topologies will always exist, and our experiments already show some traces of saturation, even for the most advanced interconnects, just for 8 attached cores - a number which is expected to be reached and surpassed in current and future high-perfomance MPSoCs.

On the other hand, while crossbars certainly provide a performance boost, they do not seem to represent a durable solution either, because of expected layout issues during the physical design of large such components.

We read these results as a motivation for the development of more advanced interconnect solutions, such as NoCs.

CHAPTER 3

Simulation and Traffic Generation

This chapter[1] discusses MPARM, a SystemC simulation platform that was developed to evaluate the performance of MPSoCs with cycle accuracy. MPARM grew over the years to encompass a number of different platform variables, such as memory hierarchies, interconnects, IP core architectures, OSes, middleware libraries, *etc.*, making it possible to study the macroscopic impact of small changes at the architectural or programming level. MPARM also became an ideal platform for our interconnect performance simulations.

3.1 Motivation and Key Challenges

The increasing complexity of current-generation MPSoCs is making it increasingly hard to (i) evaluate, (ii) debug, (iii) optimize, (iv) verify their functionality. This has led to a wide range of approaches, spanning from analytical models to cycle-accurate simulators, in pursuit of the ideal platform in which to characterize and optimize MPSoC behaviour.

A large number of system variables are involved in such a characterization, making the task a very challenging one. For example, aspects needing consideration include:

- A large variety of IP cores, ranging from microprocessors to DIGITAL SIGNAL PROCESSORS (DSPs), from accelerators to VERY LONG

[1]The author would like to acknowledge contributions by Dr. Mirko Loghi, Prof. Davide Bertozzi, Dr. Francesco Poletti, Jianjiang Ceng, Federico Ferrari, Cesare Ferri, Dr. Shankar Mahadevan and Prof. Luca Benini.

INSTRUCTION WORD (VLIW) blocks. All of these behave differently and have different input/output requirements.

- Extremely varied memory hierarchies. Many possible alternatives have been picked in the MPSoC space, including caches, scratchpad memories, on-chip and off-chip STATIC RANDOM ACCESS MEMORY (SRAM) and DYNAMIC RANDOM ACCESS MEMORY (DRAM) banks. Any of these results in different performance, area and power figures.

- A wide range of system interconnects, including shared buses of several types, bridged and clustered buses, partial and full crossbars, up to NoCs.

- An almost unlimited choice of software, from complete stacks including OS and middleware to programs in Assembler to most efficiently exploit the underlying architecture.

- A rich set of alternative communication and synchronization schemes, including shared memories, message passing, DIRECT MEMORY ACCESS (DMA) transfers, interrupts, semaphore polling, *etc.*.

It is our opinion, supported by experience, that while high-level models can provide a rough (and still valuable) approximation of the interactions in such a huge design space, it is only when low-level details are taken into account that a clear picture emerges. At times, tiny details such as the size of one particular buffer, or the choice of a different compiler flag, can have a dramatic impact on the performance of the resulting system. It is for this reason that, among all the possible MPSoC characterization approaches, we developed MPARM, a cycle-accurate SystemC simulator that enables sweeping alternatives for all the items listed above. We believe that the availability of such a platform is crucial in the context of the main goal of this dissertation, *i.e.* the assessment of the tradeoffs involved in MPSoC interconnect design.

Of course, the challenges of cycle-accurate, detailed simulation are not to be underestimated, either. Two of the main issues are:

- Achieving sufficient simulation speeds.

- Developing simulation models within a reasonable time.

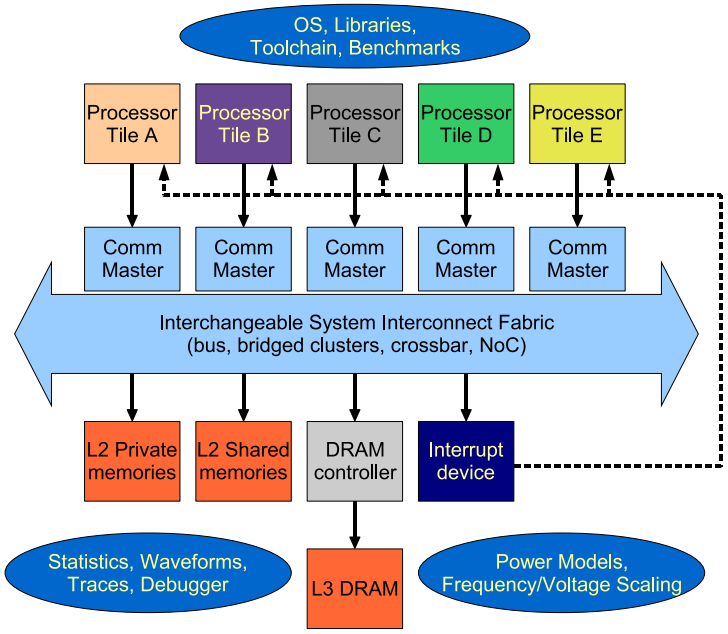


Figure 3.1: The MPARM SystemC virtual platform.

In this chapter, after a brief introduction to the facilities provided by MPARM, we will focus on exactly these issues. We will first outline how we integrated LISATek [91] cores into the platform, making it possible to directly plug several pre-designed IP core models into MPARM, and also making it possible to develop new such models in a fraction of the time it would normally take. We will then outline how we integrated a novel traffic generation scheme, which, while speeding up design and simulation time, still strives to remain remarkably faithful to traffic patterns as generated by real applications running on real architectures.

3.2 SystemC Platform Simulation

The MPARM [77, 78] environment is designed to investigate the system-level architecture of MPSoC platforms. To be able to fully assess system performance, a cycle-accurate, signal-accurate modeling infrastructure is put into place.

MPARM is a plug-and-play platform based upon the SystemC [185] simulation engine, where multiple IP cores and interconnects can be freely

mixed and composed. At its core, MPARM is a collection of component models, comprising processors, interconnects, memories and dedicated devices like DMA engines. The user can deploy different system configuration parameters by means of command line switches, which allows for easy scripting of sets of simulation runs. A thorough set of statistics, traces and waveforms can be collected to analyze performance bottlenecks and to debug functional issues. To take into account other crucial design variables, power models for many of the MPARM components are supplied. Frequency and voltage scaling can be realized at runtime thanks to dedicated programmable registers.

MPARM features a choice of several IP cores to be used as system masters. Some of these are taken from the open source or academic domain, and while spanning over a range of architectures, they typically model pre-existing industrial general purpose processors with little to no possibility of modifying the supported instruction set and architecture. Section 3.3 on the next page and Section 3.4 on page 63 discuss additional core models for MPARM, extending its simulation capabilities.

MPARM provides extensive facilities to study the performance of alternative memory hierarchies. Three layers of memory devices are defined: (1) on-tile, strongly coupled to the processor, *e.g.* caches and SCRATCH-PAD MEMORIES (SPMS); (2) on-chip, attached to the system interconnect; (3) off-chip, driven by a DRAM memory controller. In addition, to analyze inter-processor communication behaviour, memories can be defined as private or shared, and a cache snooping mechanism is provided. The latency of each memory can be freely defined.

In terms of interconnect, MPARM provides a wide choice, spanning across multiple topologies (shared buses, bridged configurations, partial or full crossbars, NoCs) and both industry-level fabrics (AMBA AHB and AXI [186], STBus [5]) and academic research architectures such as the ×pipes NoC described in this dissertation.

On top of the hardware platform, MPARM provides a port of the uClinux [187] and RTEMS [188] operating systems. The choice of RTEMS is motivated by the fact that RTEMS is a lightweight OS for embedded systems, but it offers at the same time good support for multiprocessing, and provides native calls for communication and synchronization in such multiprocessor environments.

Application code, either OS-based or not, can be easily compiled with standard GNU cross-compilers. Scripts and makefiles fully automate the process of building for a multiprocessor platform. Simple function calls, provided by support libraries of the simulator, allow flexible performance profiling: statistics can be collected during OS boot, application execution,

or critical sections of algorithms.

MPARM also features support libraries to help fast development and debugging of new applications and benchmarks. This is key for establishing a solid and flexible simulation environment. MPARM includes several benchmarks from domains such as telecommunications and multimedia, and libraries for synchronization and message passing.

Debug functions include a built-in debugger, which allows to set breakpoints, execute code step-by-step and inspect memory content; it is additionally capable of dumping the full internal status of the execution cores. When testing applications written without underlying OS support (*i.e.*, no native I/O calls are available), messages and status information can still be easily provided to the user by means of pseudo-instructions.

Multiple communication and synchronization paradigms are possible in MPARM, including plain data sharing on a shared memory bank, message passing among SPM resources of each processor, interrupts and semaphore polling.

Simulation accuracy and flexibility have to be traded off with simulation speed. However, MPARM, despite being signal-accurate and cycle-accurate, is fast and usable. Simulation performance is in the range of 200 kCPUcycles/s, which is enough to simulate applications of reasonable complexity in few minutes.

In the context of this dissertation, the MPARM features are key to interconnect performance evaluation. MPARM stimulates the communication subsystem with functional traffic generated by real applications running on top of real processors. This opens up the possibility for communication infrastructure exploration under real workloads and for the investigation of its impact on system performance at the highest level of accuracy.

3.3 Integration of Advanced Platform Cores

To face increasing architectural design complexity in deep-submicron technology nodes, the reuse-centric paradigm based on IP cores is a natural solution. However, this approach still poses significant challenges. First of all, general-purpose IP blocks lend themselves very well to quick parallel deployment in MPSoCs, but often do not provide enough performance when running complex user applications, such as multimedia streaming or floating point computation. In fact, depending on the application, dedicated IP blocks could deliver much higher efficiency thanks

to task-optimized circuitry. This observation leads to APPLICATION SPECIFIC INSTRUCTION SET PROCESSORS (ASIPS), *i.e.* to IP cores stemming from the architecture of general-purpose processors but with an instruction set comprising at least some custom instructions optimized to accelerate the task at hand. If designed with state-of-the-art CAD toolchains, ASIPs can provide most of the advantages of dedicated IPs but reduce development time by several times [189] and maintain flexibility. The commercial LISATek suite [91] was born to focus on the seamless development of ASIPs, by leveraging a dedicated modeling language (LISA) and by providing numerous development and debugging facilities.

It is immediately apparent that there is value in the integration of LISATek technology within the MPARM framework, due to several reasons:

- Instant enrichment of the MPARM portfolio of IP cores with several additional LISATek models of existing IP cores.
- Ability to develop new IP cores for MPARM leveraging an easy-to-use modeling language specifically conceived to describe processor architectures - which allows for dramatic savings in coding time.
- Ability to modify and optimize IP core models at any time, adding to the degrees of freedom already supported by MPARM.

Further, LISATek adopts a SystemC simulation backbone, which enables a clean integration with MPARM.

The joint MPARM/LISATek framework that we developed enjoys significant unique benefits compared to existing virtual platforms. Usually, two major families of tools can be easily recognized in the virtual platform space: academic and industrial. They differ in many respects, the main difference being conceptual and related to the different purpose they serve. Research tools are usually open in nature, and experimentation is encouraged and welcome; but not easy. Documentation is minimal, user interfaces are hard to use, and the do-it-yourself approach to problem solving is dominant. In stark contrast, industrial tools support a variety of useful development and verification features, but the typical expected design flow calls for deploying pre-designed and pre-verified blocks, configuring some parameters, maybe adding one or two custom blocks, and testing. This kind of flow is efficient, but does not encourage research and exploration: IP blocks are shipped in encrypted form and their internal architecture cannot be explored or extended. For the first time, by integrating LISATek within MPARM, the advantages of both approaches can be made

simultaneously available. We aim at the sweet spot between the industrial and academic approaches: the LISATek roots guarantee industrial-grade development and debugging facilities, while all of the platform code (LISA processing blocks and SystemC interconnects and memories) can be modified at any time for research purposes. Open-source software support is also provided for the required hardware abstraction layers. The result is an open platform where the architecture of each hardware module can be changed, and which is easily extensible by adding new models.

3.3.1 The LISATek Design Platform

The LISATek processor design platform is built around the LISA 2.0 ADL [190]. Figure 3.2 on the next page shows the processor design flow supported by LISATek. From a processor model written with the LISA 2.0 ADL, a set of processor development tools such as instruction-set simulator, C-compiler, assembler, and linker are automatically generated to support architecture exploration. A graphical user front-end is also available for software debugging and profiling purposes. Moreover, RTL hardware models in the most popular hardware description languages, VHDL, SystemC and Verilog, can also be generated from the LISA model for hardware implementation. With the LISA platform, the ASIP development time can be greatly reduced compared to the traditional manual approach. Design efficiency is achieved through high degree of automation.

3.3.2 The LISATek Simulation Interface

Given a LISA model, the LISATek tool is able to generate instruction-set simulators for the processor under design. Typically, the generated simulator is directly used by the debugger in form of a dynamic library. However, a compiled static simulator library is also generated, and specifications exist to integrate it into a system environment. In our case, the system environment would be MPARM. All the core models generated by the LISATek suite, regardless of the nature of the ASIP at hand, have the same interface. The interaction is based upon four key pillars:

- The simulated core can be cycled by calling specific functions. If the processor is modelled in an instruction-accurate fashion, then the generated model can be stepped on an instruction basis. On the other hand, a model derived from a cycle-accurate LISA description can be stepped on both instruction and cycle basis.

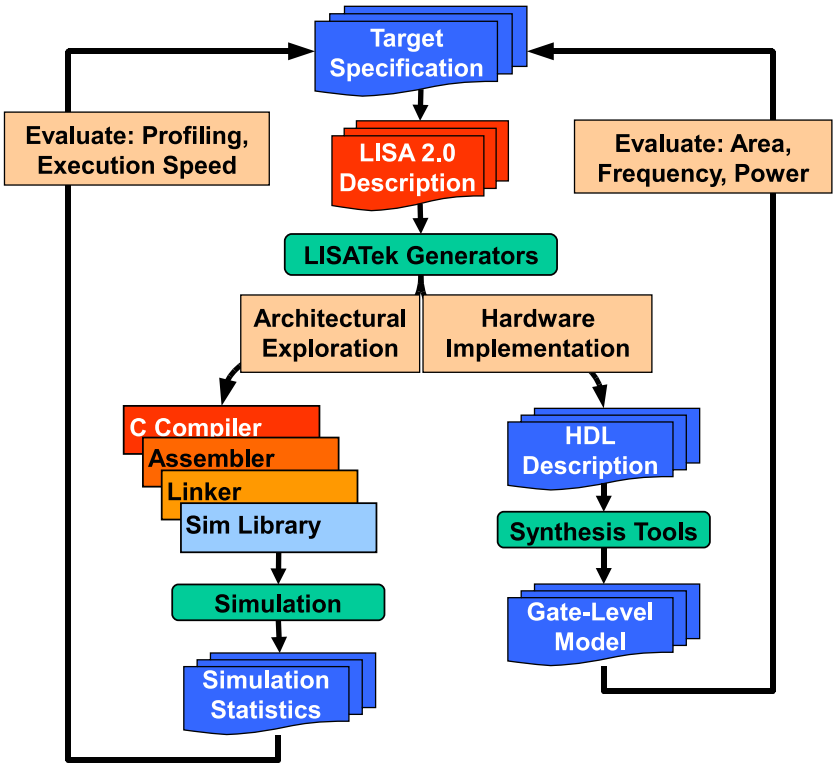


Figure 3.2: LISATek-based ASIP design flow.

- Core-initiated communication (*e.g.* reads, writes) is performed through a specific APPLICATION PROGRAMMING INTERFACE (API), which is discussed below. It is the task of the external program to provide an implementation of said API.
- System-initiated communication (*e.g.* interrupts), if any, can be forwarded to the core when cycling it, and therefore on a fine-grain cycle-by-cycle basis, by proper flipping of extra pins. Of course the LISA core model must be made aware of the meaning of these extra pins to take proper action.
- An external LISATek Debugger tool can be interfaced to the core via the INTER-PROCESS COMMUNICATION (IPC) mechanism. The external program must simply invoke the Debugger with proper references; subsequently, the LISATek model and the Debugger interact autonomously.

While all of these items were implemented during our work, the most interesting for discussion here is the API for core-initiated communication [191]. In a system environment, this LISATek API is the communication interface between the core and the external resources. It must be implemented by the external platform and passed to the processor simulator during system initialization. In addition to some control functions, the API is mainly composed of eight data-related calls (Listing 3.1).

Listing 3.1: LISATek communication API prototypes.

```
int read(AType addr, DType *data, int n, ...);
int write(AType addr, DType *data, int n, ...);
int request_read(AType addr, DType *data, int n, ...);
int request_write(AType addr, DType *data, int n, ...);
int try_read(AType addr, DType *data, int n, ...);
int could_write(AType addr, DType *data, int n, ...);
int dbg_read(AType addr, DType *data, int n, ...);
int dbg_write(AType addr, DType *data, int n, ...);
```

Three sets of calls, each of which constituting a sub-interface, can be distinguished. The first two calls represent the *blocking* sub-interface: they are based on the assumption that a non-cycle-accurate LISA core may be attached to a cycle-accurate external module. In this case, communication requests which can not be serviced immediately should yield control to the simulator, freezing the caller for as many cycles as needed to complete the transaction. As a result, no concurrent activity can be performed in the LISATek core if a transaction is pending.

The calls from the third to the sixth implement the *non-blocking* sub-interface; it is vital when designing cycle-accurate cores. The `request_read()` or `request_write()` functions are initially invoked; control is always returned. Subsequently, `try_read()` or `could_write()` can be invoked at each clock edge to try to carry the pending transaction on. The return status can be a negative acknowledge (*e.g.* if wait states are needed), but since control is always returned, the core is free to perform other tasks in background, such as shifting its pipeline.

The last two calls of the API are the *debug* sub-interface. Their purpose is to provide an instant reaction, bypassing any wait states. While of course this is not a realistic assumption for a physical system, the calls are extremely useful for debug purposes, such as monitoring or manipulating the content of an external memory while executing a benchmark. They are also useful to load the contents of a memory during the reset cycle.

The implementation of these function calls depends completely on the communication method used in the system; *e.g.* if the simulator needs to work with a system modelled at the RTL level, then the API must be implemented to translate the resource requests into RTL signals. In our case, the implemented API will translate the requests into SystemC signals which can be understood by the MPARM platform. Since MPARM is a cycle- and signal-accurate platform, implementing the first two sub-interfaces was straightforward. The third was supported by directly interfacing with the data arrays which hold the contents of simulated memories. In case caches were present, the implementation was tuned so to take them into account (*e.g.*, writing data to both the cache and the external memory when using write-through policies, and just to the cache when using write-back; or maybe only to the external memory in case of a write miss).

3.3.3 L1 Memory Placement Strategies

The LISA language makes no assumptions about how to model memory hierarchies. The language allows the specification of cache subsystems, but also permits the implementation of a flat memory array to which all accesses should be directly made. A typical LISA ASIP model is likely to take the second route, for at least two reasons: (i) implementing a complex cache controller is time-consuming, (ii) it is not very meaningful to accurately model a cache if there is no accurate model of the delays associated to an external memory.

The LISATek API mentioned in Section 3.3.2 on page 53 is transparent to the presence of caches. In fact, the API can be the *outer* interface of a

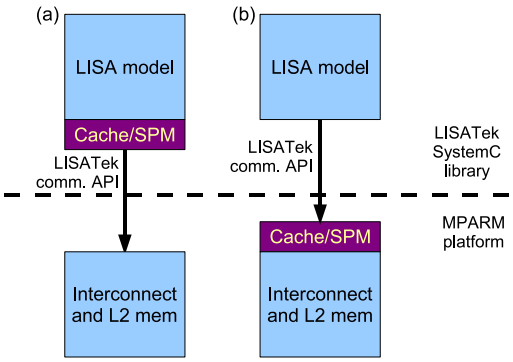


Figure 3.3: Possible placements of the L1 memory: (a) tightly coupled with the IP core, (b) as a system component.

cache layer, to handle refills and writebacks, as well as the *inner* interface, used by the processor to query a cache controller. Figure 3.3 illustrates the alternatives.

When integrating the LISATek processor models within MPARM, a choice had to be made regarding the most suitable L1 memory placement strategy. The alternatives were to develop the L1 memories together with each processor, therefore using the LISATek communication API among caches and MPARM; or to develop the L1 memories as an MPARM block, and using the API interface to drive them. Both paradigms allow for cycle-accurate modeling. Tightly coupling the L1 memory to the IP core has the advantage of allowing for arbitrarily complex interactions among the two components. Instead, an external module has the obvious advantage of reuse, where a single cache controller can be seamlessly used by any IP core.

After careful consideration, we went for the second alternative. While the LISATek communication API seems to be flexible enough to support all of the relevant core/cache interactions, thus making it less useful to develop caches inside of each core, we found that the reuse capability, given an equal development time, allows the shared cache module to support more features (different associativity levels, write-back *vs.* write-through policies, snooping capabilities, power optimizations and models), thus becoming more suitable for performance assessment.

While this subsection mostly mentioned cache memories, it is worth stressing that we also made it possible to instantiate an SPM next to (or in place of) them. Since a large body of research exists on how to exploit SPMs to improve embedded system efficiency [192], this adds a further useful degree of freedom for architectural exploration.

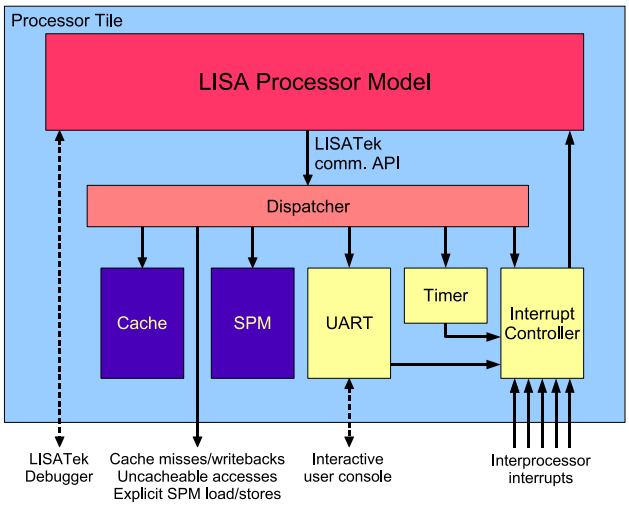


Figure 3.4: The scheme of a *processor tile*.

3.3.4 Core-Associated Devices

When developing a shared MPARM block to handle the L1 memory, we also found it useful to cluster other functionality at the same layer. The end result is a *processor tile*, comprising IP cores and the most tightly coupled components (Figure 3.4). Namely, we developed (i) a timer device, (ii) an emulated serial port, (iii) a simple interrupt controller. The first component is vital if attempting to port an operating system. The second is very useful for debugging purposes; placing it next to IP cores, instead of in a shared location accessible to all system processors, has the advantage of allowing for independent input/output, and prevents debug traffic from spilling onto the system interconnect where it could pollute performance statistics. Finally, the interrupt controller is both a requirement of the other two devices and a crucial component to develop efficient synchronization mechanisms in multiprocessor systems. The controller is externally attached to a set of system-level wires which convey inter-core interrupts. On the IP core side, we implemented a simple interrupt handshaking protocol where the value of interrupt registers is copied on some LISATek core pins which are polled every cycle by the core to take proper action. The interrupt controller is memory mapped, to let the core reset the pending interrupt flags and configure the masking status.

3.3.5 The Resulting System Architecture

The MPARM architecture is layered, to flexibly accommodate for different master devices and interconnect models. As can be seen in Figure 3.1 on page 49, the IP core tiles talk to a master device, which is in charge of handling any arbitration and/or routing phases required by the specific underlying fabric. The tile/master interface can either be MPARM-custom or comply with the OCP 2.0 [12] specifications.

The MPARM facilities allow the designer to flexibly instantiate complex platforms. Homogeneous as well as mixed processing tiles can be deployed, and selection among them is as simple as flipping a command line switch. Specific MPARM modules exist to handle addressing maps and to track simulation statistics, including cache hit rates, interconnect congestion and latencies, memory access patterns and (for the components for which a model is available) power consumption. In addition, the graphical LISATek Debugger can be launched to interactively inspect the status of each LISATek core, to set breakpoints and watchdogs, and to manually control the flow of execution.

3.3.6 Experiments and Case Studies

In this section, we demonstrate that we implemented a fully working and usable solution, and we show a sample of the kind of analysis that can be performed on our combined platform.

In order to achieve the former objective, we implemented a LISA ARMv7 core, which is instruction-equivalent to another ARMv7 core that was already available in MPARM. Since the cycle accuracy of the core itself was not important for our purposes, we kept its model very simple (no pipelining) without any timing accuracy effort. The expected result was complete functionality of the system platform. This achievement is testified by the screenshot in Figure 3.5 on the next page: the LISATek Debugger, in the foreground, is attached to a LISA core (currently paused on a breakpoint) that runs within MPARM. The console of the latter can be seen in the background. As a secondary result, by exercising the two ARMv7 implementations with the very same benchmark binary, we expected to find a perfect equivalence in the amount of memory accesses. Across several microbenchmarks and functional benchmarks from the multimedia and data encryption domains, including applications which leverage the RTEMS operating system, this result was indeed confirmed. On the other hand, we noticed a discrepancy of about 30% in the amount of execution cycles - which is perfectly normal due to the fact that the LISA ARMv7

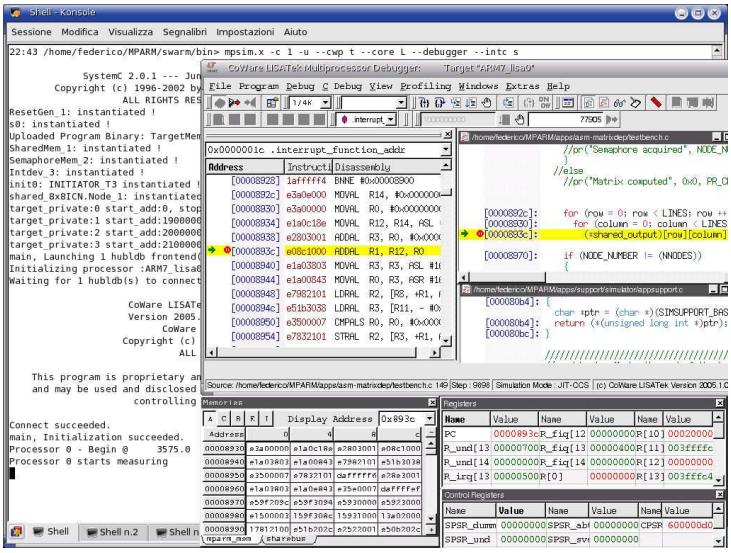


Figure 3.5: A system simulation screenshot.

model did not include a pipeline, while the MPARM one did. With LISA cores, we recorded up to 200 kCPUcycles/s on an Athlon XP 2200+ machine with 512 MB of RAM.

Next, we prove the importance of being able to model the effect of memory hierarchies and interconnect congestion on system performance. The choice of an instruction-accurate ARM model does not prevent cycle-accurate exploration of the impact of the communication fabric. Figure 3.6 on the facing page shows the execution time when a variable amount of IP cores, each of them independently performing the same benchmark, is attached to the interconnect. Each core executes an additional chunk of processing, therefore an increasing requirement of communication bandwidth is depicted (*i.e.* no parallelization). In absence of bus contention, execution times are expected to remain constant, as all cores operate simultaneously. A cache is interposed and configured with two alternative policies, namely WRITE-BACK (WB) (writes go to cache only, and are copied back in memory only when the cache line is evicted) and WRITE-THROUGH (WT) (writes always go to both cache and memory). The WB policy is clearly minimizing the amount of traffic which spills on the interconnect, but at the cost of additional complexity in the cache controller (*dirty bits* have to be tracked). The advantage of WB, which may not be fully clear when designing the IP core alone, is evident here. With WT caches, six or more processors are enough to congest an AMBA AHB interconnect, causing a progressive performance degradation. With WB caches,

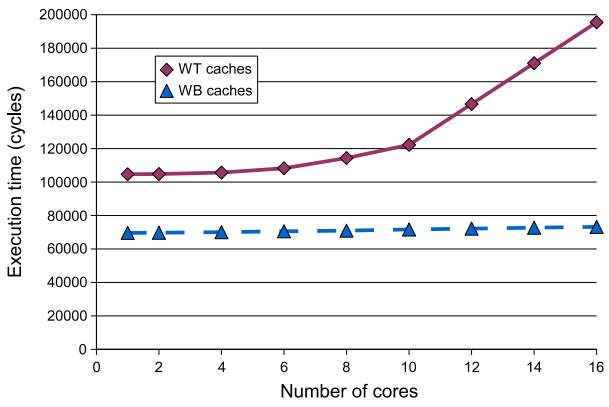


Figure 3.6: Performance *vs.* interconnect congestion.

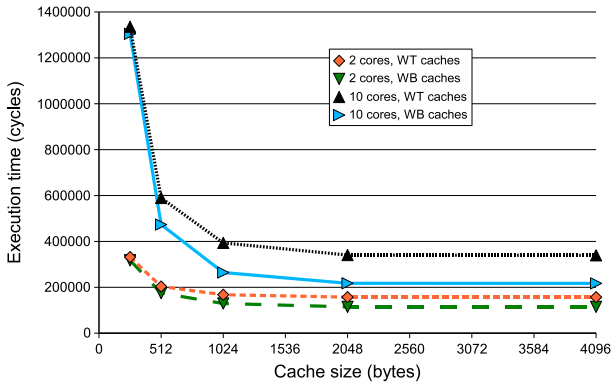


Figure 3.7: Performance *vs.* cache size.

the amount of writes on the bus is drastically lower, and up to sixteen cores can be attached to the same fabric without significant bottleneck effects.

Subsequently, just by changing a command line parameter, we repeated the same experiment with varying cache sizes (Figure 3.7). As the chart shows, bigger caches help performance, but under low interconnect congestion (few IP cores on the bus and/or WB caches), their impact is much less than under high congestion.

To further showcase possible design space scenarios, we created a mixed platform with one LISA ARMv7 core and one or more LISA FFT coprocessors. The latter devices were designed to optimally accomplish a specific task, namely a Fast Fourier Transform. Since they internally perform parallel computation (Figure 3.8 on the following page), they feature high bandwidth requirements and contribute heavily to bus con-

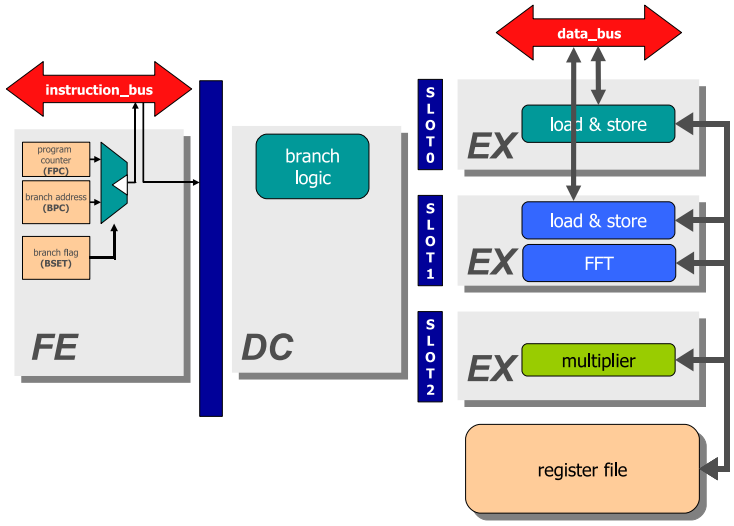


Figure 3.8: A 3-Slot VLIW FFT Processor.

gestion. Figure 3.9 on the next page plots the latency, as seen by the ARM core, to complete bus transactions when increasing numbers of FFT cores are working in the background. A steep latency increase can be noticed, prompting the designer to quantify the amount of communication resources needed for the deployment of FFT coprocessors.

To highlight how the availability of a full platform, including memory hierarchies and interconnect models, enables the study of non-trivial effects in MPSoC systems, we show in Figure 3.10 on page 64 the polling behaviour of a DES encryption benchmark, where two control tasks (*initiator* and *terminator*) supply and collect chunks of raw data to a variable amount of parallel *worker* tasks that perform the actual encryption/decryption. We tested the system with one to six worker tasks, each running on a different LISA core. The worker tasks have to synchronize with the initiator and terminator tasks by semaphore polling before being able to exchange data chunks. The plot depicts the overall amount of system polling as a function of varying frequencies of polling executed by the initiator and terminator tasks. With few workers, the workload is very unbalanced (the initiator and terminator tasks have comparatively little to do) and configuring them for frequent polling is only a waste of interconnect bandwidth. As more workers are added, frequent polling becomes increasingly useful because more data chunks have to be distributed and collected per time unit. If the polling interval of the initiator and terminator becomes too wide, roles reverse, and it is the worker tasks which have to perform heavy polling before exchanging data. Therefore, the global polling amount in-

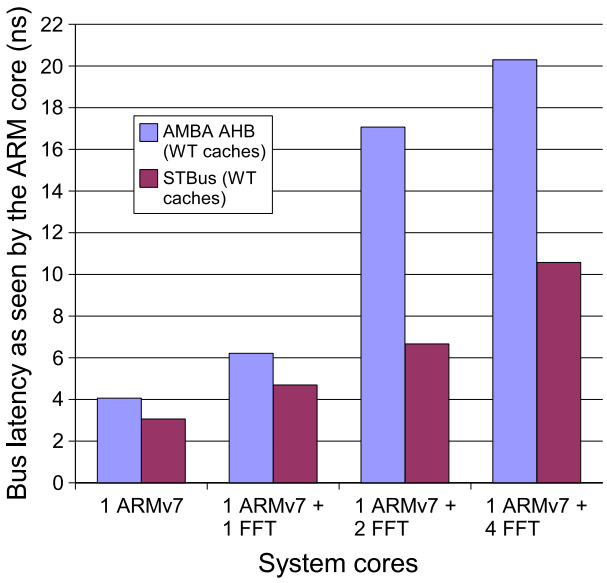


Figure 3.9: Bus latency of a mixed ARM + FFT platform.

creases again. The case with a single worker has the rightmost knee point (the control tasks are very lightly loaded, and can afford sparse semaphore checks) but the highest absolute polling amounts (the chance of hitting optimal synchronization points without much polling is very low). When the polling interval becomes large, all lines exhibit a shaky trend, because randomly missed synchronization points imply a long wait before the next semaphore release event and long strings of polling activity.

3.4 Reactive Traffic Generation

A primary design paradigm for MPSoCs is the separation of the communication and computation concerns, as this enables IP reuse and shorter design times. When tackling the communication part, whose optimization is key to the overall performance of an MPSoC platform, it is key to rely on traffic models that are realistic and accurate. A critical problem however is that traffic models should capture not only the behaviour of the applications, but that of the applications *running on top of a stack of hardware and software*; this includes properties which are not easy to reproduce, such as synchronization.

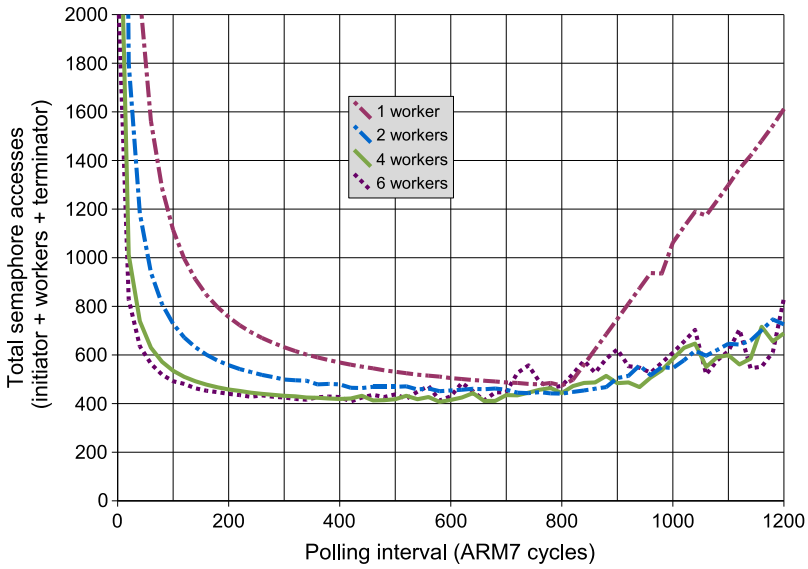
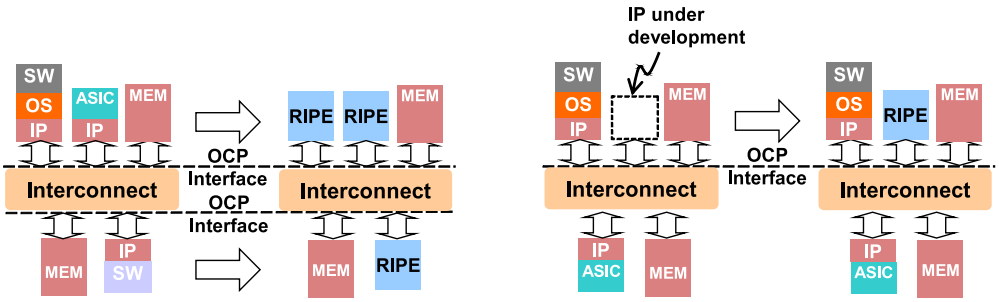


Figure 3.10: Polling behavior in the DES benchmark.

In presence of concurrent tasks running on multiple processors, the characterization of traffic patterns is not simply a matter of stochastic modelling [98, 193] or trace-based regeneration [108]. For example, an inter-processor synchronization mechanism based on semaphore polling generates different amounts of traffic depending on the relative timing of accesses. This may create traffic spikes and localized congestion of the interconnect, but is very hard to predict in advance, impacting the accuracy of the traffic model. A less simplistic way of modeling the MPSoC system is to describe it entirely as a cycle-true model [194]. This yields the most accurate information for performance analysis and subsequent interconnect optimization. However, the implementation time and the simulation speed of such models is clearly a limit to widespread adoption. In an industrial design, such complete and cycle-accurate platform models may actually only become available after the product tapeout - long after the deadline for the optimization of the system interconnect.

For the purposes of the interconnect designer, a valuable tool for exploration and optimization would be a black-box model that, when plugged at the ports of the interconnect, would act like an IP core, injecting realistic traffic with clock cycle accuracy. A key desired property would be *reactiveness* to the surrounding environment, that is, the ability to adjust traffic patterns depending on synchronization events which are associated to the system as a whole, and could not be properly rendered by any



(a) RIPE as a simulation aid, replacing existing IP cores.

(b) RIPE as a design tool, acting as an IP core still under development.

Figure 3.11: Possible usage scenarios of RIPE.

traffic generation device in isolation. An example has been given above with synchronization by semaphore polling; more complex scenarios include system-triggered interrupts. Only a tool featuring such reactiveness really allows for meaningful analysis of the interconnect choice and performance. The fundamental problem, however, is how to generate such realistic traffic patterns.

We investigate this problem and propose a solution in the form of a REACTIVE IP EMULATOR (RIPE) model. RIPE is a tool that can reproduce IP traffic with cycle accuracy. This is done by influencing the type and the timing of the communication transactions based on the current internal state *and* the synchronization properties of the MPSoC system as a whole. A part of the novelty of our approach is that we use additional and readily available system-level information (such as, for example, the knowledge of the location of semaphore variables in the memory space) to automatically detect synchronization events and respond to them during runtime. These elements allow us to reach the goal of reactiveness. RIPE is implemented as a SystemC module with OCP pinout, readily allowing for integration within MPARM Section 3.2 on page 49.

The proposed RIPE device can be used in several ways. One possibility (Figure 3.11(a)) is to leverage its features to replace existing IP cores [195, 196]. The idea is to accurately reproduce communication transactions based on prerecorded system traces. By swapping away IP cores for RIPE blocks in the reference cycle-true system, subsequent design space exploration of the interconnect can be performed independently while keeping a very high level of accuracy and speeding simulation up. A validation scheme for this type of flow will be presented in Section 3.4.3 on page 76.

On the other hand, a RIPE device can also be used (Figure 3.11(b) on the preceding page) in the early stages of the design space exploration, when not all IP cores may be finalized yet, to explore co-simulation effects and see the impact of hardware changes on the software stack. In this scenario, the interconnect designer may want to leverage RIPE as a design tool, by hand-writing programs to test specific realistic synchronization-intensive scenarios which would be very difficult to study with traditional traffic generation flows. For example, in Section 3.4.6 on page 88 we will present a case study where the impact on execution time of variable densities of interrupt events can be investigated.

3.4.1 Application Reactiveness in MPSoC Environments

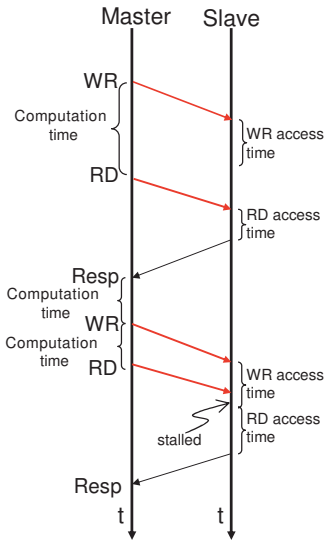
A first mandatory condition for this work is to investigate the requirements for accurate modeling of communication events on MPSoCs. In MPSoC environments, several different types of system-level communication may occur. We identify three broad categories: (i) processor-initiated communication towards a private resource but across a shared medium, (ii) processor-initiated communication towards a shared resource, (iii) system-initiated communication towards a processor, which typically happens by means of interrupts. Especially the second and third types are examples that illustrate the reactiveness of IP cores, a property which must be carefully emulated to accurately model their traffic patterns. In the following, we present examples of representative applications which are impacted by system-level constraints, such as the sharing of interconnect and memory resources.

Communication with a Private Resource

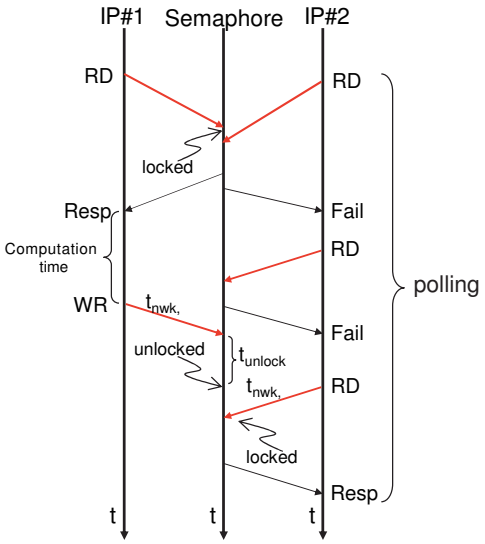
Let us first consider a simple case of processor-initiated communication towards an exclusively owned slave peripheral, but across a shared medium (Figure 3.12(a) on the facing page). We code a simple application, **matrix**, which involves one task per processor, each performing some private computation. No inter-task or inter-core synchronization is required. However, all tasks compete for access to the same interconnection resource.

In this example, the communication needs of the application are quite easy to model; the result is a simple list of transactions interleaved with computation. The model is made only slightly more complex by the issue of bus congestion, which makes the data access time unpredictable.

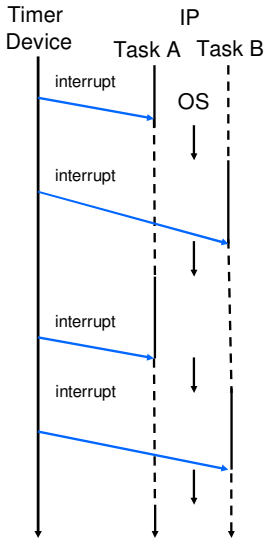
To better understand this issue, please consider the first two master transactions, a write (`WR`) and a read (`RD`). The `WR` transaction can be as-



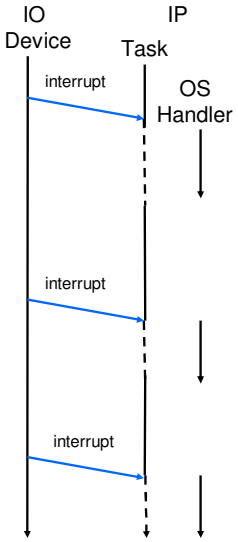
(a) Communication with a private memory (**matrix**).



(b) Communication with a shared memory including polling (**poll**).



(c) System-initiated timer interrupt causing a context switch (**multi**).



(d) System-initiated I/O interrupt causing a driver to respond (**IO**).

Figure 3.12: Timelines for relevant MPSoC application scenarios.

sumed to be non-blocking for the IP core, which therefore simply issues a request and continues its computation. The RD, on the other hand, uses blocking semantics. Therefore, the response has to make its way back to the master, and only then can computation resume. The overall latency is also a function of the congestion on the interconnect. Therefore, it is not enough to capture a time-annotated list of transactions, as the timing information depends on the specific interconnect. From the emulation point of view, however, a model can be easily achieved as follows. The latency due to congestion and actual slave response time can be discarded; the only essential points to capture are just the two transactions, the delay between the WR assertion and the RD assertion (which is computation time), and the delay between the RD *response* and the following command. This information makes it possible to emulate the IP core behaviour on any given interconnect, even one having very different latency properties.

Similarly, the stalling behavior observed in the next set of instructions (WR-RD) does not need to be explicitly captured in a RIPE model, since, from a processor perspective, it simply appears to be part of the slave response time.

Requirement #1: This observation leads to the concept of *time-shifting* behaviour: consecutive transactions are tied to each other, and are issued at times which are a function of the delay elapsed before receiving responses to previous transactions. For emulation purposes, only the length of the computation periods (which can be modeled by idle waits) and the transaction types are needed.

Modeling requirements of this simple category of traffic can be predicted or inferred given an algorithmic specification. In [197, 98] such an inference is drawn to test the interconnect. However, these models do not hold for more complex traffic types, as those that will be shown below, unless extremely detailed models of the underlying hardware and software are provided. This includes cache replacement policies, simultaneous tracking of each processor state, *etc.*.

Communication with a Shared Resource

In the simplest synchronization case, one or more processors competing for a shared resource may poll a semaphore to gain resource access. As an example, let us consider a multimedia application called **poll** (Figure 3.12(b) on the previous page). For this case, we map a single task onto each IP core. Tasks are programmed to communicate with each other in a point-to-point producer-consumer fashion; every task acts both as a consumer (for an upstream task) and as a producer (for a downstream task),

therefore logical pipelines can be achieved by instantiating multiple cores. Synchronization is needed in every task to check the availability of input data and of output space before attempting data transfers. To guarantee data integrity, semaphores are provided. A semaphore is a special binary-valued memory mapped device for which test&set functionality is provided in hardware. Therefore, an `RD` returns the semaphore state and, if the semaphore is currently "unlocked", also changes its state to "locked". By checking the return value of the `RD`, the master issuing the command can decide if the locking was successful or if the semaphore had already been locked by another task. The unlocking can be performed with an explicit `WR` command of the "unlocked" value. In the **poll** application, the consumer checks a semaphore before accessing producer output. If the semaphore is found locked upon the first read, the application *reacts* with a continuous polling strategy, whereby it regularly issues read events until eventually the semaphore is found unlocked. Since the transactions occur over a shared interconnect, the unlock event (in this case the `WR` issued by IP#1) and the success of the next request (`RD` event by IP#2) are interdependent.

In the figure, only if the IP#2 `RD` event is issued at least $t_{nwk,IP\#1} + t_{unlock,S} - t_{nwk,IP\#2}$ after the unlocking by IP#1, then IP#2 will be granted the semaphore and additional polling events will not be required. Therefore, depending on network properties, a variable amount of transactions might be observed at the ports of the IP cores. This demonstrates that the *time-shifting* behaviour introduced before is not sufficient when multi-master systems are taken into account. The arbitration for resources in such designs is timing-, and thus architecture-, dependent.

Requirement #2: The state of the shared resources needs to be tracked. For emulation purposes, the semaphore locations must be known and monitored, and the devices must make use of this information to adjust their execution flows.

System-Initiated Communication

System-initiated communication towards a processor is generally performed by means of interrupts, and an OS is in charge of the handling. In reacting to the interrupt, however, the degree of interaction between the OS and the application can vary noticeably. We present here three examples, which are representative of a vast class of execution flows. The RIPE model we will propose can capture all the dynamics of these test cases, given proper insight on the mechanics of the applications and the OS.

As a first example (Figure 3.12(c) on page 67), we create a test application (**multi**) where timer-generated interrupts are used to drive the OS scheduler. In this case, only the OS is aware of the interrupts, while the user tasks are transparently paused and resumed while executing a single stream of operations. In our application, we introduce two tasks per processor, having unbalanced bandwidth needs; therefore, every interrupt causes an abrupt shift in traffic workload for the interconnect.

In a second example (Figure 3.12(d) on page 67), the **IO** application is composed of a main execution task and of a driver for an Input/Output (I/O) device; the latter is in charge of responding to interrupts sent by the hardware device. The driver operation is bandwidth-intensive, causing traffic spikes on the interconnect.

A third test case (**pipe**) features the same logical behaviour of the **poll** example shown above, that is a pipeline of multimedia processing tasks, but leverages interrupts instead of polling to reduce the congestion on the interconnect and the energy waste upon synchronization points (see Figure 3.13 on the facing page). This scenario features a very tight coupling between the application and the interrupt handling; for example, upon an unsuccessful lock acquisition by a consumer, the application interacts with the OS to be descheduled and to be resumed only upon the next interrupt event, which will flag the availability of new data. On the other hand, if the lock is immediately available, the application proceeds directly. Interrupts may be ignored if they are issued ahead of time, that is, if the notification of new data availability arrives before the consumer is ready to process a new message.

As can be seen, modeling these applications and their impact on interconnect performance is not trivial, presenting a major hazard for any traffic emulation device. A complete emulation of the hardware and software stacks is needed to properly determine the traffic behaviour at the IP core pinout boundary.

Requirement #3: In presence of interrupt facilities and of an OS, the execution of every application task, of the OS kernel and of interrupt handlers must be independently identified and modeled. This can be achieved by tracking the occurrence time of interrupt events and application resumptions. The traffic emulator should then be able to model the IP core behaviour independently of the interrupt occurrence time.

Timing Dependency of Applications

So far, we have evaluated the implications of different MPSoC traffic categories. These requirements are not derived in a ad-hoc fashion, but are

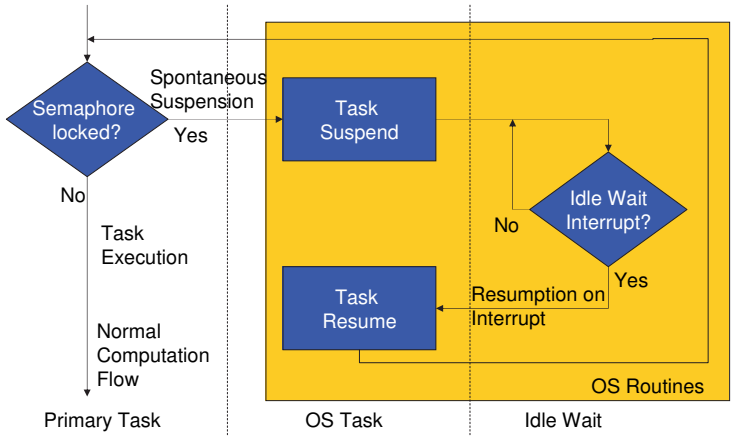


Figure 3.13: Application flow of **pipe**.

representative of typical timing-sensitive real-life applications [198], such as multimedia stream processing, time slicing mechanisms in OS schedulers, and I/O device handling. In such applications, the overall performed computation does not change depending on the order of arrival of external events. So, while an execution trace of these examples show widely varying traffic patterns depending on external timings, the major computation blocks are still recognizable. Even though applications with even more timing-dependent behaviour do exist, modeling them would require an intra-task notion of context switching. At this stage, we believe that the complexity of such an effort for a whole MPSoC in a generic way would be excessive and anyway unsuited for a black-box component such as the RIPE intends to be.

3.4.2 RIPE Model and Implementation

The key principles of RIPE can be rendered in several types of devices, including behavioural modules, programmable simulation devices and even programmable hardware blocks. We choose to explore the second alternative, which provides the maximum flexibility while leaving future embodiments open. Therefore, we specify an abstract RIPE multi-threaded Instruction Set Architecture (ISA) and we build a RIPE SystemC simulation device with OCP (Open Core Protocol) 2.0 [12] sockets at its ports. The RIPE model allows for easy programming of sequences of communication transactions interleaved with idle waits, and is capable of sensing and responding to system events and properties. The details of our implementation, and an example RIPE program modeling one of

the applications introduced in Section 3.4.1 on page 66, can be found in Section 3.4.2 on the preceding page.

In this section, we describe a particular implementation of the RIPE concept based on an instruction set architecture, which is capable of fulfilling the above presented requirements of reactive behaviour.

RIPE Instruction Set Architecture

Applications such as those outlined in Section 3.4.1 on page 66 can be emulated either within a behavioural/transaction-level module or with an Instruction Set Architecture (ISA)-based device. While our RIPE model and the supporting toolchain (Section 3.4.3 on page 76) could also be targeted at the deployment of behavioural models, we choose to develop an ISA-based RIPE implementation, and we describe it in SystemC [185]. While the behavioural model may have a slight advantage in simulation speed over a programmable device, it also requires a recompilation of the simulation platform every time the application to be modeled changes. During design exploration, such a step would be required to study multiple applications on the same platform. A programmable model, with a fixed emulation device and user-written programs, avoids this time consuming operation, introducing instead a simple programming language paradigm. The designer may not even need to use the language at all when using automatic translation of the traffic specification into a RIPE program, as outlined in Section 3.4.3 on page 76. Further, a future goal of our project is to build test chips containing interconnect prototypes. The ISA-based approach is very attractive for this purpose, because it can naturally map onto a hardware device to inject traffic on test chips. In [199], the potential of this type of architecture has already been shown within an FPGA-based emulation platform.

The RIPE is implemented as a non-pipelined processor with a very simple instruction set, as listed in Table 3.1 on the facing page. Its external pinout matches the OCP 2.0 [12] specifications for a master interface. Hardware interrupts are available on the sideband portion (`SInterrupt`) of the OCP interface, and an internal software interrupt facility is also present. A future planned extension is the support for the multithreading extension of the OCP protocol, thus supporting outstanding and out-of-order transactions. Any other interface standard, such as AMBA AXI [4], could also be supported depending on the interface required by the interconnect under study.

The RIPE program that controls the device behaviour contains code to model one or multiple tasks. These tasks might be actual tasks running

Instruction	Description
Communication Instructions:	
`Read(AddrReg)`	Read from an address
`Write(AddrReg, DataReg)`	Write to an address
`BurstRead(AddrReg, CountReg)`	Burst read from address set
`BurstWrite(AddrReg, DataReg, CountReg)`	Burst write to address set
Flow Control Instructions:	
`If(arg1, arg2, operand)`	Branch on condition
`Jump(label)`	Branch direct
`Idle(counter)`	Wait for given no of cycles
`SetRegister(reg, value)`	Set register (load immediate)

Table 3.1: RIPE instruction set.

on the IP core which is being emulated, or chunks of the OS layer, such as its native interrupt handlers and scheduler. We instantiate in the device a Program Counter (`PC`) register, an instruction memory and a register file for each task specified by the program; no data memory is needed. A context switch among tasks in the task pool is realized simply by referring to the corresponding set of `PC` and register file.

The instruction set comprises four instructions for data transfers, whose operation can be controlled by putting proper values in the operand registers. These instructions are blocking, *i.e.* the RIPE execution is suspended until completion of the OCP handshake, which for a read will include the latency of the response over the network.

Four flow control instructions are also available to realize the reactive behaviour. The `SetRegister` instruction loads an immediate 32-bit value, which is written into the specified register. The `If` and `Jump` instructions are used to change the execution flow, while the `Idle` instruction models the IP computation periods with idle waits. Within the register file, most registers are general purpose (typically used to set address and data values for OCP transactions), and their number can be configured. Some registers are designated as special purpose. For example, since in specific flow control scenarios the data returned by a `Read` command must be available for evaluation (*e.g.* in case of semaphore checks), the RIPE device provides in Register 4 the response to the preceding read. Table 3.2 on the next page shows all designated special purpose registers.

Of the interrupt-related registers, Register 2 is used to (un)mask critical sections of the RIPE program from external, system-issued interrupts. For

Special Register	Name	Usage
	Interrupt Registers:	
2	`IntrpMaskReg`	Masks or unmasks interrupts
3	`TaskIDReg`	Stores a task ID
5	`SWIntrpReg`	Sends a software interrupt from within the program
	Other Registers:	
4	`RDReg`	Stores the data value returned by a `Read(AddrReg)` instruction

Table 3.2: RIPE special registers.

example, as seen in **pipe** (Figure 3.13 on page 71) the interrupts are only enabled after the task has suspended, while they are masked during normal operation. Register 3 can be programmed to hold the task ID of the next task to be loaded and run on the RIPE device out of the available task pool. Register 5 allows the RIPE program to assert "software interrupts". The RIPE model instantly reacts to unmasked hardware or software interrupts by loading the program and register set corresponding to the next task to be emulated, which is identified by Register 3. The usage of the special registers will be shown in Section 3.4.2.

Programming Language and Assembler

To better understand the programming model of the RIPE device, Listing 3.2 on the facing page presents the main structure of a program to model the **IO** application introduced in Section 3.4.1 on page 66. Statements starting with a semicolon (`;`) are inlined comments.

The RIPE program starts with a header describing the core and the task identifier: `MASTER[<coreID>, <taskID>]`. All of the tasks running on any given IP core are described within a single program, so that there is one program per RIPE device. Recall that **IO** models an application with a linear program flow, which can be suspended by the OS to process I/O interrupts. Therefore, two tasks are described: task #0 (the main application) and task #1 (the interrupt handler) within the same master IP (core ID 1).

The next few statements express initialization of the register file for this task. Unique labels should be used for register names/tags. This allows correct initialization and easy identification of the registers within the program. For task #0, the main body of the RIPE program, the execu-

tion flow is linear, composed of sequences of reads and writes interleaved with register accesses (mostly, to set up transaction addresses and data). Flow control instructions might be inserted where appropriate, but are not needed for this application. Note the initialization of interrupt-related registers at the top of task #0; upon a hardware interrupt, the RIPE swaps the context to the task having the ID provided in `TaskIDReg`, *i.e.* to task #1 (the I/O interrupt handler). Since task #0 can be suspended by the OS to process I/O interrupts, `IntrpMaskReg` is set as unmasked, allowing for such suspension.

The OS-driven context switch traffic and the I/O handler routine are programmed in task #1. Within the interrupt routine (starting with label `IntrptHandler`), which is the critical section of the flow, interrupts are disabled (first instruction of the task body). At the end of the flow, a software interrupt is artificially triggered to restore the normal program flow to task #0. Upon another hardware interrupt in the main task, the interrupt handler routine will be executed again from the top. The flow therefore mimics Figure 3.12(d) on page 67.

Listing 3.2: RIPE program for the **IO** application.

```
 MASTER[1, 0]                          ; Main application (Task 0)
 ; Special Registers
   REGISTER IntrpMaskReg 0             ; Unmask interrupts
   REGISTER TaskIDReg 1                ; Next task ID
 ; General Purpose Registers (GPRs)
   REGISTER AddrReg 0xd0abcdef         ; Initialize address GPR
   REGISTER DataReg 0                  ; Initialize data GPR
   ...
BEGIN                                  ; Comments
   ; Normal application flow
   Idle(10)                            ; Idle for 10 cycles
   Read(AddrReg)
   ...
   SetRegister(AddrReg, 0x10fedcab0)   ; Setup an address
   SetRegister(DataReg, 0x10abcdef0)   ; Setup a data value
   Write(AddrReg, DataReg)
   ...
END


MASTER[1, 1]                           ; I/O driver task (Task 1)
 ; Special Registers
   REGISTER IntrpMaskReg 0             ; Unmask interrupts
   REGISTER SWIntrpReg 0               ; Disable SW interrupts
   REGISTER TaskIDReg 0                ; Next task ID
 ; General Purpose Registers (GPRs)
   REGISTER AddrReg 0                  ; Initialize address GPR
```

```
   REGISTER DataReg 0                ; Initialize data GPR
   ...
BEGIN                                ; Comments
 ; Interrupt Handling Routine
 IntrptHandler
  ; OS Suspension Routine
   SetRegister(IntrpMaskReg, 1)      ; Mask interrupts
   SetRegister(AddrReg, 0x30bebeef)  ; Setup an address
   Read(AddrReg)
   ...
  ; I/O Routine
   SetRegister(AddrReg, 0x30beefcd)
   SetRegister(DataReg, 0x10101010)
   Write(AddrReg, DataReg)
   Idle(121)
   ...
  ; OS Release Routine
   ...
   SetRegister(SWIntrpReg, 1)        ; Trigger SW interrupt
   SetRegister(SWIntrpReg, 0)        ; Deassert SW interrupt
   Jump(IntrptHandler)               ; Get ready for next event
 ; End Interrupt Handling
 END
```

The RIPE program containing the aforementioned instructions must be transformed into a binary file for use within the RIPE device. An assembler tool takes care of this step, with a one-to-one correspondence between program instructions and binary opcodes. Within the binary, the individual task sections are appended in order of their task ID. A header with a small task lookup table is prepended.

During the setup phase, the RIPE device loads the binary, and based on the information encoded at the start of the binary file, determines the number of tasks and the amount of program memory and the register file size to be allocated to each one.

3.4.3 Using RIPE Programs

Depending on the IP model availability to the designer, different ways exist to write RIPE programs which best represent the desired type of traffic.

Trace Parsing and Replay

In this scenario, as is seen in Figure 3.11(a) on page 65, the availability of a pre-existing model for the IP under study is assumed. Here, the RIPE program generation goes through two steps. First, a reference simulation

is performed by using the available IP models, and an execution trace for each IP master in the system is collected. The trace is a very straightforward log of events on the OCP pinout; entries include requests, responses and interrupts, all of which annotated with timestamps. A sample trace snippet is presented in Figure 3.15(a) on page 80. Second, the trace is parsed with an off-line tool. The output of the tool is the desired RIPE program. The resulting program is coded to behave exactly as the original IP model in the native system, and to behave as the core would do when plugged to a different interconnect. This program is now ready to be used for cycle-accurate interconnect design space exploration with extremely realistic test traffic.

This type of flow is useful whenever the pre-existing IP model is not available, due to licensing or technical issues, for the next co-exploration phase. In this case, the RIPE can provide a quick functional yet cycle-accurate port of the IP model to an MPSoC interconnect. Even if the IP model is available, a simulation speedup can be achieved without significant losses of accuracy (Section 3.4.5 on page 83). The off-line parsing tool must of course have some knowledge about the traced application in order to correctly analyze and rearrange execution traces into RIPE programs. While this effort is not trivial, it is feasible and provides a path for validation of the presented RIPE device in a complete cycle-accurate flow, as described in Section 3.4.3 on the facing page.

Trace Editing

In a related scenario, an IP model might be available, but it may differ under some respect from the IP that will eventually be deployed in the SoC device. In this scenario, the RIPE may be used to approximate the IP, as seen in Figure 3.11(b) on page 65. The designer may then follow a route similar to the one outlined above, but with an additional step of editing the reference trace so that it more closely resembles that of the target IP. Some examples of the editing steps which are possible include:

- Removing or adding bus transactions to model a more or less efficient cache subsystem
- Removing or adding bus transactions to model a more or less comprehensive emulated ISA
- Altering the delay among bus transactions to reflect different pipeline designs or timing properties

- Grouping or ungrouping bus accesses to reflect write-back *vs.* write-through cache policies

It is certainly reasonable to expect that the alteration time of the RIPE program will be substantially shorter than that required to develop or refine the target IP model, thus allowing for earlier exploration of the interconnect design space.

In this scenario, overall cycle accuracy with respect to the eventual system is of course not guaranteed. However, the RIPE will still be able to react with cycle accuracy to any optimization in the SoC interconnect. Provided that the transaction patterns are kept close to the ones of the target IP core, the approach will result in valuable guidelines.

Direct Development

Finally, RIPE programs can be written from scratch without reference IP traces. In this case, the flexible RIPE instruction set allows for a full-featured traffic generation system. The availability of built-in flow control management lets the designer implement the same synchronization patterns which are present in real world applications (see Section 3.4.2 on page 71). Additionally, the application chunks enclosed within synchronization points can quickly be rendered by exploiting the flexible loop structures provided by the RIPE ISA, thus providing capabilities at least on par with those of traditional stochastic traffic generator implementations as seen in [193, 98, 199]. In the very first stages of development, the RIPE can also be deployed as a validation tool, to check the correct functionality of the interconnect under the load of the supported transaction types. An alternate possibility, as demonstrated in [200], is using the RIPE as an interface between formal and simulation models in a hybrid environment. Here, the RIPE programs are written based on guidelines provided by the arrival curves obtained by formal analysis methods. These programs are then used to generate communication events for the simulation environment. Thus, the versatility of our RIPE flow allows for deployment in a number of situations.

3.4.4 RIPE as a Simulation Aid

As an example of RIPE functionality, we now adopt the flow presented in Section 3.4.3 on page 76 to show its feasibility and to create a validation environment for the RIPE device accuracy.

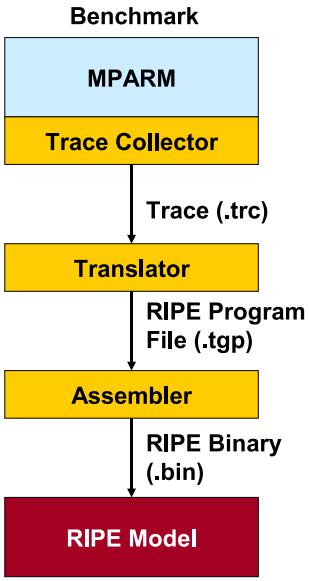


Figure 3.14: Trace to RIPE Program Flow.

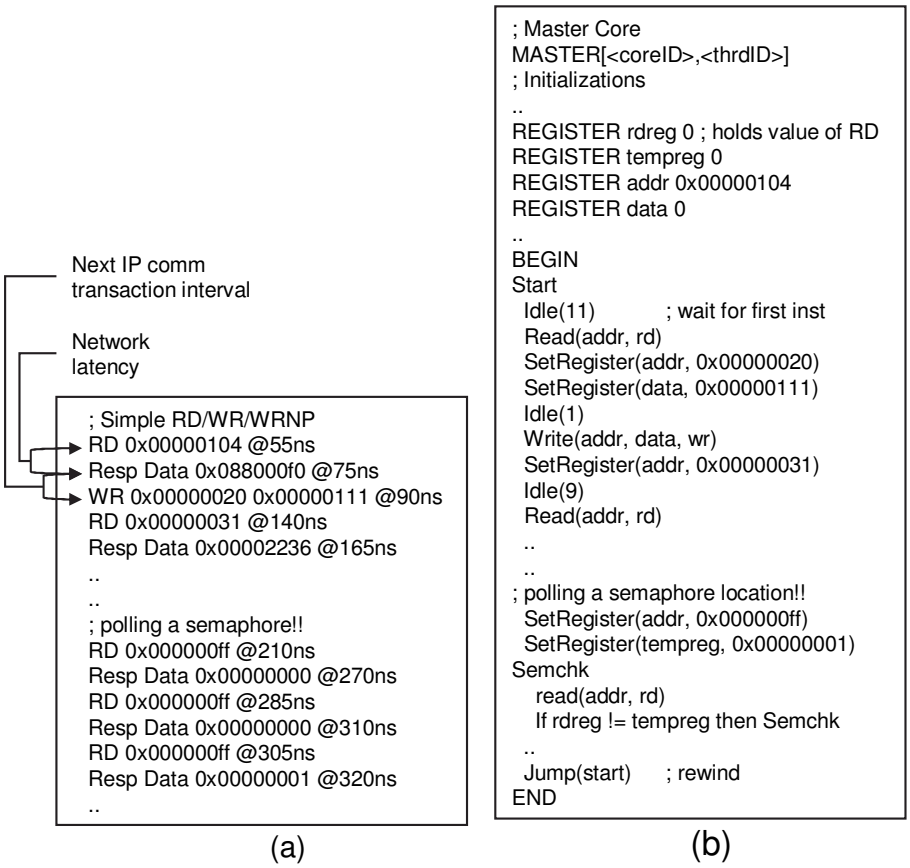
MPARM Trace to RIPE Program

We integrate the RIPE model into MPARM Section 3.2 on page 49. The use of the OCP v2.0 protocol at the interfaces between the IP cores and the interconnect allows for easy exchange of native cores with RIPE blocks (Figure 3.11(a) on page 65). To record execution traces, the OCP interface modules within the MPARM system (the network interfaces in the case of ×pipes and the AMBA AHB bus master) were adapted to collect traces of OCP requests, responses and interrupt events in a predefined file format (`.trc`).

It is worth stressing that modeling the communication patterns described in Section 3.4.1 on page 66 is not trivial. The amount of annotations that can be extracted from the application and its traces reflects the programmer's degree of knowledge and access to the application synchronization schemes, to the interrupt routines and to the OS internals.

The RIPE validation flow is illustrated in Figure 3.14. During the reference simulation, traces are collected from all OCP interfaces in the system. The address and (if any) data fields of the transactions are also observed. Trace entries may contain one of many transaction types: single or burst read/write requests, assertion of hardware interrupt, arrival of response, *etc.*. Figure 3.15(a) on the following page shows an example trace.

The next step is to convert the traces into corresponding RIPE programs (`.tgp`). The off-line translator tool outputs symbolic code; Fig-



(a) MPARM trace. (b) RIPE program.

Figure 3.15: Example of conversion from MPARM trace to RIPE program.

ure 3.15(b) shows the RIPE program derived from traces in Figure 3.15(a). The automated algorithm in the conversion flow is capable of detecting and capturing many synchronization behaviours, without the need for the designer to handle them manually. We will explain the translator operation in detail in Section 3.4.4. Finally, the assembler tool is used to convert the symbolic RIPE program into a binary image (`.bin`) which can be loaded into the RIPE instruction memory and executed. The entire flow is fully automated and the time taken for this process is discussed in Section 3.4.5 on page 83.

Automated Translation of IP Traces into RIPE Programs

As discussed in Section 3.4.1 on page 66, some prior knowledge about the MPSoC system used in the reference simulation is required to accurately

program the RIPE device. Apart from the sequence of transaction requests and responses, following is a list of information needed for correct operation of the translator:

- The global identifier of the IP core in the system
- The clock period of the IP core
- The addressing ranges representing semaphore (pollable) resources
- The timestamp of interrupt events
- The timestamp of the return from an interrupt handling routine
- The timestamp of a spontaneous request for descheduling by an application

The first three pieces of information are encoded in the trace filename, the rest are explicitly or implicitly (provided some knowledge of the application functions) annotated within the trace file. For example, incoming interrupts are detected on the OCP pinout and explicitly recorded in the trace. On the other hand, returns from interrupt handling routines must be located implicitly by detecting known behaviour, such as a specific memory access at the end of the handler or at the return point in the main code.

We use the system traces given in Figure 3.15(a) on the facing page as an example source for transformation into a RIPE program, and the result is in Figure 3.15(b) on the preceding page. Let the clock period be 5 ns and the semaphore location be `0x000000ff`. As seen in Figure 3.15(b) on the facing page, and described in Section 3.4.2 on page 74, the RIPE program starts with the typical core identifiers. Register `RDReg` is defined as the name of the special register where the value of read transactions is stored (Table 3.2 on page 74).

At the beginning of the trace file, the first communication request, a read (`RD`), occurs at 55ns, meaning that the RIPE has to wait 11 (55/5) cycles before issuing this transaction. Therefore, an `Idle` wait is observed in the RIPE program. When parsing this trace statement, the translator collects the `RD` address and initializes one of the registers marked as available in the register table (tagged as `addr` on top of the program). Based on the principle of *time-shifting* (Requirement #1) discussed in Section 3.4.1 on page 68, we ignore the response to this RD event at 75ns, but note the time interval of three ((90-75)/5) cycles to the next trace event, the `WR` at 90ns. New values have to be set up in the address and data registers, which takes a cycle each (either for updating the already used `addr` and `data` or

for setting up a new pair of registers). An ensuing `Idle` wait is added to fill the gap, then the `WR` instruction is appended.

This translation process continues until the trace entry at time 210ns, when the semaphore address is encountered. By identifying the address as belonging to a semaphore location and knowing the polling behaviour of the original IP core, the translator inserts the `Semchk` label and an `If` conditional statement. This statement checks whether the read value is equal to "1", which reflects an unblocked semaphore. This loop effectively models the semaphore polling behavior. The semaphore address and expected unblock value are set up prior to the loop label to avoid repeated initialization, thus allowing for continuous polling at the maximum frequency rate for unlimited periods. Idle waits can obviously be added in the loop should the original IP core have a low-frequency polling behaviour. All master devices attempting to access this semaphore incorporate the same routine in their RIPE program, thus capturing the system dynamics to meet the Requirement #2.

Within the translator, a register allocation algorithm correctly sets up all the required data in registers before the OCP or the flow control instructions that need them are scheduled for execution. It is possible that streams of closely packed communication requests may leave few or no interleaved idle cycles available for preparing their address (and data, if any). In this case, the translation algorithm exploits the slack (idle wait time) available further above in the transaction sequence for setting up register values ahead of time. However, in case of very long streams of back-to-back writes, a lack of free registers may occur. In this case, the size of the register file must be increased to avoid an accuracy loss due to hiccups in the sequence of writes. We expect the problem to occur with minimal frequency, as two idle cycles among transaction entries are enough to allow for streams of arbitrary length. The problem is of no importance in the context of a simulation RIPE device (as described here), but would have an area penalty in a hardware implementation.

Handling Interrupt Reactiveness

For modeling interrupt routines and OS internals (Requirement #3), specific locations within the trace file, such as interrupt handling routine entry and exit points, have to be recognized by the translator tool to optimally insert the corresponding code as a task into the RIPE task pool. The trace files are always annotated with the time of occurrence of interrupt events. However the exit points need to carefully screened since they depend on the degree of cooperation between the application and OS.

Using the **pipe** example (Figure 3.13 on page 71), let us consider this aspect in more detail. Here, the task is explicitly interacting with the OS internals, as described in Section 3.4.1 on page 66. Usually this interaction can be achieved by OS API calls, without direct access to the interrupt handler code, whose exit point is therefore assumed to be not accessible to the programmer. As a result, the only annotations of significance within the trace file are the synchronization points (semaphore checks) and the interrupt arrival time. The RIPE program thus mimics the flow shown in Figure 3.13 on page 71, first by reading the semaphore location, then choosing to continue or suspend depending on the lock. Upon resumption by a hardware interrupt, a final (re-)check of the semaphore unlock is done to ensure safe task operation. In the RIPE program, this is realized via three tasks; the dotted lines in Figure 3.13 on page 71 mark their boundaries. The primary task represents the main application flow. The interrupts are masked here, as the application is insensitive to hardware interrupts unless in suspension state. If the semaphore is found locked, the flow is routed to load the OS routine which leads the processor to an idle wait. The translator captures the chunk of trace after the semaphore check in an independent OS task, which always yields control to a third task consisting of an infinite loop of idle wait instructions. The easily identifiable sequence of transactions between the eventual arrival of the hardware interrupt and the semaphore re-check is the OS wake-up routine to reschedule the suspended main program, and the translator appends it as a part of the OS task. In the RIPE program, hardware interrupts are used to wake up from the suspension state within OS routines, while software interrupts redirect the execution flow towards the main task. Note that `IntrpMaskReg` is set to "masked" for the regular program and OS execution, and is only unmasked within the suspension task.

After performing the translation described in this Section and after RIPE program assembling, a second set of simulations can be run on a platform with RIPE and a variety of interconnects, thereby evaluating performance of interconnect design alternatives.

3.4.5 Validation Results

The outcome of the validation process should show that the requirements outlined in Section 3.4.1 on page 66 are sufficient to extract IP traffic patterns in a manner which is accurate yet independent of interconnect characteristics. For this purpose, we simulate different applications within the MPARM framework, first using the native ARM cores and then using the RIPE model, and compare the resulting benchmark statistics. We under-

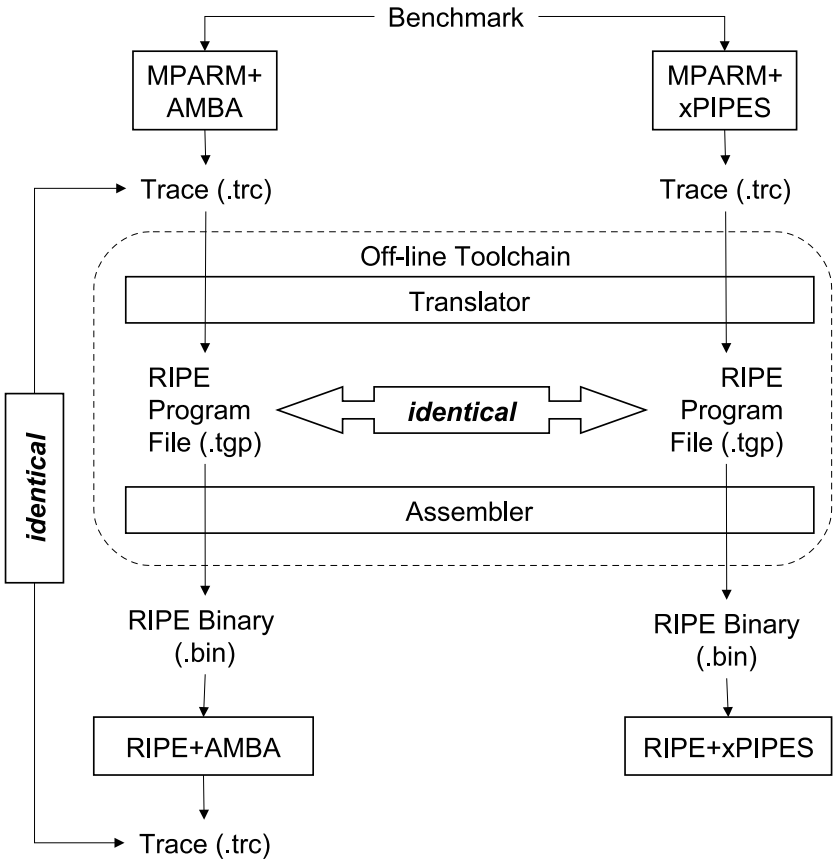


Figure 3.16: RIPE Accuracy Validation Test.

take this experiment for six benchmarks. Each is tested with one to twelve (1P-12P) system processors with cache (see Figure 3.11(a) on page 65) simultaneously plugged to the system interconnect. The aim is to ascertain the accuracy of the RIPE model, device and translation framework when stressed by complex transaction patterns.

Five of the benchmarks are based on the examples introduced in Section 3.4.1 on page 66: **matrix**, **poll**, **multi**, **IO** and **pipe**. One more application (**cacheloop**) is added as a reference to make the validation more comprehensive. **cacheloop** is a dummy program, which continuously performs cache fetches. As such, it is generating no bus transactions, except for a few at boot and shutdown. It is intended as a metric of the maximum simulation time speedup achievable by the replacement of IP cores with another simulation device.

In the first experiment, we only aim at validating the trace collection and translation. Figure 3.16 outlines the process. We run the same bench-

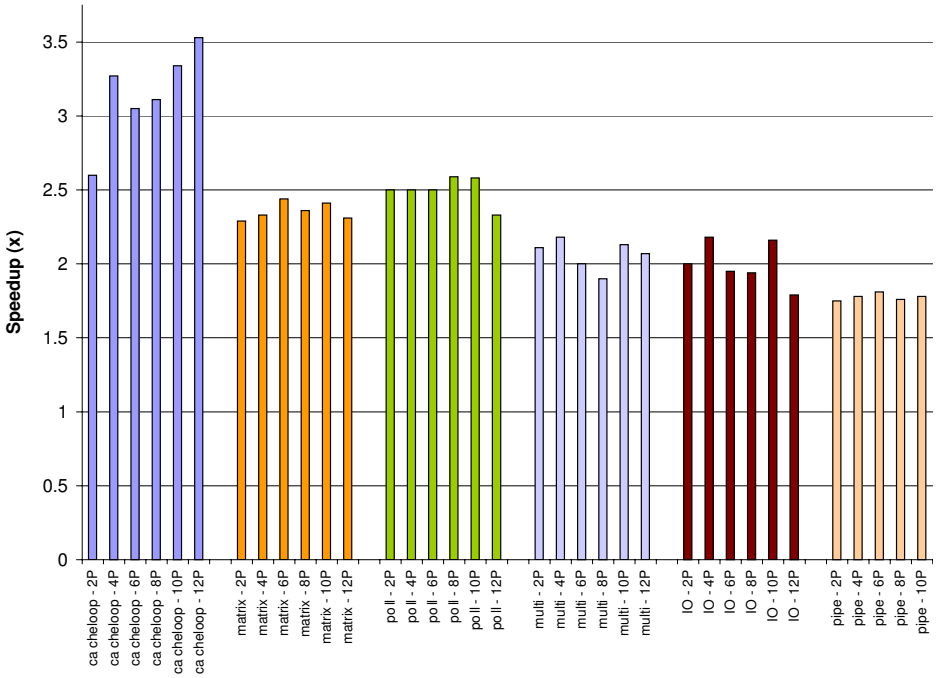


Figure 3.17: RIPE vs. native ARM speedup.

marks over two of the interconnects of MPARM, namely AMBA AHB (Section 2.2 on page 32) and the ×pipes NoC (Chapter 4 on page 95). As expected, we measure very different execution times due to the different interconnect features, and the execution traces reflect these differences. However, after translation, a check across `.tgp` programs shows no difference at all, because the network latency factor is completely abstracted from in the RIPE programs. As a consequence, a trace collected on one interconnect is indeed usable to generate a program to be run on another. This result validates our approach and strengthens the postulate of the requirements outlined in Section 3.4.1 on page 66, which decouples simulation of the IP cores and of the underlying interconnect.

We now proceed to measuring the accuracy of the our design flow, *i.e.* how well the RIPE programs extracted from ARM execution traces match the original execution. Table 3.3 on the next page summarizes the results of simulations[2] done on the AMBA AHB interconnect with ARM processors from MPARM and then with RIPE devices. The columns report the

[2]Benchmarks taken on a Pentium 4 2.26GHz with 1 GB of RAM. The absence of disk swapping effects is checked during simulation. Especially for benchmarks with a short duration, time measurements are taken by averaging over multiple runs and care is put in minimizing disk loading effects.

Benchmarks	# IPs	RIPE				Native ARM				Inaccuracy	
		Execution Cycles	SR	SW	BR	Execution Cycles	SR	SW	BR	Exec %	SR %
cacheloop	2	2500916	0	32	51	2500908	0	32	51	0.000%	0.000%
	4	2501721	0	64	106	2501714	0	64	106	0.000%	0.000%
	8	2503321	0	128	2018	2503314	0	128	201	0.000%	0.000%
matrix	2	1324711	0	117502	186	1324717	0	117502	186	0.000%	0.000%
	4	1326582	0	235004	374	1326588	0	235004	374	0.000%	0.000%
	8	1421281	0	470008	7502	1421272	0	470008	750	0.001%	0.000%
poll	2	881839	7176	71764	254	883977	7201	71764	254	0.242%	0.347%
	4	975267	18241	143596	508	976488	18183	143596	508	0.125%	0.319%
	8	1139110	46044	287356	1016	1140199	46300	287356	1016	0.096%	0.553%
multi	2	1823882	14	85729	24764	1824135	14	85729	24764	0.014%	0.000%
	4	2224333	42	192745	52242	2225867	42	192745	52242	0.069%	0.000%
	8	3482223	98	407707	109820	3482793	98	407707	109820	0.016%	0.000%
IO	2	1156047	2560	68494	18271	1158639	2560	68495	18271	0.224%	0.000%
	4	1446888	2560	145826	36966	1449109	2560	145827	36966	0.153%	0.000%
	8	2325228	2560	300514	74435	2325625	2560	300515	74435	0.017%	0.000%
pipe	2	745386	2601	56004	16293	754998	2601	56004	16293	1.273%	0.000%
	4	1051512	5246	114118	33257	1055056	5247	114298	33313	0.336%	0.019%
	8	1829005	10530	229675	66321	1833183	10530	229675	66321	0.228%	0.000%

Table 3.3: RIPE *vs.* native ARM performance on AMBA. The trend is similar for 6, 10 and 12 IP systems.

overall execution time of the benchmarks (in clock cycles) and the number of single read (SR), single write (SW) and burst read (BR) transactions observed on the bus.

The column labeled Inaccuracy is a measure of the relative difference in simulated cycles and bus accesses when replacing ARM cores with RIPE devices.

The table shows that replacing ARM processors with RIPE devices yields excellent precision, with inaccuracies close to 0% in most cases, resulting in a faithful reproduction of the original execution flow and traffic pattern. The inaccuracies in the SR count and the execution time in **poll** are due to the compounding of minimal timing mismatches caused by the semaphore polling mechanism in RIPE programs. In the real system, the first few semaphore polls are found to occur at a slightly different rate than subsequent ones, due to assembler-level and caching effects. Eventually, polling occurs at periodic intervals. This initial timing mismatch is not captured in the RIPE model, which performs all polling loops at the asymptotic rate. This causes RIPE to be affected by a small timing skew, which impacts subsequent simulation.

The inaccuracies in interrupt-related benchmarks are due to minor issues in properly pinpointing different sections of OS code in the execution trace, as discussed before in Section 3.4.3 on page 76. The near-matching statistics however fully prove the role of the RIPE as a powerful design tool to mimic complex application behaviour in replacement of a real IP core.

Scalability tests, based on simulation time in seconds, performed by increasing the number of processors attached to the bus, exhibit two main different trends, as seen in Figure 3.17 on page 85. **cacheloop** exhibits a fundamentally monotonic trend, showing the advantage of replacing a progressively increasing amount of system cores with a faster device model. Other benchmarks show a fundamentally constant figure, or an asymptotic increase (for example, **matrix**). This seemingly strange behaviour can be explained as follows. An increase in the number of processors implies more traffic on the interconnect, thereby shifting the simulation load towards the interconnect model. At a certain point, the interconnect becomes completely saturated. In this condition, no further speedup is achievable because the simulation time of ARM processors is anyway mostly spent in idle waits for bus responses - leaving no room for improvement to RIPE devices, regardless of their efficiency. To support this analysis, we observe that the lowest speedup is achieved for **pipe**, which is also found to be the benchmark with the highest bandwidth requirements, and therefore the highest load on the interconnect model. We

	Interval among interrupts to same core (ms)	Notes
Reference	2	
Case I	1	
Case II	2	Processors receive interrupts staggered by a 0.5 ms offset
Case III	2	Two processors receive an extra interrupt just after the boot

Table 3.4: Interrupt issue frequency for four different multitasking patterns.

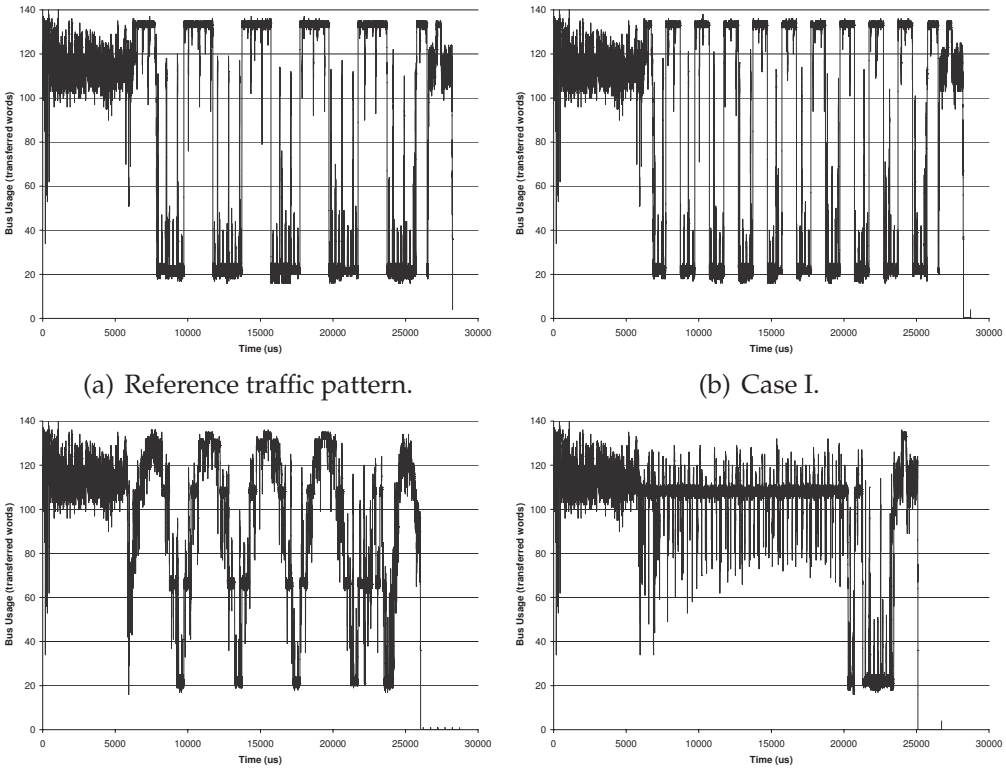
would like to stress that, as **cacheloop** demonstrates, this decrease in simulation speedup is not a shortcoming of our RIPE approach, and is instead a direct consequence of benchmark and system behavior. In absolute terms, a gain of 1.75x to 3.53x is observed when running the benchmark code on RIPE devices as opposed to ARM processors, thanks to the removal of the computation logic within cores. It is noteworthy that even though speedup is not the primary objective of RIPE, it compares favorably to previous work in the area (a speedup of 1.55x is reported in [201]), especially given the fact that it is achieved at the cycle-true level of abstraction.

The time penalty for trace collection is small, and is incurred only once. For example, when running the relatively complex **pipe** benchmark on the AMBA interconnect with four ARM processors, a benchmark run augmented to collect reference traces takes 20 s, and subsequent translation and elaboration requires an additional 12 s for a 5.6 MB trace file. Only one such iteration is needed to validate the RIPE model and for subsequent design space exploration. Additionally, since processed RIPE programs are identical regardless of the reference interconnect in which raw traces are collected, such a collection could be performed on top of a fast transactional interconnect model, further reducing the impact of the reference simulation.

3.4.6 RIPE as a Design Tool

To demonstrate the potential of RIPE as a co-exploration tool, we look at a variant of the **multi** application, first discussed in Section 3.4.1 on page 69, in more detail.

Specifically, we consider a five processor bus-based system with one RIPE configured to act like a timer device. This core triggers the delivery



(a) Reference traffic pattern.

(b) Case I.

(c) Case II.

(d) Case III.

Figure 3.18: Traffic profiles under different traffic injection conditions.

of interrupts at regular intervals to the other four RIPE devices, which as a result switch among two tasks. The two tasks are tuned to have very different bandwidth requirements; one task performs matrix manipulations (**MM**), and heavily relies on data caches to minimize memory transactions, while the second task performs streams of writes (**WS**) to a memory attached to the bus. The **WS** task is very demanding on the interconnect and can easily saturate it, therefore impacting overall system performance. In MPARM, interrupts are triggered by writing to a specific address of a memory-mapped device; therefore, to trigger the interrupts that should come from a timer device, we write a small RIPE program issuing OCP writes at the right times. In turn, this is achieved by parameterized idle waits. Such a program is written in a dozen of lines of RIPE code.

In this case study, using the RIPE, we test the behaviour of this system for different interrupt delivery policies and study the resulting traffic profiles (Figure 3.18 on the previous page). This type of exploration may be useful to schedule bus accesses for real-time tasks in critical systems. The traffic plots show the profile of the bus traffic over time, expressed as transferred data words over a time window of 2 μs.

In all the plots, until about the 6000 μs mark, the bus activity during the OS boot is observed. The boot activity is irregular, but on average quite intensive in terms of required bandwidth, since all the processors are loading the OS and application instructions from the memory across the interconnect. After this mark, application code begins to be executed. In Figure 3.18(a) on the preceding page, a straightforward scheduling policy is used: a timer interrupt is sent to each core simultaneously, therefore causing all of the cores to switch among **MM** and **WS** at the same time. Since interrupts arrive simultaneously to all processors, all of them are in the same task group during any given time slice of execution. As expected, the bus load shifts depending on the task characteristics; the traffic profile exhibits a clear alternating pattern among two disproportionate usage values, with peaks above 130 and a floor of around 20 transactions per time window. The number of transitions between these two limits and the width of each peak correspond to the number of issued interrupt events and the interval between them (see Table 3.4 on page 88). The tail of the plot is representing shutdown code, and is not relevant.

Since excessive contention inflates the response latency of the bus, therefore decreasing performance, the traffic profile must be reshaped to decrease congestion. As is observed in Figure 3.18(b) on the preceding page, as compared to Figure 3.18(a) on the previous page, doubling the interrupt issue frequency does little to mitigate the bus congestion issue; it only shifts the contention to a different time slot. Execution time remains

constant at about 28200 μs.

Let us now consider the impact on the bus activity of staggering the interrupt events. In Figure 3.18(c) on page 89, an interrupt is sent every 500 μs, but two interrupts to the same processor are spaced 2000 μs apart. The traffic profile is smoother; thanks to staggering, **MM** tasks on some cores run in parallel to **WS** tasks on other cores. Over time, the system shifts from running four **MM** tasks to running four **WS** tasks and back, which results in a sinusoidal-like trend with visible steps. Peak congestion is only reached during a shorter fraction of the time, therefore reducing the execution time to about 26000 μs.

To balance the traffic even better, the clear choice is to always overlap two **MM** and two **WS** tasks. This is achieved in Figure 3.18(d) on page 89, where two processors are forced to perform a context switch just after the OS boot, and the subsequent interrupt pattern is the same as in Figure 3.18(a) on page 89. Thanks to much better traffic balancing, the bus never saturates, providing good performance and decreasing the execution time to 25200 μs.

In Figure 3.19 on the next page, the benchmark execution time and the average communication latency for a write transaction on the bus are plotted for the four configurations. As can be seen, Case I exhibits basically identical performance to the reference, while Case II improves 18% on communication latency (and thus 8% on execution time) and Case III improves 24% on latency (and thus 11% on execution time). Therefore, Case III is the best among the alternatives under evaluation.

These experiments highlight that RIPE can be an extremely useful tool to explore communication bottlenecks even without having the real IP cores and benchmarks attached to the interconnect. The flexibility guaranteed by the interrupt handling support provides the designer with additional degrees of freedom and accuracy, allowing a realistic system exploration even in presence of complex communication and synchronization patterns.

3.5 Conclusions

We have shown a complete virtual platform environment, capable of cycle-accurate simulation. This environment, called MPARM, features total customizability in all respects, including the ability to model different IP cores, interconnects, memory hierarchies and software. MPARM in-

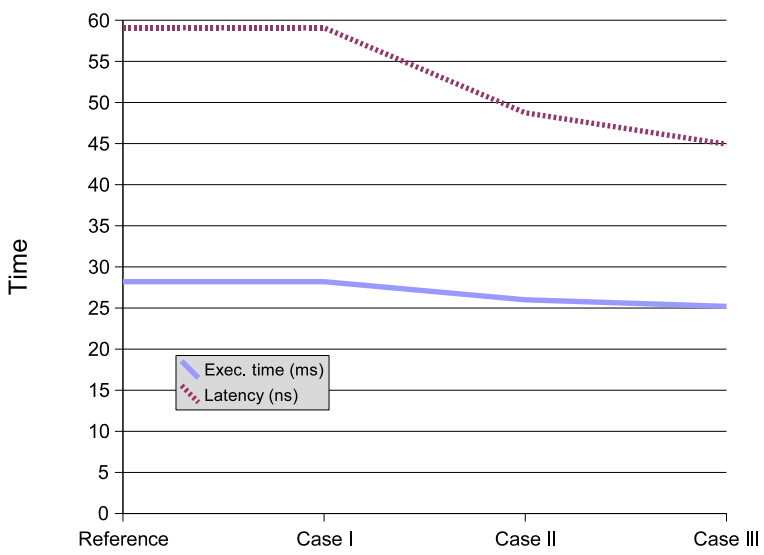


Figure 3.19: Performance of the four synchronization patterns under test.

cludes powerful facilities for debugging, tracing and statistics collection.

MPARM paves the way for the exploration of the degrees of freedom available to the system interconnect designer. Its cycle accuracy is key in modeling often-neglected effects, such as synchronization details, which may have a tangible impact on system performance.

However, the usefulness of MPARM for interconnect design could be limited by two factors: (i) the development time to implement equally accurate models of the IP cores, (ii) the potentially too long simulation times which are intrinsic in simulating at this level of abstraction. To fix these remaining issues, we take two routes.

First, we integrate a state-of-the-art ASIP toolchain within MPARM. This effort merges the platform-level strengths of MPARM, including its openness, with the best features of the LISATek environment, including an environment for the quick development and modification of IP core models, a thorough runtime debugger with graphical interface, and the ability to generate complete toolchains for application compilation and profiling.

Second, we develop RIPE, a reactive traffic generator. Based on requirements extracted from the analysis of significant MPSoC application scenarios, we devise a traffic generation architecture which is capable of interacting with the surrounding environment exactly like a real application stack running on a real core would. The key enabler in this respect is the ability to react to external events, such as interrupts. The result-

ing device can be useful both to estimate the performance of new, not-yet-developed components, or to speed up the simulation of systems for which models already exist, while still keeping cycle accuracy and realistic behaviour.

Many features could be added to MPARM as future research work; an almost unlimited list of components could be integrated to assess its performance and other tradeoffs, from multimedia accelerators (which could be developed in LISA) to dedicated power management blocks. Focusing more specifically on the usage of MPARM as a tool for the study and optimization of interconnect, future work includes certainly the transparent instantiation of bridges among different types of interconnects, to study heterogeneous interconnection architectures.

CHAPTER 4

The ×pipes NoC Architecture

This chapter[1] describes a NoC architectural implementation, called ×pipes. This architecture is the foundation on top of which this doctoral work has mostly been based. ×pipes is a flexible NoC component library, allowing for interconnection in arbitrary topologies. ×pipes development was focused on low area and low power consumption.

4.1 Motivation and Key Challenges

The shortcomings of existing bus- and hierarchical bus-based designs (Section 1.1 on page 3) mandate the development of more advanced architectures, such as NoCs. However, NoCs feature a huge design space and endless possibilities for customization. Further, NoCs have been introduced relatively recently, which leaves many unknowns in terms of design tradeoffs. Therefore, we base our development effort on an analysis of the criteria that we believe are crucial for a successful NoC deployment:

- The potential applications of NoCs are numerous, and each application will require different solutions. Therefore, it is key to design a NoC architecture which is *arbitrarily composable* and *customizable*, so as to optimally match the target requirements.
- NoCs must solve the physical design issues that bus designers are facing. In order to do so, it is imperative to conceive NoCs so as to

[1]The author would like to acknowledge contributions by Paolo Meloni, Prof. Salvatore Carta, Antonio Pullini, Stergios Stergiou and Prof. Luca Benini.

be *predictable* and *tolerant to increasing wiring parasitics*.

- High performance must be delivered, in terms of *bandwidth* and *latency*. Latency is especially critical given the number of components (NIs and switch hops) to be traversed in a NoC-based system.
- The implementation cost must be minimal, including *low power consumption* and *low area occupation*.
- Due to the effect of variability in upcoming technologies, the NoC must be designed for *fault tolerance*.
- The integration effort must be minimal, with *plug&play* platform composability.

4.2 General Concepts

The ×pipes NoC library strives to comply with the requirements listed above, by providing:

- A library of fully synthesizeable components which can be parameterized in many respects, such as buffer depth, data width, arbitration policies, *etc.*. Further, the radix of the ×pipes switches can be arbitrarily chosen (including an asymmetric number of input and output ports), and the switches can be connected in arbitrary topologies.
- A set of link design methodologies and flow control mechanisms to tolerate any wiring parasitics.
- The ability to operate at very high frequencies and/or high data widths, therefore complying with any bandwidth requirement. Simultaneously, ×pipes is optimized for minimum latency, by providing a 1-clock traversal latency of each block in some configurations.
- A minimal architecture, tuned for the lowest possible implementation cost in area and power.
- Facilities for fault tolerance and recovery, in terms of flow control mechanisms (Section 4.4.1 on page 105) and architectural extensions (Chapter 7 on page 207).

- Network Interfaces which can be directly plugged to existing IP cores, thanks to the usage of the standard OCP [12] interface.

×pipes is based on a set of architectural choices:

- ×pipes is fully synchronous. This choice makes it much easier to perform the physical design of the architecture. Facilities to support multiple frequencies are provided in the NIs (Section 4.3.1 on the next page) but only by supporting integer frequency dividers. This choice reduces the implementation cost of the NoC dramatically. Mesochronous and GALS approaches are still possible, by inserting synchronizers at appropriate places in the NoC (Section 8.5 on page 240).

- Routing is static and determined in the NIs (*source routing*). This choice minimizes the implementation cost. While wide area networks often feature dynamic routing schemes, in our internal testing, we found dynamic routing to be too expensive (and deadlock-prone) for the benefits it brings.

- ×pipes adopts wormhole switching [42] as the only method to deliver packets to their destinations. While there is consensus on wormhole switching for best-effort data transfers p:siguenzatortosa2002, p:stergiou2005, p:andriahantenaina2003, some other NoC architectures have proposed a variety of mechanisms, including priorities, timeslots and circuit switching j:bolotin2004, j:goossens2005, p:bjerregaard2005, in order to also support QUALITY OF SERVICE (QOS) for selected data flows. While agreeing on the usefulness of QoS provisions, we observe that implementing QoS as a hardware facility incurs an area and power cost. We thus prefer to support soft QoS provisions by means of a design flow (Chapter 5 on page 115).

- ×pipes supports both input and/or output buffering [42], depending on circumstances and designer choices. In fact, since ×pipes supports multiple flow controls, the choice of the flow control protocol is intertwined with the selection of a buffering strategy. More details can be found in Section 4.4 on page 101.

- ×pipes does not leverage virtual channels [42], as this allows for a much leaner implementation. Instead, parallel links can be deployed

among any two switches to fully resolve bandwidth issues. Deadlock resolution is demanded to the topology design phase (Chapter 5 on page 115).

4.3 The ×pipes Building Blocks

Several architectures have been proposed in the NoC literature. However, all NoCs have three fundamental building blocks, namely, *switches* (also called *routers*), NETWORK INTERFACES (NIS) (also called NETWORK ADAPTERS (NAS)) and *links* [17, 21, 16] (Figure 1.4 on page 6). A NoC is instantiated by deploying a set of these components to form a topology and by configuring them in terms of buffer depth, *etc.*. Some NoCs rely on specific topological connectivity, such as octagon [202] or ring [203], to simplify the control logic, while others, such as ×pipes, allow for arbitrary connectivity, providing more flexible matching to the target application.

4.3.1 Network Interfaces

An NI (Figure 4.2 on the next page) is needed to connect each core to the NoC. NIs convert transaction requests/responses into packets and vice versa. Packets are then split into a sequence of FLOW CONTROL UNITS (FLITS) before transmission, to decrease the physical wire parallelism requirements. The width of the flits in ×pipes is a fully configurable parameter; depending on the needs, ×pipes designs can have as few as 4 wires carrying data, or as many as in a highly parallel bus (64-bit buses typically have close to 200 wires, considering a read bus, a write bus, an address bus, and several control lines). NIs also optionally provide buffering resources to improve performance; since the buffering modules are the same as for switches, we refer the reader to Section 4.3.2 on page 100 for more details.

In ×pipes, two separate NIs are defined, an *initiator* and a *target* one (Figure 4.1 on the next page), respectively associated to system masters and system slaves. A master/slave device will require an NI of each type to be attached to it. The interface among IP cores and NIs is point-to-point as defined by the OCP 2 [12] specification, guaranteeing maximum reusability. OCP 2 supports features such as non-posted and posted writes

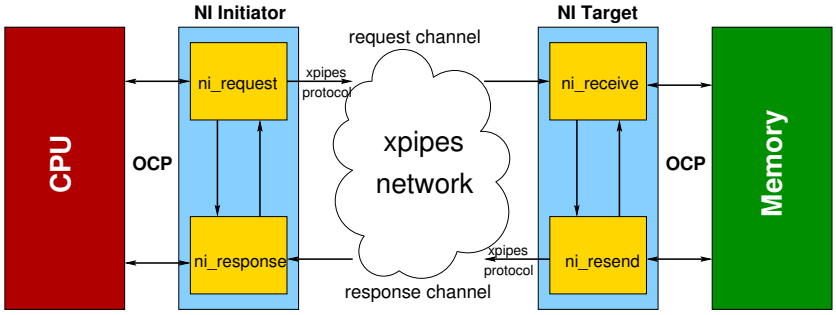


Figure 4.1: ×pipes initiator and target NIs.

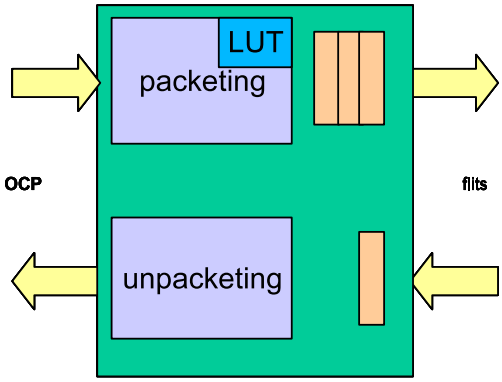


Figure 4.2: ×pipes NI block diagram. The depiction of buffering is for the ACK/NACK case.

(*i.e.* writes with or without response) and various types of burst transactions, including single request/multiple response bursts. To provide complete deployment flexibility, in addition to being parameterizable in terms of flit width, the ×pipes NI is also parameterizable in the width of OCP fields. Depending on the resulting ratios, a variable amount of flits will be needed to encode an OCP transaction.

×pipes NIs optimize the transmission efficiency and latency with an optimized packet format. ×pipes packets minimize the transmissions of information that could instead be regenerated at the receiver (*e.g.*, addresses during bursts). At the same time, the packet format is based on fixed offsets, so that the packeting and unpacketing logic can operate at high frequencies.

×pipes leverages static source routing, which means that a dedicated LOOK-UP TABLE (LUT) is present in each NI to specify the path that packets will follow in the network to reach their final destination. This type of routing minimizes the complexity of the routing logic in the switches. The

alternative, *i.e.* having routing performed by the switches, normally in an adaptive manner, has unclear performance advantages, and in-order delivery and deadlock/livelock freedom concerns are still to be fully solved.

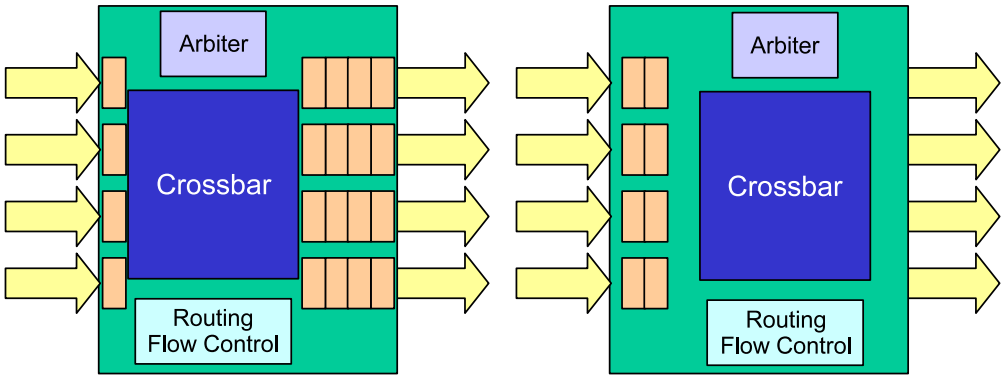
Two different clock signals can be attached to ×pipes NIs: one to drive the NI front-end (OCP interface), the other to drive the NI back-end (×pipes interface). The back-end clock must, however, have a frequency which is an integer multiple of that of the front-end clock. This arrangement allows the NoC to run at a fast clock even though some or all of the attached IP cores are slower, which is crucial to keep transaction latency low. Since each IP core can run at a different divider of the ×pipes frequency, mixed-clock platforms are possible. The constraint on integer divider ratios is instrumental in reducing the implementation cost of this facility down to almost zero.

4.3.2 Switches

The backbone of the NoC is composed of switches (Figure 4.3 on the facing page), whose main function is to route packets from sources to destinations. Switches can be fully parameterized in the number of input and, independently, of output ports. Arbitrary switch connectivity is possible, allowing for the implementation of any topology. The flit width can also be arbitrarily set.

Full connectivity among the input and output ports is provided in a central crossbar. An arbiter is attached to each output port to resolve conflicts among packets when they overlap in requesting access to the same output links. The arbiter can be configured for round-robin or fixed-priority policies. Multiple links can be deployed among the same pair of switches, offering an inexpensive solution to localized bandwidth and congestion issues. Since ×pipes performs source routing, the switch does not include routing LUTs.

Switches provide buffering resources to lower congestion and improve performance. As mentioned above, buffering resources are instantiated also depending on the desired flow control protocol (see Section 4.4 on the next page). If a retransmission-based flow control protocol is chosen, then relatively deep output buffers must be provided, so as to hold previously sent packets for some cycles. In this scenario, input buffering is not really required (although it can optionally be deployed) - the switch inputs only need being registered with a single flip-flop per data wire. The traversal latency is 2 clock cycles. On the contrary, when credit-based flow control is chosen, only input buffering is mandatory. In this scenario, ×pipes optionally allows the designer to do completely without output buffers,



(a) ACK/NACK case (inputs and outputs are registered).

(b) STALL/GO case (inputs are registered, outputs are optionally registered).

Figure 4.3: ×pipes switch block diagram. The STALL/GO variant can optionally feature output buffering.

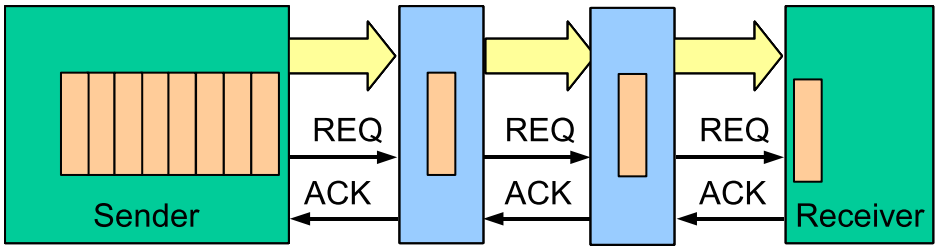
reducing the traversal latency of a switch to a single clock cycle. Output buffers can still be deployed to decouple the propagation delays within the switch and along the downstream link; the downside is a second cycle of latency and additional area and power overhead.

4.3.3 Links

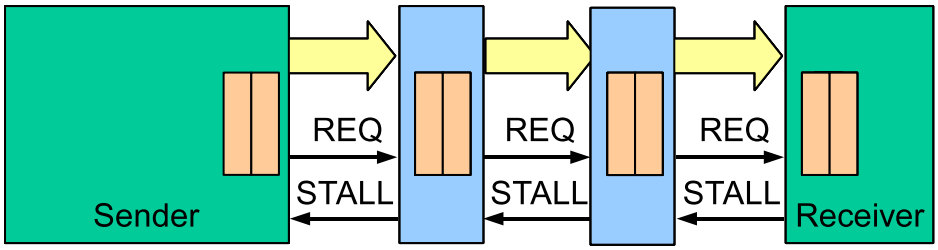
Inter-block links are a critical component of NoCs, given the technology trends for global wires [3]. The problem of signal propagation delay is, or will soon become, critical. For this reason, ×pipes supports link pipelining [204] (Figure 4.4 on the following page), *i.e.* the interleaving of logical buffers along links. The link data introduction rate is then decoupled from link delay by trading it with latency. Proper flow control protocols are implemented in link transmitters and receivers (NIs and switches) to make the link latency transparent to the surrounding logic (latency insensitive operation) [205]. Therefore, the overall platform can run at a fast clock frequency, without the longest wires being a global speed limiter.

4.4 NoC Flow Control Protocols

Flow control determines how network resources are allocated to packets traversing the network, and can be seen either as a problem of resource



(a) ACK/NACK case.



(b) STALL/GO case.

Figure 4.4: ×pipes pipelined link block diagram. The STALL/GO variant can optionally do without the output buffer in the sender (switch or NI).

allocation or one of contention resolution [42]. Flow control in NoCs is crucial, as it is playing a decisive role in the determination of:

- The amount of resources, especially buffers, that have to be deployed in the system. Efficient flow control protocols will minimize the number of required buffers and their idling time.
- The latency that packets incur while traversing the network. While bandwidth is exceedingly available in NoCs, a lot of effort needs to be devoted to cutting down on network latency, else the applications' performance specifications may be violated. Especially under heavy traffic conditions, fast packet propagation with maximum resource utilization are key.
- The degree of support for link pipelining. Buffer insertion is a well-known technique to address the problem of high-frequency transmission over data wires [206]. However, even with repeaters, wire delay can exceed one clock period and multiple clock cycles will be needed to transfer data from one synchronous block to another. By inserting flip-flops to improve throughput, wire pipelining techniques deal with this concern [207]. However, not all flow control protocols can support this technique transparently or efficiently.

In circuit-switched NoCs providing QoS guarantees, minimum buffering flow control can be used: a circuit is formed from source to destination nodes by means of resource reservation, over which data propagation occurs in a contention-free regime. However, circuit-switched approaches are normally only used for some specific traffic flows in the network, or for limited periods of time, as otherwise they typically would impose worst-case design principles. Best-effort networks are normally purely packet-switched, and typically buffering increases the efficiency of flow control mechanisms. The amount of buffering resources in the network depends on the target performance and on the implemented switching technique. Switches need to hold entire packets when store-and-forward or virtual-cut-through switching are chosen, but only flits when wormhole switching is used.

In some proposed NoC architectures, flow control is combined with error control in a unified mechanism. Error control is becoming a growing concern as technology scales toward deep submicron, because of the increased impact on signal reliability of noise sources such as crosstalk, power-supply noise, ELECTROMAGNETIC INTERFERENCE (EMI) and soft errors. Corrupted flits can be detected either in hardware by means of error correction or error detection/retransmission mechanisms, or can be handled at higher network layers (*e.g.*, connection oriented transport layer). However, fast error recovery requires a hardware implementation of error control, thus increasing switch and/or NI complexity. The re-use of flow control mechanisms for error handling allows to save some area and power and to avoid duplication of control lines.

We consider, and implement in ×pipes, three alternative schemes for buffer and channel bandwidth allocation in presence of pipelined switch-to-switch links. These protocols provide varying degrees of fault tolerance support, resulting in different area and power tradeoffs. Our analysis is aimed at determining the overhead of such support. However, as error events are not expected to be frequent, we expect that the normal operating mode would be error-free. Therefore, we are mostly interested in pointing out whether the overhead for implementing combined flow and error control in hardware is such to degrade application perceived performance in this normal operating mode. The concern is that enhanced flow control schemes may risk to provide for efficient recovery from infrequent errors at the cost of a significant performance penalty in the typical fault-free regime.

Two of the most usual flow control protocols involve switch-to-switch communication and are retransmission-based (whereby packets are optimistically sent but a copy of them is also stored by the sender, and, if the

	ACK/NACK	STALL/GO	T-Error
Buffer area	$3N + k$	$2N$ or $2N + 2$	$> 3M + 2$
Logic area	medium	low	high
Performance	depends	good	good
Power (est.)	high	low	medium/high
Fault tolerance	supported	unavailable	partial

Table 4.1: Flow control protocols at a glance.

receiver is busy, a feedback wire to request retransmission is raised) or credit-based (whereby the receiver constantly informs the sender about its ability to accept data, and data is only sent when resources are certainly available). End-to-end flow control schemes [21], where peripheral NIs directly exchange flow control information with each other, are more rarely used because of their buffering requirements; the most common usage scenario involves NoCs that implement circuit-switching [19].

In ×pipes, three radically different flow control protocols have been implemented.

- ACK/NACK, a retransmission-based protocol supporting increased error resilience if paired with error detection logic.
- STALL/GO, a simple variant of credit-based flow control allowing for pipelined links to be transparently deployed.
- T-Error, a timing-error-tolerant flow control scheme, whose capabilities can either be used to negate timing-related errors, to overclock the links, or to extend their maximum length.

Each of these offers different fault tolerance features at different performance/power/area points, as sketched in Table 4.1. STALL/GO is a low-overhead scheme which assumes reliable flit delivery. T-Error is much more complex, and provides logic to detect timing errors in data transmission; this support is however only partial, and usually exploited to improve performance rather than to add reliability. Finally, ACK/NACK is designed to support thorough fault detection and handling by means of retransmissions.

We implemented each of these flow control protocols to support links having a variable physical length. Clock frequency was kept invaried by pipelining the links with repeater stages, trading off latency for clock speed.

4.4.1 Retransmission-Based Flow Control Protocol

The main idea behind the ACK/NACK flow control protocol (Figure 4.4(a) on page 102) is that transmission errors may happen during a transaction. For this reason, while flits are sent on a link, a copy is kept locally in a buffer at the sender. When flits are received, either an ACKNOWLEDGE (ACK) or NOT ACKNOWLEDGE (NACK) is sent back. Upon receipt of an ACK, the sender deletes the local copy of the flit; upon receipt of a NACK, the sender rewinds its output queue and starts resending flits starting from the corrupted one, with a Go-Back-N policy. This means that any other flit possibly in flight in the time window among the sending of the corrupted flit and its resending will be discarded and resent. Other retransmission policies are feasible, but they exhibit higher logic complexity. ACK/NACK can either be implemented as end-to-end over a whole fabric, or as switch-to-switch; due to complex issues with possible flit misrouting upon faults in packet headers, we implemented the latter. Fault tolerance is built in by design, provided encoders and decoders for error control codes are implemented at the source and destination respectively. We do not focus on such decoders here and refer the reader to the vast amount of literature is available in the field of coding.

In an ACK/NACK flow control, a sustained throughput of one flit per cycle can be achieved, provided enough buffering. Repeaters on the link can be simple registers, while, with N repeaters, $2N + k$ buffers are required at the source to guarantee maximum throughput, since ACK/-NACK feedback at the sender is only sampled after a round-trip delay since the original flit injection. The value of k depends on the latency of the logic at the sending and receiving ends. Overall, the minimum buffer requirement to avoid incurring bandwidth penalties in a NACK-free environment is therefore $3N + k$.

ACK/NACK provides ideal throughput and latency until no NACKs are issued. If NACKs were only due to sporadic errors, the impact on performance would be negligible. However, if NACKs have to be issued also upon congestion events, the round-trip delay in the notification causes a performance hit which is very noticeable especially with long pipelined links. This will be investigated in Section 4.4.4 on page 108. Moreover, flit bouncing between senders and receivers causes a waste of power.

In ACK/NACK, since output buffers need to be deployed to account for potential retransmissions, NoC links are always enclosed between two clocked buffers at the sending and receiving ends. Hence, a whole clock period is available for signal propagation along the wires of the inter-switch links. Therefore, the link length and the switch logic are decoupled

by the output buffer.

4.4.2 Credit-Based Flow Control Protocol

STALL/GO is a very simple realization of an ON/OFF flow control protocol (Figure 4.4(b) on page 102). It requires just two control wires: one going forward and flagging data availability, and one going backward and signaling either a condition of buffers full ("STALL") or of buffers free ("GO"). STALL/GO can be implemented with distributed buffering along the link; namely, every repeater can be designed as a very simple two-stage FIFO. The sender can do completely without output buffering or can deploy two buffers to cope with stalls in the very first link repeater, thus resulting in an overall buffer requirement of $2N$ or $2N + 2$ registers, with minimal control logic. Power is minimized since any congestion issue simply results in no unneeded transitions over the data wires. Performance is also good, since the maximum sustained throughput in absence of congestion is of one flit per cycle by design, and recovery from congestion is instantaneous (stalled flits get queued along the link towards the receiver, ready for flow resumption).

In the NoC domain with pipelined links, STALL/GO indirectly reflects the performance of credit-based policies, since they exhibit equivalent behaviour. The main drawback of STALL/GO is that no provision whatsoever is available for fault handling. Should any flit get corrupted, some complex higher-level protocol must be triggered.

In STALL/GO, output buffers can be optionally skipped. This means that, contrary to ACK/NACK, the switch logic and the link propagation time (up to the following switch or to the first link pipeline stage) can be contributing to the same timing path, which becomes the bottleneck for the system. While ACK/NACK transparently allows for links of arbitrary propagation time, possibly just requiring the insertion of pipeline stages, with STALL/GO (at least in its leanest embodiment) the link delay directly impacts the maximum operating frequency of the switches and of the whole NoC.

4.4.3 Timing-Error-Tolerant Flow Control Protocol

The T-Error [49] protocol (Figure 4.5 on the next page) aggressively deals with communication over physical links, either stretching the distance among repeaters or increasing the operating frequency with respect to a conventional design. As a result, timing errors become likely on the link.

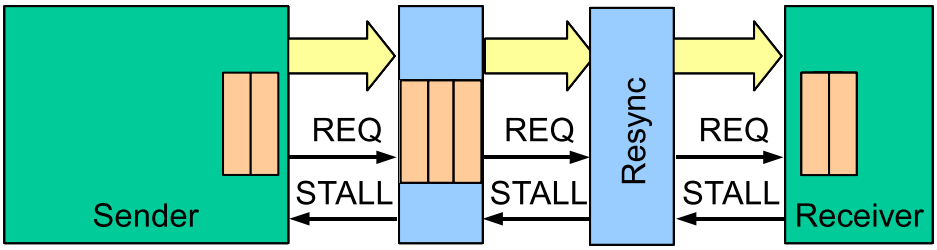


Figure 4.5: T-Error protocol implementation.

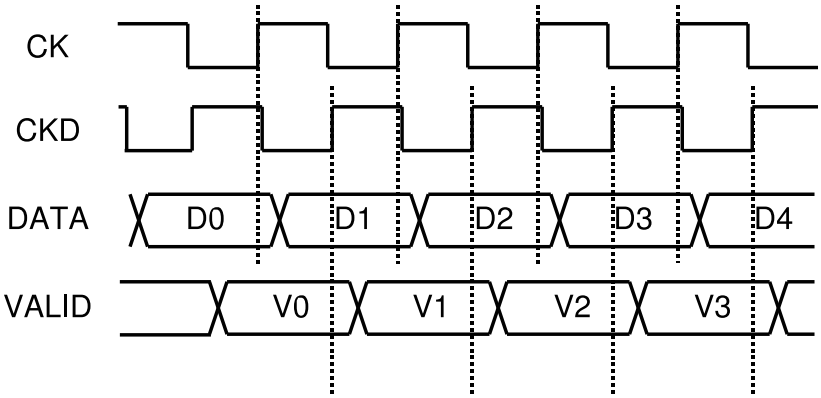


Figure 4.6: T-Error concept waveforms.

Faults are handled by a repeater architecture leveraging upon a second delayed clock to resample input data, to detect any inconsistency and to emit a VALID control signal (Figure 4.6). If the surrounding logic is to be kept unchanged, as we assume here, a resynchronization stage must be added between the end of the link and the receiving switch. This logic handles the offset among the original and the delayed clocks, thus realigning the timing of DATA and VALID wires; this incurs a 1-cycle latency penalty. Given this resynchronization stage, T-Error becomes pin-compatible with STALL/GO, and can be deployed in a NoC with STALL/GO switches and NIs.

The timing budget provided by the T-Error architecture can, alternatively, be exploited to achieve greater system reliability, by configuring the links with spacing and frequency as conservative as in traditional protocols. However, T-Error lacks a really thorough fault handling: for example, errors with large time constants would not be detected. Mission-critical systems, or systems in noisy environments, may need to rely on higher-level fault correction protocols.

The area requirements of T-Error include three buffers in each repeater

and two at the sender, plus the receiver device and quite a bit of overhead in control logic. A conservative estimate of the resulting area is $3M + 2$, with M being up to 50% lower than N if T-Error features are used to stretch the link spacing. Unnecessary flit retransmissions upon congestion are avoided, but a power overhead is still present due to the control logic. Performance is of course dependent on the amount of self-induced errors and will be analyzed in detail in Section 4.4.4.

4.4.4 Experiments on Alternative Flow Controls

Fault tolerance is an ever more important feature as deep submicron lithographic processes get deployed, to counter increasingly prevalent noise sources. In the following, the assumption is made that with a conservatively clocked design, errors can be made rare enough to have a negligible performance impact. For this reason, during benchmarking, we assume an environment free from external faults and we instead explore performance during normal operation.

This holds also for T-Error. When used to increase system reliability, by deploying it with conservative link parameters, we simulate fault-free communication. When used to aggressively space link repeaters, thus artificially causing and handling a non-trivial amount of data corruption in exchange for better performance, we instead inject varying amounts of random transmission errors. These however attempt to reproduce self-induced corruption only, and not external faults.

We choose as a testbench a star-like topology, where up to eight clusters of three processors and their private memories can be instantiated. At the heart of each cluster is a 7x7 ×pipes switch. Therefore, up to 24 processors can be deployed in the platform. Shared slaves exist and are attached to a central 11x11 switch (see Figure 4.7 on the facing page). The central switch is connected to the computation clusters by means of ×pipes pipelined links, whose length can be customized. Depending on the amount of instantiated processors, the simultaneous traffic on the links towards this central switch increases, resulting in congestion. All of the processors are executing the same benchmark, which encompasses local computation (which happens within the peripheral clusters) and performs communication and synchronization functions by accessing the shared slaves on the central switch.

An analysis of the link congestion trend under increasing pressure by IP cores can be found in Figure 4.8 on page 110, which plots the density of ACKs and NACKs (ACKs and NACKs over clock cycles) when using ACK/NACK flow control in the system. The chart is only plotting figures

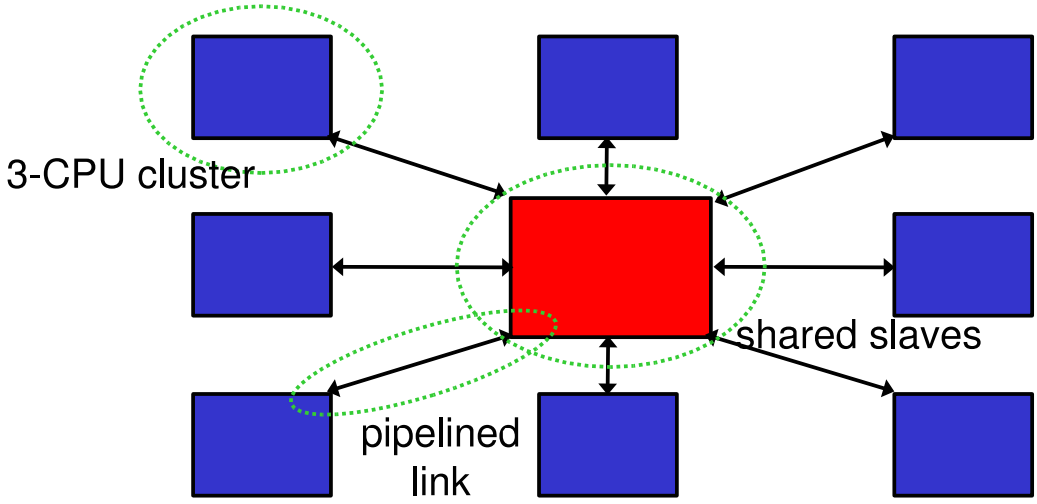


Figure 4.7: The star-like topology under test.

for the links connecting the central switch to the clusters; the links are here assumed to be very long, with six repeaters. As can be seen, with more processors, initially the ACK density increases thanks to the increase in offered bandwidth. Just before hitting the value 0.5 (one ACK every two cycles), growing congestion and the intrinsic inefficiency of this flow control protocol impose a bandwidth ceiling, while a growing amount of NACKs can be observed.

To evaluate the efficiency of the three flow control protocols, we measure the average latency for communication transactions across the congested links and the overall benchmark execution time. Please note that clock frequency and physical layout were assumed to be the same for all schemes. Results are plotted as a function of the length of the pipelined links in Figure 4.9 on page 111. For T-Error, the same simulations are carried out in two scenarios:

- The first, **aggressive**, accounts for 50% less repeater stages (four instead of six, two instead of three) and assumes a 5% error rate.
- The second scenario is more **conservative**, with as many repeaters as in other protocols and no errors.

In both cases, a resynchronization stage is needed before the receiving switch. If a link is short enough to allow for operation in a single clock cycle, T-Error logic is unneeded and this case reduces to the direct STALL/GO connection.

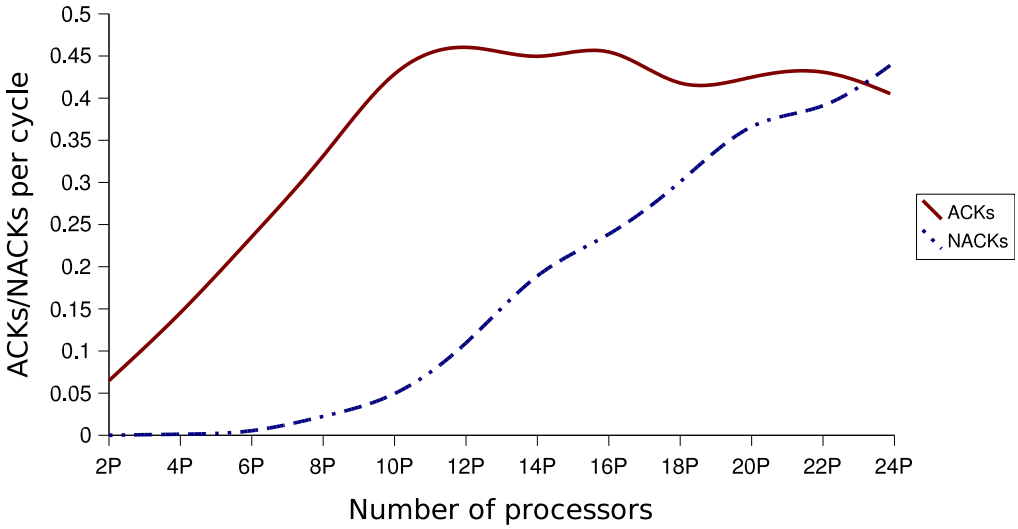
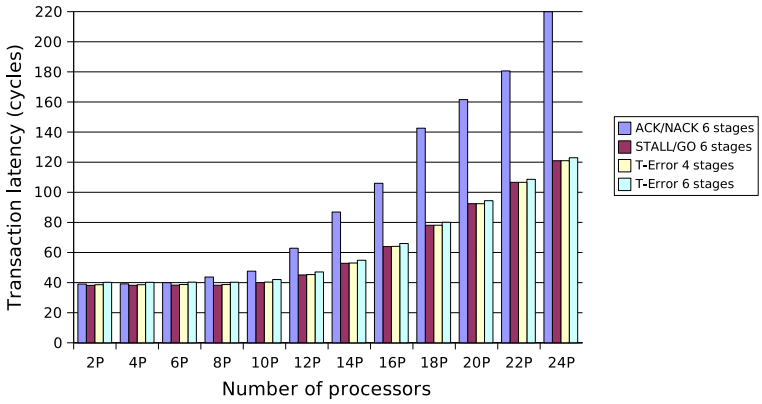


Figure 4.8: Link congestion under increasing traffic.

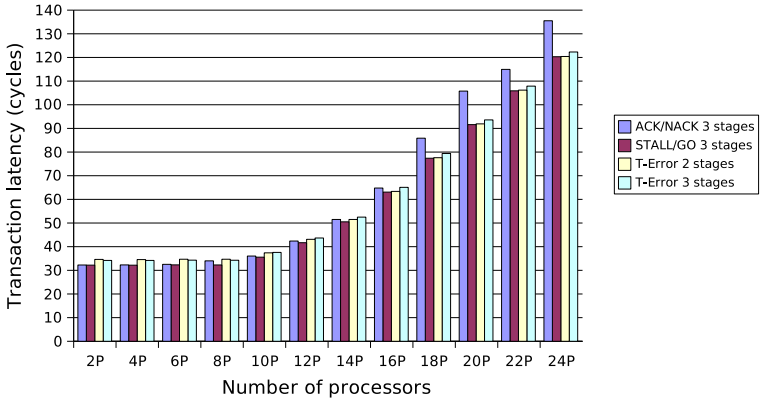
Under all circumstances, both variants of T-Error and STALL/GO exhibit similar performance. The latency advantage gained over STALL/GO by the **aggressive** deployment of T-Error is mostly offset by the need for a resynchronization stage and by some penalty upon transmission errors; for short links and in low congestion environments, T-Error can even perform worse. The **conservative** T-Error links perform almost on par with the **aggressive** ones because they trade repeater stages for error-free operation. The **conservative** T-Error links always perform slightly worse than the corresponding STALL/GO schemes due to resynchronizer latency, but the transaction overhead is negligible. As expected, if links are very long (six repeaters: Figure 4.9(a) on the next page), the round trip delay imposed by the ACK/NACK protocol proves to be a major drawback, and latencies increase steeply with congestion. With shorter links (three repeaters: Figure 4.9(b) on the facing page), the ACK/NACK overhead decreases, and substantial performance parity is achieved if the switches are directly attached (Figure 4.9(c) on the next page).

In Figure 4.10 on page 112, the overall benchmark execution time is reported just for the longest link case. The trend is of course similar to that of the communication latency over the congested links, but differences are smaller because they are masked by the time spent in local computation.

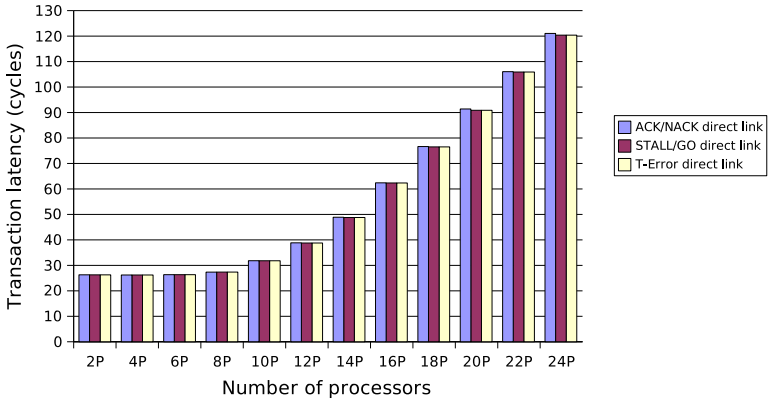
The **aggressive** T-Error variant achieves lower communication laten-



(a) Six repeaters.



(b) Three repeaters.



(c) Direct connection.

Figure 4.9: Communication latency over congested links of different lengths.

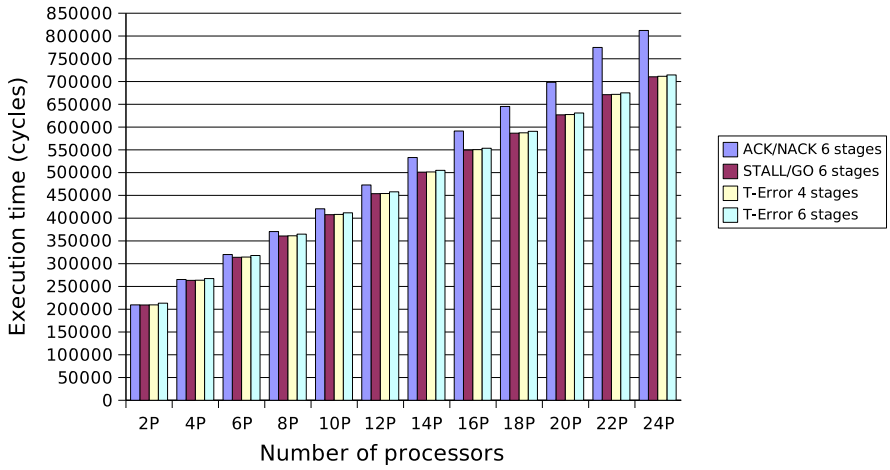


Figure 4.10: Benchmark runtime under increasing congestion, long links.

cies by accepting a certain amount of transmission errors as a tradeoff. Determining this amount is not a focus of this work. So, in Figure 4.11 on the next page we explored the design space by assuming different self-induced error probabilities. The plot is reporting figures for a long link, which is assumed to have a 0% to 27% error rate percentage. This number expresses the percentage of errors per clock cycle (not per transmitted flit), and is for the whole link. Latencies are normalized against the ideal error-free case. As the plot shows, under heavy congestion, T-Error is by design able to almost completely mask errors, because error penalties can be hidden behind congestion-triggered stalls. Under light traffic, transmission faults have a more noticeable impact, with 6% worse transmission latency when comparing a 27% link error probability against the ideal case. Still, T-Error is very good at minimizing the impact of faults on performance.

As expected (Table 4.1 on page 104), STALL/GO proves to be a low-overhead efficient design choice showing remarkable performance, but unfortunately is fault-sensitive. T-Error can be either deployed to improve link performance, or to improve system reliability by catching timing errors. In the former design, we observed average latencies on par with STALL/GO, but no error detection capability was present; in the latter case, speed degraded slightly, in exchange for a partial but significant reliability boost. In both alternative schemes, some area and power overhead is incurred. Overall, we believe that using T-Error to decrease the number of pipeline stages does not bring significant performance benefits, while the partial detection capability can be effectively exploited in a conservative design. Another possible option is the conversion of the timing

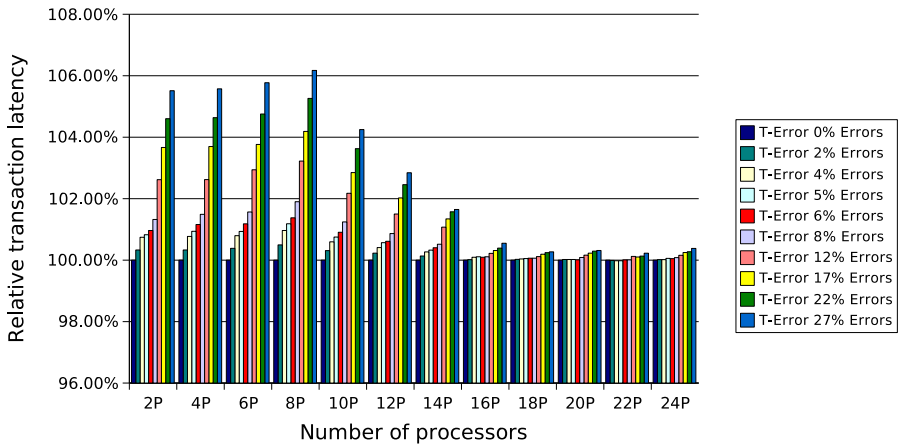


Figure 4.11: T-Error performance under varying error probabilities.

margin budget into a frequency overclock; while this choice holds good potential, we did not explore it yet since it is only feasible if the surrounding NoC components (switches, network interfaces) are designed to work at the same extremely high frequencies. ACK/NACK pays its most extensive fault handling support with significant power and area overheads - especially considering that, on top of the reported results, error detection circuitry would also need to be instantied. Performance penalties were also noticed in presence of heavy congestion and long pipelined links. However, we expect that, in current and imminent design technologies, links will need no more than three repeaters, the very long link scenario being representative of a more distant future. In low-congestion or short-link scenarios, the application-perceived latency overhead of ACK/NACK turned out to be negligible.

4.5 Conclusions

We have presented ×pipes, a synthesizeable library of NoC components. ×pipes design is driven by several criteria, including high customizeability, low latency, low area occupation, low power dissipation. ×pipes components can be freely composed into arbitrary topologies, and they support IP cores with the standard OCP pinout. ×pipes allows for the instantiation of highly customized and highly efficient NoCs. Further, the ×pipes library includes extensive support for wiring management, including link

pipelining if needed.

Subsequently, we have presented studies on three variations of a key design variable of the NoC - the choice of the flow control. We picked three representative flow control schemes, targeted at different fault tolerance/performance ratios, and compared them in terms of latency and buffer requirements. The results confirm that very efficient flow controls exist (STALL/GO) for the normal operation case, but that an overhead must be paid to support more extensive fault tolerance, such as in ACK/-NACK. T-Error, which features built-in correction of timing errors, proves to be an interesting solution too, especially to increase the fault tolerance of the system.

Future work will revolve around the development of frequency and data width conversion components for ×pipes, and around the addition of the support for more protocols in the NIs.

CHAPTER 5

NoC Design Flow Front-End: Topology Design

This chapter[1] describes the front-end of the proposed flow (Figure 5.1 on the next page). The goal is the design of the optimal NoC topology and hardware configuration (*e.g.*, switch radix, data width, *etc.*) for a given application. The required inputs are as high-level as possible, so as to streamline the designer's task: (i) a communication graph specifying the communication requirements of the application at hand, (ii) a set of area and power models for NoC components implemented in the target technology (see Section 6.4 on page 172 for more details), and finally (iii) a set of optimization objectives (a linear combination of minimum power and minimum latency goals) and constraints (such as area, power, latency bounds).

5.1 Motivation and Key Challenges

5.1.1 Custom Topology Design and Floorplan Awareness

In order to effectively deploy NoCs in MPSoCs, as discussed in Chapter 1 on page 1, it is crucial to be able to leverage design automation tools. The development of such tools is unfortunately not an easy task, given the huge design space to be explored and the large set of constraints.

[1] Most of the credit for the work described in this chapter goes to Dr. Srinivasan Murali, Prof. Luca Benini and Prof. Giovanni De Micheli.

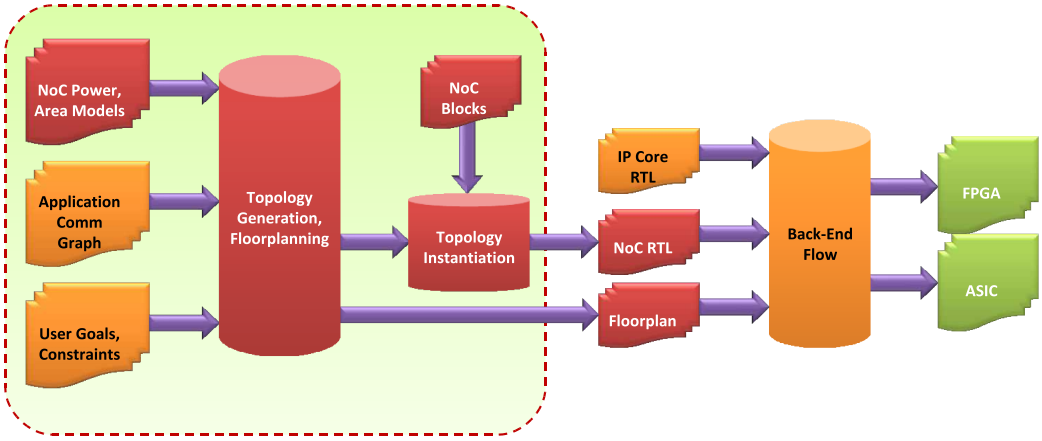


Figure 5.1: The proposed NoC design flow: front-end.

We believe that there are some specific issues that NoC design tools should at all costs tackle in order to instantiate efficient NoCs:

- Several works in the NoC domain (for instance [208]) have been electing to use standard, regular topologies (mostly, meshes) derived from research on macro-networks, under the assumption that the wires can be well structured in such topologies. While some MP-SoCs are indeed homogeneous, especially in the CMP domain, a vast majority are in fact heterogeneous, with each core having different physical size, interface and communication requirements [9, 1, 10]. Regular topologies would handle very poorly these cases, leading to floorplans with large slacks and to power overheads. Therefore, it is imperative to generate cutomized, application-tailored NoCs.
- For many of the same reasons discussed just above, NoC topology design should not be performed separately from floorplanning. It is counterproductive to design a NoC where some cores are clustered around a particular switch if, in the final implementation, those same cores will be positioned at opposite corners of the chip. This would unnecessarily lead to complex wiring and to operating frequency penalties. The design of the NoC should instead be synchronized with the chip floorplanning, either by devising an optimal floorplan together with the generation of the NoC, or by accepting as an input a reference floorplan defined by the designer, and by placing a NoC in the optimal positions.

As a motivating example, the network (switch and link) power consumption, hop count, wire length and design area of two different NoC

Parameter	Mesh	Application-specific
Power (mW)	301.78	79.64
Average hop count	2.58	1.67
Total wire length (mm)	185.72	145.37
Design Area (mm^2)	51.0	47.68

Table 5.1: Topology Comparisons.

topologies for a video processor SoC with 42 cores are presented in Table 5.1. The first topology is a regular mesh, while the second is a custom topology generated using the methodology presented in this dissertation. The wire lengths and design area are obtained from floorplanning of the NoC designs. A more detailed explanation of the topologies and the floorplanning process can be found in Chapter 6 on page 145. The custom topology leads to a 3.8× reduction in network power consumption, a 1.55× reduction in average hop count and a 1.28×reduction in total length of wires when compared to the mesh.

5.1.2 Deadlock Freedom

Another key point that needs to be tackled by a proper NoC topology generation flow is that of deadlocks. Two classes of deadlocks can occur in NoCs: *routing-dependent* deadlocks and *message-dependent* deadlocks [42, 130, 129]. Deadlock freedom must be guaranteed, else the resulting system would be subject to malfunctions.

In NoCs, wormhole switching [42] is usually employed to reduce switch buffering requirements and to provide low-latency communication [209]. With wormhole flow control, deadlocks can happen during the routing of packets due to cyclic dependencies of resources, such as buffers [42]. For regular topologies, such as meshes or tori, the use of restricted routing functions based on turn models is an effective way to avoid routing-dependent deadlocks [126, 125]. For custom application-specific NoCs, obtaining deadlock-free paths is a bigger challenge; only some works have addressed deadlock-free path selection mechanisms for custom NoC designs [129, 132].

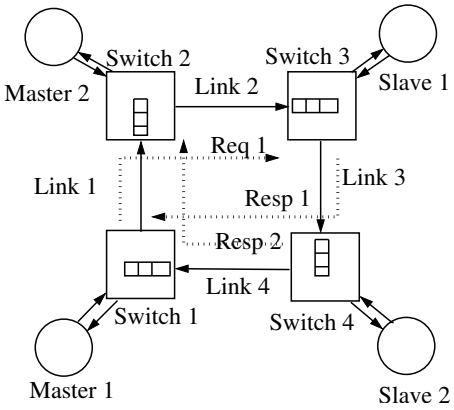
Message-dependent deadlocks occur when interactions and dependencies are created between different message types at network endpoints, when they share resources in the network. Even when the underlying network is designed to be free from routing-dependent deadlocks, message-level deadlocks can block the network indefinitely, thereby af-

fecting proper system operation.

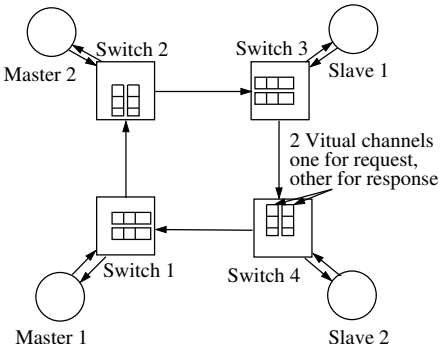
Example 1 *An example situation where a message-dependent deadlock occurs is presented in Figure 5.2(a) on the facing page. In this example, two of the cores are masters and two other cores are slaves. In this system, we assume two kinds of messages, requests and responses. Consider the following situation: Master 1 sends a request to Slave 1 (Req 1), Slave 1 is replying to a previously issued request by Master 1 (Resp 1), while Slave 2 is simultaneously sending a response to Master 2 (Resp 2). Since requests and responses share the same links, Resp 2 is waiting for link 1 which is used by Req 1, and Resp 1 waits for link 4 used by Resp 2. Meanwhile, Req1 is waiting for Slave 1, the operation of which has been stalled as Resp 1 could not complete. Thus, none of the messages can move ahead, leading to a deadlock situation. An interesting point to note here is that message-level deadlocks can be avoided if the receivers have infinitely large buffering or if they have perfectly ideal operation (consuming all received data instantly), which would avoid queuing of the packets in the network. Obviously, this is not feasible in practice.*

In traditional multi-processor interconnection networks, the most common ways to avoid message-dependent deadlocks are the use of separate logical or physical networks for the different message types [128, 134, 135, 136, 210, 211, 5, 139]. This would ensure that the different message types do not share the network components, thereby guaranteeing freedom from message-dependent deadlocks. The most common method to achieve separate logical networks is the of use of separate virtual channels for the different message types [128]. For the example design presented in Figure 5.2(a) on the next page, each router input will need two virtual channels: one for the request messages and the other for the response messages (Figure 5.2(b) on the facing page). This separation of message types is maintained at all the switches in the network. In the case of separate physical networks, the request network is built separately from the response network, an example of which is shown in Figure 5.2(c) on the next page. This is the most commonly used solution in complex bus designs, such as STBus [5] and several multi-processor designs [138, 139].

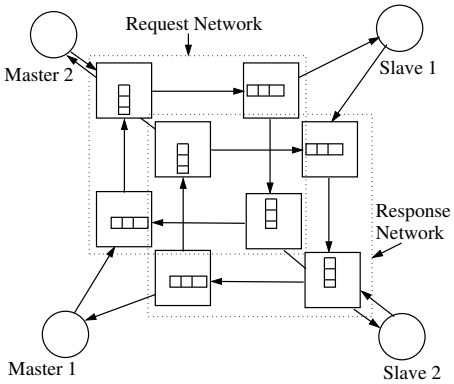
Methods that can lead to deadlock-free operation with minimum power and area overhead are important for designing application-specific NoCs. We believe that, by considering this issue during topology generation, we can obtain a significantly better NoC design than traditional methods, where the deadlock avoidance issue is dealt with separately.



(a) Example of message-dependent deadlock.



(b) Logically separate networks.



(c) Physically separate networks.

Figure 5.2: Deadlocks and deadlock-free architectures.

5.1.3 Target Operating Frequency

A further item to be accounted for is the fact that, to comply with performance constraints, certain operating frequency goals must be met. Topology design tools must make sure that all portions of the NoC can fulfill these goals. This task includes picking appropriate components from the NoC library (for example, high-radix switches will support a lower frequency than smaller switches). This task also requires making sure that NoC links can be traversed within a clock cycle, which may be particularly hard in large designs. As discussed in Chapter 4 on page 95 and Section 6.5.2 on page 192, the easiest workaround is simply to deploy pipeline stages along the affected links, increasing their latency but not impacting the NoC frequency.

5.1.4 Proposed Solution

We present a design tool, called SunFloor, that automates the generation of such customized NoC architectures, satisfying the communication constraints of the target application. We present a floorplan-aware design method that considers the wiring complexity of the NoC already during the topology design process. This leads to detecting timing violations on the NoC links early in the design cycle and to have accurate power estimations of the interconnect. We generate NoC instances where topology, architectural parameters (*e.g.* data width) and operating parameters (*e.g.* operating frequency) are all automatically tuned for optimal results; the designer can manually override some of these if desired. We incorporate mechanisms to prevent deadlocks during routing, which is critical for proper operation of NoCs. This methodology is then integrated into a NoC synthesis flow (Chapter 6 on page 145), automating NoC synthesis, generation, simulation and physical design processes. We also present ways to ensure design convergence across the abstraction levels. Our flow guarantees deadlock freedom and is aware of link pipelining needs; based on floorplanning analyses, it is able to automatically instantiate pipeline repeaters where needed, while accounting for their latency cost.

5.2

Required Input Models

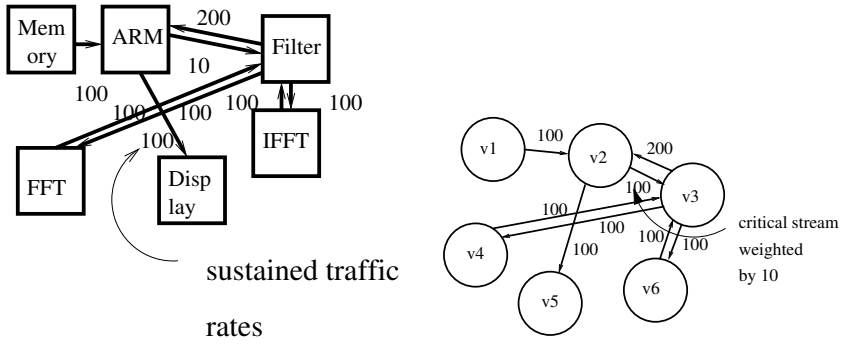
To operate correctly, SunFloor requires a set of inputs. The first is a communication graph [208, 116, 118], describing the traffic requirements among any pair of cores in the system. This information is typically well known by the designer, based on application properties, previous design experience, IP core datasheets, or initial simulations or estimations. The sustained rate of communication between the cores is obtained based on the average, peak rates and the latency constraints of the flows, as presented in [118]. It is shown there that the network has to satisfy the sustained rate of the traffic flows to satisfy the application design constraints. Whether a traffic stream is critical or not is also obtained from the application characteristics. From these values, we construct the core graph for the application:

Definition 1 *The communication graph is a directed graph, $G(V, E)$ with each vertex $v_i \in V$ representing a core and the directed edge (v_i, v_j), denoted as $e_{i,j} \in E$, representing the communication between the cores v_i and v_j. The weight of the edge $e_{i,j}$, denoted by $comm_{i,j}$, represents the sustained rate of traffic flow from v_i to v_j weighted by the criticality of the communication. The set F represents the set of all traffic flows, with value of each flow, f_k, $\forall k \in 1 \cdots |F|$, representing the sustained rate of flow between the source (s_k) and destination (d_k) vertices of the flow.*

The communication graph for a small filter example (Figure 5.3(a) on the next page) is shown in Figure 5.3(b) on the following page. The edges of the communication graph are annotated with the sustained rate of traffic flow, multiplied by the criticality level of the flow.

SunFloor also requires accurate analytical models for the power consumption and area of the network components, in order to be able to assess the best design points. We derived such models for the ×pipes architecture (Chapter 4 on page 95 and Section 6.4 on page 172). Power consumption values were obtained from layouts with back-annotated resistance, capacitance information, and based upon injection of functional traffic, which translates into realistic switching activities in the components.

An example of the required input models is shown in Table 5.2 on the next page, where we report results for the layout-level characterization of some components implemented in 130nm technology. Due to the intrinsic modularity and symmetry of NoC switches, the analytical models built based on a training set of instances can be very close to the actual area and power results, with typical errors below 10% Section 6.4.6 on page 184. For



(a) Filter application.

(b) Communication graph for a filter application, with sustained rates and critical streams.

Figure 5.3: Filter application.

Component	Parameter	Analytical	Experimental
4x4 switch	area(mm^2)	0.036	0.035
	power(mW)	22.16	22.54
	max frequency (MHz)	900	897
5x5 switch	area(mm^2)	0.048	0.047
	power(mW)	28.38	28.70
	max frequency (MHz)	880	885
link (2mm)	power(mW)	0.57	0.57

Table 5.2: Area and power models for some NoC components in 130nm technology, running at 900 MHz, featuring a data width of 32 bits, and a switch buffer depth of three registers.

the purposes of topology design, modeling the power consumption of the NIs is not crucial. In fact, the power cost of the NIs is basically unaffected by the chosen topology, and can even be assumed to be part of the power budget of the IP cores.

SunFloor can operate in two modes. In the first, it generates a system floorplan as an output, together with the optimal NoC topology. Alternatively, a floorplan can be taken as an input; this is useful for best integration in existing design flows, and it allows designers to override any placement decision whenever needed (*e.g.*, because an IP core needs to be close to the chip edge because it drives some output pins). In this second mode of operation, SunFloor takes as an input a very simple high-level description of the floorplan, where IP cores are characterized with a size and a position.

The final input required by SunFloor is a set of objectives and constraints. Our topology design process supports two objective functions: minimizing network power consumption and minimizing the hop count (zero-load latency) for data transfer. The designer can optimize for one of the two objectives or a linear combination of both. The topology design process further supports constraints on several parameters, such as the hop count (when the objective is power minimization), the network power consumption (when the objective is hop count minimization), the design area and the total wire length. All these choices can be simply passed to SunFloor as command line parameters.

5.3 Topology Design Algorithm

The topology generation process sweeps multiple design points, varying architectural parameters (data width, operating frequency) within a range. Several topologies with different numbers of switches are explored, starting from a topology where all the cores are connected to one switch, to one where each core is connected to a separate switch. The generation of a topology includes finding the radix of the switches, establishing the connectivity between the switches and connectivity with the cores, and finding deadlock-free routes for the different traffic flows. To have an accurate assessment of the design, the floorplanning of each topology is automatically performed, fixing the position of the IP cores and NoC components on the chip surface. Based on the frequency point and the obtained wire lengths, the timing violations on the wires are detected and the power con-

sumption on the links is obtained. From the set of all generated topologies, the design point that best optimizes the user's objectives, satisfying all the design constraints, is chosen.

In the first step of Algorithm 1 on the next page, a design point θ is chosen from the set of available or interesting design points ϕ for the NoC architectural parameters. In our current implementation, the engine automatically tunes two critical NoC parameters: operating frequency ($freq_\theta$) and link width (lw_θ). As both frequency and link width parameters can take a large set of values, considering all possible combinations of values would be infeasible to explore. The system designer has to trim down the exploration space and give the interesting design points for the parameters. The designer usually has knowledge of the range of these parameters. As an example, the designer can choose the set of possible frequencies from a minimum to a maximum value, with allowed frequency step sizes. Similarly, the link data widths can be set to multiples of 2, within a range (say from 16 bits to 128 bits). Thus, we get a discrete set of design points for ϕ, as done in [113]. In the experiments shown in Section 5.5.1 on page 133, we support 8 frequency steps and 4 link width steps, providing 32 discrete design points in the set ϕ. The rest of the topology design process (steps 2-14 in Algorithm 1 on the next page) is repeated for each design point in ϕ.

As the topology synthesis and mapping problem is NP-hard [121], we present efficient heuristics to synthesize the best topology for the design. For each design point θ, the algorithm synthesizes topologies with different numbers of switches, starting from a design where all the cores are connected through one big switch (basically, a crossbar configuration) until the design point where there is a switch per IP core. The reason for synthesizing these many topologies is that it cannot be predicted beforehand whether a design with fewer, bigger switches would be more power efficient than a design with more, smaller switches, or vice versa. A larger switch features a higher power consumption than a smaller switch to support the same traffic, due to its bigger crossbar and arbiter. On the other hand, in a design with many smaller switches, the packets may need to travel more hops to reach the destination. Thus, the total switching activity would be higher than in a design with fewer hops, which can lead to higher power consumption.

For the chosen switch count i, the input core graph is partitioned into i min-cut partitions (step 3). The partitioning is done in such a way that the edges of the graph that are left across the partitions have lower weights than the edges within partitions (refer to Figure 5.4(a) on page 126) and the number of vertices assigned to each partition is almost the same. Thus,

Algorithm 1 Topology Design Algorithm.

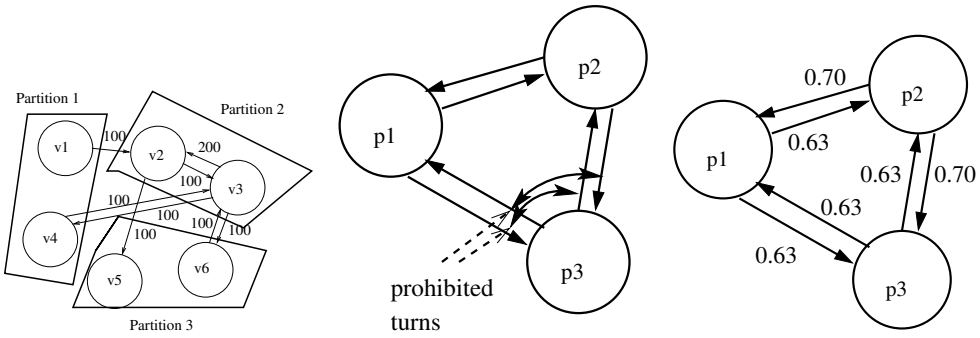
1: Choose design point θ from ϕ: $freq_\theta$, lw_θ
2: **for** i = 1 to $|V|$ **do**
3: Find i min-cut partitions of the core graph
4: Establish a switch with N_j inputs and outputs for each partition, $\forall j \in 1 \cdots i$. N_j is the number of vertices (cores) in partition i. Check for bandwidth constraint violations
5: Build SWITCH COST GRAPH (SCG) with edge weights set to 0
6: Build PROHIBITED TURN SET (PTS) for SCG to avoid deadlocks
7: Set ρ to 0
8: Find paths for flows across the switches using function *PATH_COMPUTE(i, SCG, ρ, PTS, θ)*
9: Evaluate the switch power consumption and average hop count based on the selected paths
10: Repeat steps 8 and 9 by increasing ρ value in steps, until the hop count constraints are satisfied or until ρ reaches ρ_{thresh}
11: If ρ_{thresh} reached and hop count not satisfied, go to step 2
12: Perform floorplan and obtain area, wire lengths. Check for timing violations and evaluate power consumption on wires
13: If target frequency matches or exceeds $freq_\theta$, and satisfies all constraints, note the design point
14: **end for**
15: Repeat steps 2-14 for each design point available in θ

those traffic flows with large bandwidth requirements or higher criticality level are assigned to the same partition and hence traverse only one switch for communication. Therefore, the power consumption and the hop count for such flows will be lower than for the other flows that cross the partitions. For partitioning, we use Chaco, an efficient external hierarchical graph partitioning tool [212].

At this point, the communication traffic flows within a partition have been resolved. In steps 5-9, the connections between the switches are established to support the traffic flows across the partitions. In step 5, the SWITCH COST GRAPH (SCG) is generated.

Definition 2 *The SCG is a fully connected graph with i vertices, where i is the number of partitions (or switches) in the current topology.*

Please note that the SCG does not imply the actual physical connectivity between the different switches. The actual physical connectivity be-



(a) Min-cut partitions. (b) Switch Cost Graph. (c) Path selection.

Figure 5.4: Algorithm examples.

tween the switches is established using the SCG in the PATH_COMPUTE procedure, which is explained in the following paragraphs.

To prevent routing deadlocks (message-dependent deadlocks will be tackled in a few paragraphs), we pre-process the SCG and prohibit certain turns, to break any cyclic dependencies. This guarantees that deadlocks will not occur when routing packets. In order to find the set of turns that need to be prohibited to break cycles, we use the turn prohibition algorithm presented in [129, 117]. The algorithm has polynomial time complexity (very fast in practice, see Section 5.5.1 on page 133) and guarantees that at most one third of the total number of turns would be prohibited to remove cycles. The algorithm also guarantees connectivity between all nodes in the SCG after prohibiting the turns. From the algorithm, we build the PROHIBITED TURN SET (PTS) for the SCG, which represents the set of turns that are prohibited in the graph. To guarantee deadlock freedom, no path for routing packets should take any prohibited turn. These concepts are illustrated in the following example:

Example 2 *The min-cut partitions of the core graph of the filter example (from Figure 5.3(a) on page 122) for three partitions are shown in Figure 5.4(a). The SCG for the partitions is shown in Figure 5.4(b). After applying the turn prohibition algorithm from [129], the set of prohibited turns is identified. In Figure 5.4(b), the prohibited turns are indicated by circular arcs in the SCG. For this example, both turns around the vertex P3 are prohibited to break cycles. So no path that uses the switch P3 as an intermediate hop can be used for routing packets.*

The actual physical connections between the switches are established in step 8 of Algorithm 1 on the preceding page using the PATH_COMPUTE

Algorithm 2 PATH_COMPUTE(i, SCG, ρ, PTS, θ).

1: Initialize the set $PHY(i_1, j_1)$ to false and $BW\_avail(i_1, j_1)$ to $freq_\theta \times lw\theta$, $\forall\ i_1,\ j_1 \in 1 \cdots i$
2: Initialize $switch\_size\_in(j)$ and $switch\_size\_out(j)$ to N_j, $\forall\ j \in\ 1 \cdots i$. Find $switching\_activity(j)$ for each switch, based on the traffic flow within the partition.
3: **for** each flow f_k, $k \in 1 \cdots |F|$ in decreasing order of f_c **do**
4: **for** i1 from 1 to i and j1 from 1 to i **do**
5: {Find the marginal cost of using link i_1, j_1}
6: {If physical link exists, has enough bandwidth for the current flow, and supports the same message type of the current traffic flow}
7: **if** $PHY(i_1, j_1)$ and $BW\_avail(i_1, j_1) \geq f_c$ and same message type **then**
8: Find $cost(i_1, j_1)$, the marginal power consumption to re-use the existing link
9: **else**
10: {We have to open a new physical link between i_1, j_1}
11: Find $cost(i_1, j_1)$, the marginal power consumption for opening and using the link. Evaluate whether switch frequency constraints are satisfied.
12: **end if**
13: **end for**
14: Assign $cost(i_1, j_1)$ to the edge $W(i_1, j_1)$ in SCG
15: Find the least cost path between the partitions in which source (s_k) and destination (d_k) of the flow are present in the SCG. Choose only those paths that have turns not prohibited by PTS
16: Update PHY, BW_avail, $switch\_size\_in$, $switch\_size\_out$, $switching\_activity$ and message type for chosen path
17: **end for**
18: Return the chosen paths, switch sizes, connectivity, message types

procedure. The objective of the procedure is to establish physical links between the switches and to find paths for the traffic flows across the switches. Here, we only present the procedure where the user's design objective is to minimize power consumption. If the design objective is hop count instead, or a combination of both, the same algorithm structure is followed, just with different cost metrics.

An example illustrating the working of the PATH_COMPUTE procedure is presented in Example 3 on the next page. In the procedure, the flows are ordered in decreasing rate requirements, so that more demanding flows are assigned first. The greedy heuristics of assigning these flows first have been shown to provide better results (such as lower power consumption and more easily satisfying bandwidth constraints) in several earlier works [116, 117]. The bandwidth available on each NoC link is the product of the NoC frequency and of the link width. The algorithm ensures that the traffic on each link is less than or equal to its available bandwidth value. For each flow in order, we evaluate the amount of power that would be dissipated across each of the switches, if the traffic for the flow were to use that switch. This power dissipation value on each switch depends on the size of the switch, the amount of traffic already routed on the switch and the architectural parameter point (θ) used. It also depends on how the switch is reached (from what other switch) and whether an already existing physical channel will be used to reach the switch or a new physical channel will have to be opened.

In ×pipes, we permit the instantiation of multiple physical links between any two switches. When finding whether a switch is reachable from another switch for the current traffic flow, we evaluate whether any physical links between the switches have already been established. If so, we check the message types of the traffic flows that have already been routed onto the links. The message types can either be fed explicitly by the user, or can be implicitly considered by the tool - for example, traffic flows that originate from processors and are sent to memory devices are typically "requests", while those in the opposite direction "responses". In many systems, all of the inter-processor communication occurs through memory devices, so this auto-detection suffices. If more complex message types may occur, then each message type will need to be annotated and treated as a separate traffic flow. Based on the gathered message type information, among the set of established links, we search for one that already carries the same message type and still has enough bandwidth available to support the current flow. If no such link is available between the switches, we evaluate the cost of opening up a physical link. Opening a new physical channel increases the switch size and hence the power consumption of this

flow and of the others that are routed through the switch. These marginal power consumption values are assigned as weights on each of the edges reaching the vertex representing that switch in the SCG. This is performed in steps 8 and 11 of the procedure. When opening a new physical link, we also check whether the switch radix is small enough to satisfy the particular frequency of operation. As the switch size increases, the maximum frequency of operation it can support reduces, since the critical path inside the switch gets longer. This information is obtained from the input switch models.

Example 3 *For the SCG from Figure 5.3(b) on page 122, let us consider routing the flow of value* 100 *between the vertices* $v1$ *and* $v2$*, across the partitions* $p1$ *and* $p2$*. Initially no physical paths have been established across any of the switches. If we have to route the flow across a link between any two switches, we have to first establish the link. The cost of routing the flow across any pair of switches is obtained from step 8 of Algorithm 2 on page 127. The SCG with the edges annotated with the costs is presented in Figure 5.4(c) on page 126. The costs on the edges from* $p2$ *are different from the others due to the difference in initial switching activity in* $p2$ *compared to the other switches. This is because the switch* $p2$ *has to support flows between the vertices* $v2$ *and* $v3$ *within the partition. The least cost path for the flow, which is across switches* $p1$ *and* $p2$ *is chosen. Now we have actually established a physical path between these switches and this is considered when routing the other flows. Also, the size and switching activity of these switches have changed, which is noted.*

Example 4 *Let us consider the example from Figure 5.4(a) on page 126. The input core graph has been partitioned into 4 partitions. We assume 2 different message types: request and response for the various traffic flows. Each partition* p_i *corresponds to a set of cores attached to the same switch. Let us consider routing the flow with a bandwidth value of* $100MB/s$ *between the vertices* $v1$ *and* $v2$*, across the partitions* $p1$ *and* $p2$*. The traffic flow is of the message type request. Initially no physical paths have been established across any of the switches. If we have to route the flow across a link between any two switches, we have to first establish the link. The cost of routing the flow across any pair of switches is obtained. We annotate the edges between the switches by the cost (marginal increase in power consumption) of sending the traffic flow through the switches (Figure 5.4(c) on page 126). Switch* $p2$ *has to support internal flows between the vertices* $v2$ *and* $v3$ *within the partition. For this reason, the costs on the edges from* $p2$ *are different from the others, due to the the difference in initial traffic rates within* $p2$ *when compared to the other switches. The least cost path for the flow, which is across switches* $p1$ *and* $p2$*, is chosen. Now we have actually established*

a physical path and a link between these switches. We associate the message type request to this particular link. This is considered when routing other flows, and only those traffic flows that are of request type can use this particular physical link. We also note the radix and switching activity of those switches that have been involved in the routing of the current flow.

Once the weights have been assigned, choosing a path for the traffic flow is equivalent to finding the least cost path in the SCG. This is done by applying Dijkstra's shortest path algorithm [213] in step 15 of Algorithm 2 on page 127. Only those paths that do not use the turns prohibited by the PTS are considered; for the others, the objective function for establishing the best paths is initially set to minimizing power consumption in the switches. The `PATH_COMPUTE` procedure returns the sizes of the switches, the connectivity between the switches and the paths for the traffic flows.

Once the paths have been established, if hop count constraints are not satisfied, the algorithm gradually modifies the objective function to minimize the hop count as well, using the parameter ρ (in steps 7, 10 and 11 of Algorithm 1 on page 125). The upper bound for ρ, denoted by ρ_{thresh}, is set to the value of power consumption of the flow with maximum rate, when it crosses the maximum size switch in the SCG. At this value of ρ, for all traffic flows, it is beneficial to take the path with least number of switches, rather than the most power efficient path. The ρ value is varied in several steps until the hop count constraints are satisfied or until it reaches ρ_{thresh}.

In the next step (step 12 of Algorithm 1 on page 125), the algorithm invokes an external floorplanner, called Parquet [214], to compute the design area and wire lengths. The floorplanner minimizes a dual-objective function of area and wire length, with equal weights assigned to both. Parquet also supports soft cores, fixed pin/pad locations and aspect ratio constraints for the generated design. From the obtained wire lengths, the power consumption across the wires is calculated. Also, the length of the wires is evaluated to check for any timing violations that may occur at the particular frequency ($freq_0$). If needed, pipeline repeaters (Section 4.3.3 on page 101) are automatically deployed (see Section 6.5.8 on page 201 for an example).

The presented NoC synthesis process scales polynomially with the number of cores in the design. The number of topologies evaluated by the methodology also depends linearly on the number of cores. Thus, the algorithms are highly scalable to a large number of cores and communication flows. The synthesis time for several different SoC benchmarks is presented in Section 5.5 on page 133.

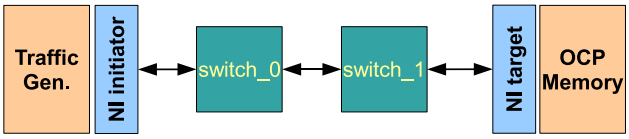


Figure 5.5: A simple NoC topology.

5.4 Topology Instantiation

Once it has chosen the best topology (based on the user's objectives) that satisfies all the design constraints, SunFloor produces two outputs: a NoC topology description and a chip floorplan (Figure 1.5 on page 27). The floorplan is ready for being fed to standard placement tools commonly used in the industry (Chapter 6 on page 145). The topology description, on the other hand, is fed to a topology instantiation tool that we developed, called ×pipesCompiler [215]. A slightly simplified example of a topology description for the topology of Figure 5.5 is in Listing 5.1. ×pipesCompiler takes care of generating the REGISTER TRANSFER LEVEL (RTL) SystemC code of the complete platform, by configuring and interconnecting the ×pipes soft macros based on the specifications of SunFloor.

Listing 5.1: Example topology description file.

```
// In this topology: 1 processor, 2 switches
topology(2switch_1link);

// Cores:
// 1 traffic generator, clock divider 1:1, 4 NI
// buffers
// 1 memory, clock divider 1:1, 6 NI
// buffers, at address 0x00000000
core(core_0, 1, 4, tester, initiator);
core(mem_0,  1, 6, ocpmemory, target:0x00);

// Switches:
// Two 2x2 switches, 2 input buffers, 4 output
// buffers
switch(switch_0, 2, 2, 2, 4);
switch(switch_1, 2, 2, 2, 4);
```

```
// Links:
// All of them non-pipelined
link(link_0,   core_0,    switch_0);
link(link_1,   switch_0,  core_0);
link(link_2,   pm_0,      switch_1);
link(link_3,   switch_1,  pm_0);
link(link_4,   switch_0,  switch_1);
link(link_5,   switch_1,  switch_0);

// Routes:
route(core_0, mem_0,  switches:0,1);
route(mem_0,  core_0, switches:1,0);
```

The tasks performed by ×pipesCompiler involve:

- Performing checks on the input file, verifying the full connectivity of all the system components. (This step is key since the input topology description does not necessarily come from SunFloor; it can also be manually written).
- Configuring the blocks of the ×pipes component library according to the specifications in its input description.
- Creating top-level modules to connect all the blocks together, according to the desired topology.
- Producing suitable routing tables for the NIs, based on the specified communication flows and routes.
- Creating testbenches for the whole topology, capable of stressing all the paths among communicating IP cores in the topology.
- Generating component lists to be used by physical implementation scripts.

×pipesCompiler generates code at the RTL level, both in SystemC and Verilog. This code is suitable for simulation, for FPGA emulation and for ASIC implementation flows.

5.5

Experiments and Case Studies

5.5.1 Experiments on MPSoC Benchmarks

We have applied our topology design procedure to six different SoC benchmarks: IMAGE PROCESSING APPLICATION (IMP), 23 cores, VIDEO PROCESSOR (VPROC), 42 cores, MOTION PICTURE EXPERT GROUP (MPEG)4 decoder, 12 cores, VIDEO OBJECT PLANE DECODER (VOPD), 12 cores, MULTI-WINDOW DISPLAY APPLICATION (MWD), 12 cores and PICTURE-IN-PICTURE (PIP), 8 cores. The communication characteristics of some of these benchmarks are presented in [216].

For comparison, we also generated mesh topologies for the benchmarks by modifying the design procedure to synthesize NoCs based on the mesh structure. To obtain mesh topologies, we generated a design with each core connected to a single switch and restricted the switch radix to 5 input/output ports. We also generated a variant of the basic mesh topology, an optimized mesh (*opt-mesh*), where those ports and links that are unused by the traffic flows are removed. The communication graph and the floorplan for the custom topology synthesized by our tool for one of the benchmarks (VOPD) are shown in Figure 5.6 on page 135. The network power consumption (power consumption across the switches and links), average hop count and design area results for the different benchmarks are presented in Table 5.3 on the following page. Note that the average hop count is the same for the mesh and the opt-mesh, as in the opt-mesh only the unused ports and links of the mesh have been removed and the rest of the connections are maintained. The custom topology results in an average of 2.78× improvement in power consumption and 1.59× improvement in hop count with respect to the standard mesh topologies. The large power savings are due to two reasons: (i) the switch power consumption is reduced in the custom topology, and (ii) the total wire length in the mesh topologies is 1.38× longer than that of the custom topology, which also results in increased link power consumption.

The area of the designs with the different topologies is similar, thanks to efficient floorplanning. It can be seen from Figure 5.6 on page 135 that only very little slack area is left in the floorplan. This is because we consider the area of the network elements during the floorplanning process, and not after the floorplanning of blocks. The total run time of the topology synthesis and architectural parameter setting process for the different benchmarks is also presented in Table 5.3 on the following page. Given the large problem sizes and very large solution space that is explored (8 different frequency steps, 4 different link widths, 42 cores for VPROC and

Application	Topology	Power (mW)	Avg. Hops	Area mm^2	Generation time (mins)
VPROC	custom	79.64	1.67	47.68	68.45
	mesh	301.8	2.58	51.00	
	opt-mesh	136.1	2.58	50.51	
MPEG4	custom	27.24	1.5	13.49	4.04
	mesh	96.82	2.17	15.00	
	opt-mesh	60.97	2.17	15.01	
VOPD	custom	30.0	1.33	23.56	4.47
	mesh	95.94	2.0	23.85	
	opt-mesh	46.48	2.0	23.79	
MWD	custom	20.53	1.15	15.00	3.21
	mesh	90.17	2.0	13.60	
	opt-mesh	38.60	2.0	13.80	
PIP	custom	11.71	1.0	8.95	2.07
	mesh	59.87	2.0	9.60	
	opt-mesh	24.53	2.0	9.30	
IMP	custom	52.13	1.44	29.66	31.52
	mesh	198.9	2.11	29.40	
	opt-mesh	80.15	2.11	29.40	

Table 5.3: Comparison among SunFloor-generated topologies and standard topologies.

several calls to the floorplanner) and the fact that the NoC parameter setting and topology synthesis are important phases, the run-time of the engine is not large. This is mainly due to the use of hierarchical tools for partitioning and floorplanning and our development of fast heuristics to synthesize the topology.

We also performed comparisons among custom-generated topologies and several other standard topologies. For mapping the cores onto the standard topologies, we use the tool from [116]. We optimized the topologies for performance, subject to the design constraints. The comparisons against 5 standard topologies (mesh, torus, hypercube, Clos and butterfly) for an image processing benchmark with 25 cores is presented in Figure 5.7 on the next page. The custom topology created by SunFloor shows large performance improvements (1.73× on average) over the standard topologies.

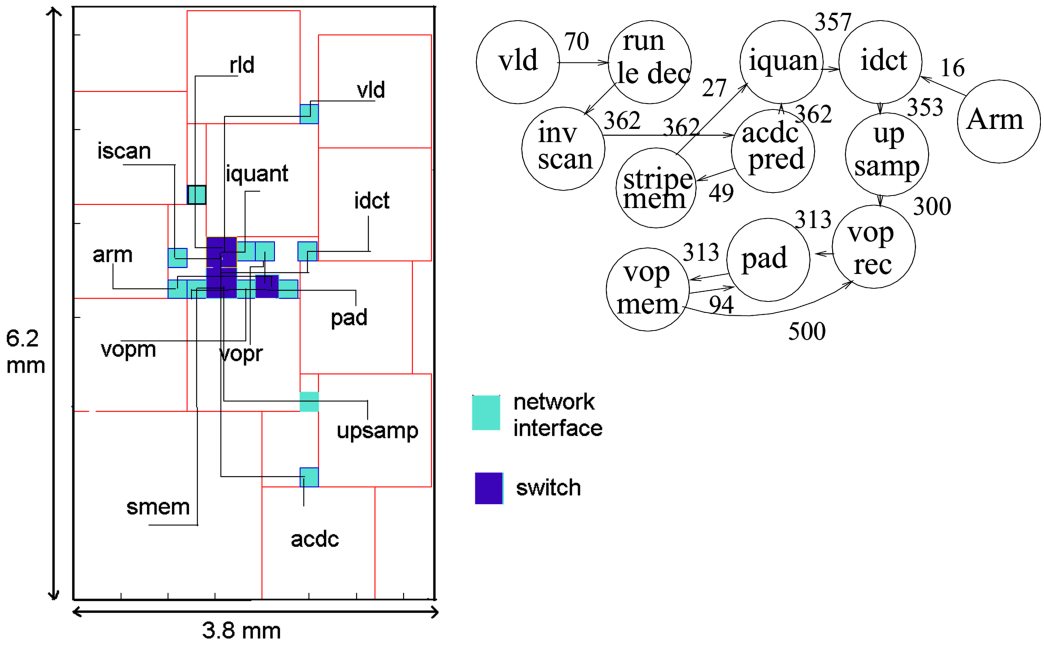


Figure 5.6: VOPD Application: custom topology floorplan and communication graph.

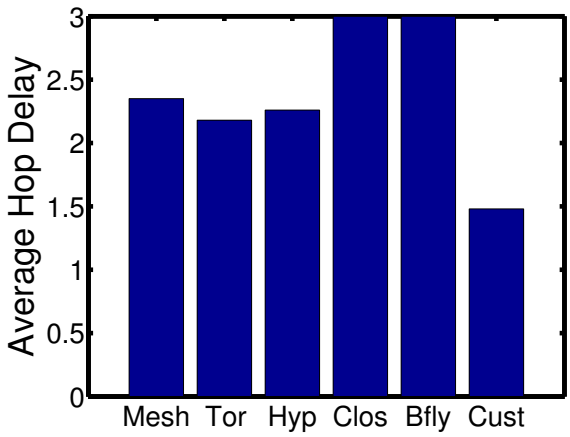


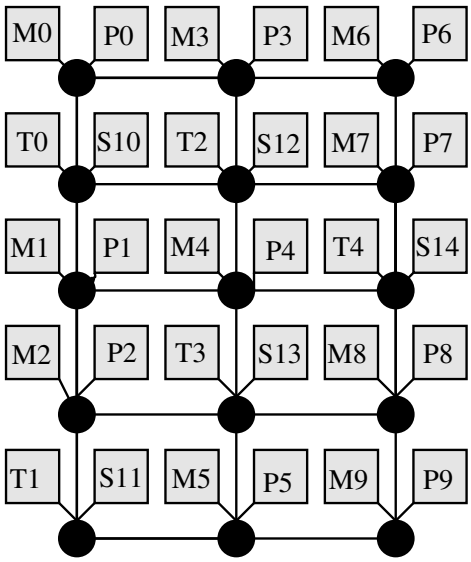
Figure 5.7: Image processing application: hop delay comparison across regular and custom topologies.

5.5.2 Case Study: A Layout-Level Comparisons

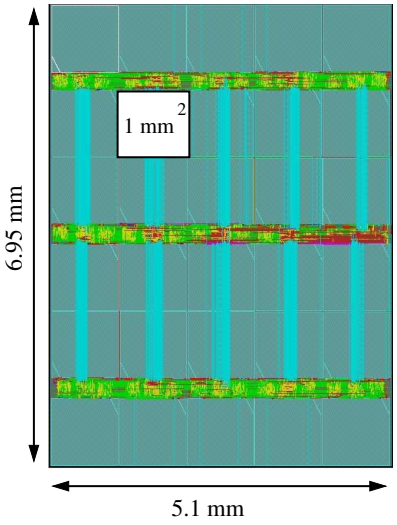
We present here a comparison based on a NoC design for a multimedia SoC that will be discussed in more detail in Section 6.3 on page 152. The design consists of 30 cores. The NoC was implemented twice, in 130nm technology. The first time, it was manually instantiated as 15 switches forming a 5x3 quasi-mesh (two cores connected to each switch), as per Figure 5.8(a) on the next page. The design is highly optimized, with most memories very close to the processors accessing them (only one hop away). The layout of the design was performed using SoC Encounter preserving the mesh physical structure. Each of the cores has an area of 1 mm^2 (Section 6.3.3 on page 158) in the design. The entire process, from topology specification to layout generation, took several weeks. The post-layout NoC could support a maximum frequency of operation of 885 MHz, determined by the critical path in the switch pipeline. The power consumption of the topology for functional traffic was evaluated to be 368 mW.

We then applied our topology synthesis process to automatically synthesize the NoC for this application, with the objective of minimizing power consumption. We set the design constraints and the required frequency of operation to be the same as those of the hand-designed topology. The synthesized NoC topology and the layout obtained using SoC Encounter are presented in Figure 5.8(c) on the next page and Figure 5.8(d) on the facing page. The synthesized topology has only 8 switches, half of the hand-designed topology. It can support the same maximum frequency of operation (885 MHz) without any timing violations on the wires. As we considered the wire lengths during the synthesis process to estimate the frequency that could be supported, we could synthesize the most power efficient topology that would still meet the target frequency. The result is a power consumption of just 277mW, 1.33× lower than in the hand-designed case. Given the fact that the hand-designed topology is highly optimized, with much of the communicating traffic traversing only one switch, these savings are achieved entirely from efficiently spreading the shared memories around the different switches. The layout of the hand-designed NoC was manually optimized to a large extent (by moving switches, network interfaces) to reduce the area of the design. The layout of the synthesized topology is obtained completely automatically, and still the area of the design is only marginally (4.3%) worse.

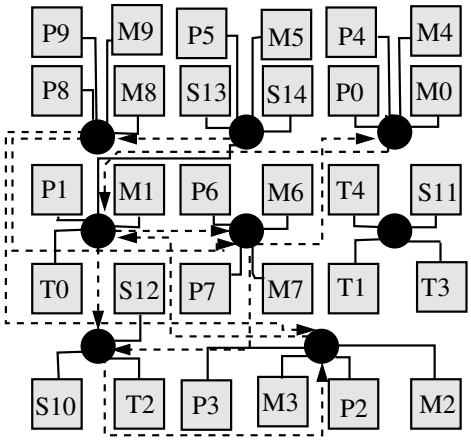
The synthesized topology also exhibits a lower design area (about 1.2× lower) compared to the hand-designed NoC. This is also because, in the hand-designed NoC, the mesh network was introduced after fixing the positions of the cores, while in our method the network components



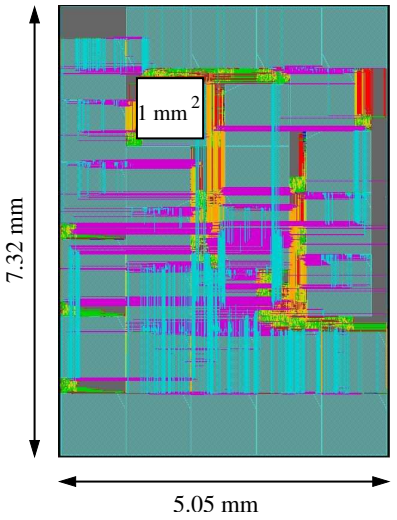
(a) Hand-designed topology.



(b) Layout of the hand-designed topology.



(c) Automatically-generated topology. Bidirectional links are solid, unidirectional links are dotted.



(d) Layout of the automaticallygenerated topology.

Figure 5.8: Hand-designed and automatically-generated topologies and their layouts. **M**: ARM7 masters; **T**: traffic generators; **P**: privately accessed slaves; **S**: shared slaves.

are considered during the floorplanning process itself.

Finally, we also performed cycle-accurate simulations of the hand-designed and the synthesized NoCs for two multimedia benchmarks. In terms of performance, the custom topology not only matches the performance of the hand-designed topology, but provides an average of 10% reduction in total execution time and of 11.3% in packet latency, thanks to a lower average hop count.

Reaching such an optimized design point manually would have required a large number of iterations of topology design, synthesis, placement and routing, which is a very time consuming process. On the contrary, with our proposed flow, the time to get a complete, final design was around just four hours.

5.5.3 Message-Dependent Deadlock Removal

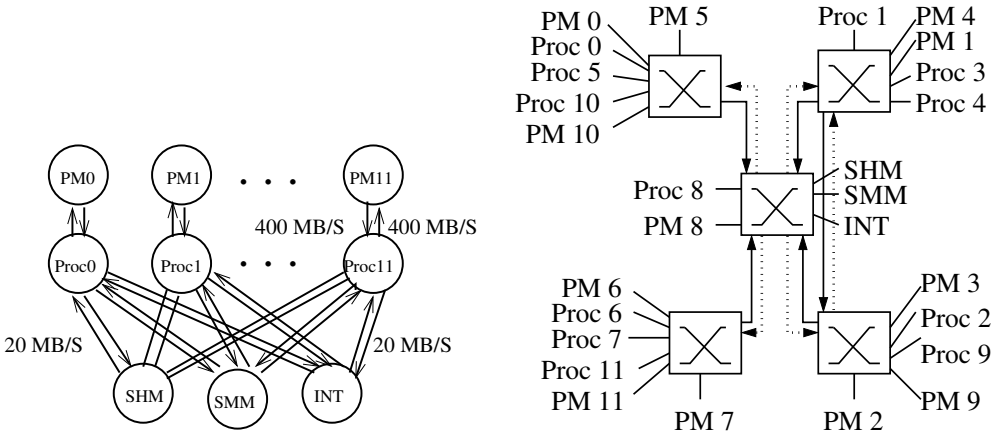
In this section, we present detailed experimental studies of our approach (subsequently referred to as message-dependent deadlock avoidance INTEGRATED WITH THE TOPOLOGY (INT-TOP) synthesis process), and compare it in terms of power and area against traditional approaches:

1. Using LOGICALLY SEPARATE NETWORKS (L-SEP): separate buffers are instantiated at each input, with as many buffers as the different message types, modeling the virtual channel based approach to remove message-dependent deadlocks.

2. Using PHYSICALLY SEPARATE NETWORKS (P-SEP): physically different networks are deployed for each message type. For both P-SEP and L-SEP, we apply our topology synthesis procedure to obtain the network topologies.

3. A design that has no support to avoid message-dependent deadlocks, called ORIGINAL (ORIG). Note that this base system cannot typically be employed in SoCs, as it cannot guarantee proper system operation (unless some deadlock recovery support is provided higher up in the protocol stack). We present the experimental results for this scheme only to evaluate the overhead incurred by the other schemes to support deadlock-free operation.

We apply the message-dependent deadlock prevention methods to five different SoC designs: MULTI-MEDIA SYSTEM (MULT), 30 cores, IMAGE PROCESSING APPLICATION (IMP), 27 cores, VIDEO PROCESSOR (VPROC), 42 cores, MOTION PICTURE EXPERT GROUP (MPEG)4 decoder,



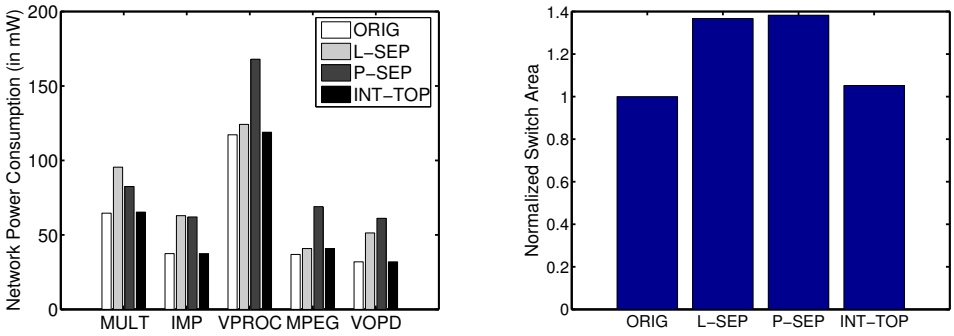
(a) Communication graph. (b) Designed topology.

Figure 5.9: IMP Application.

12 cores and VIDEO OBJECT PLANE DECODER (VOPD), 12 cores. The communication characteristics of some of these benchmarks are presented in [216]. Two types of messages are used in each design: requests and responses. Each design consists of an almost equal number of request and response traffic flows, since every processor core communicates with memory cores, and this two-way communication is a request/response pair. To make a fair comparison of the different schemes, we use the same synthesis approach and design constraints for synthesizing all the topologies.

The communication graph for the IMP application and the best corresponding topology achieved by our proposed scheme INT-TOP are presented in Figure 5.9(a) and Figure 5.9(b). The design consists of 12 processors, a PRIVATE MEMORY (PM) for each processor, a SHARED MEMORY (SHM), a SEMAPHORE MEMORY (SMM) device and an INTERRUPT (INT) device. In the application, all communication from the processors belongs to the request message type, while communication to the processors is of the response type. In Figure 5.9(b), request links are plotted with continuous lines, while response links are dotted.

The network power consumption, based on functional traffic, for the various designs using the different schemes is presented in Figure 5.10(a) on the following page. As seen from this figure, the proposed INT-TOP scheme outperforms the two conventional message-dependent deadlock avoidance schemes L-SEP and P-SEP, leading by an average of 38.5% reduction in NoC power consumption when compared to the state-of-the-art deadlock avoidance schemes. When compared to INT-TOP, the L-



(a) Power consumption. (b) Normalized switch area.

Figure 5.10: Power and area cost of alternative message-dependent deadlock avoidance schemes.

SEP scheme requires larger buffering resources, as each virtual channel needs separate buffers. The P-SEP scheme requires more switches than INT-TOP, as the request and response messages utilize different networks. Interestingly, our proposed scheme incurs only a 2.5% increase in power consumption when compared to the ORIG scheme, where no message-dependent deadlock avoidance support is provided. This is mostly due to the efficient allocation of links to the different message types by our topology synthesis procedure. The switch area for the different schemes for the SoC designs, normalized with respect to the area of the ORIG base system, is presented in Figure 5.10(b). The proposed method results in an average of 30.7% reduction in area when compared to the state-of-the-art schemes.

We now examine the power consumption of the proposed scheme when the amount of different types of messages is varied. The number of message types in a system depends on the underlying computation architecture. Cache coherent systems typically support several different message types. As an example, the S-1 multi-processor supports 4 different message types [211] and each type must be mapped onto different resources in the network. In [137], a more sophisticated protocol is used, which leads to seven different message types. To see the impact on the number of different message types, we create a synthetic benchmark having the traffic characteristics of the VPROC design. In this benchmark, around 80 different traffic flows exist, each one representing a message. We keep the number of messages fixed and vary the number of message types in the design from 1 to 7. The network power consumption of INT-TOP for the different number of message types is presented in Figure 5.11 on the facing page. This figure shows that our proposed scheme results in

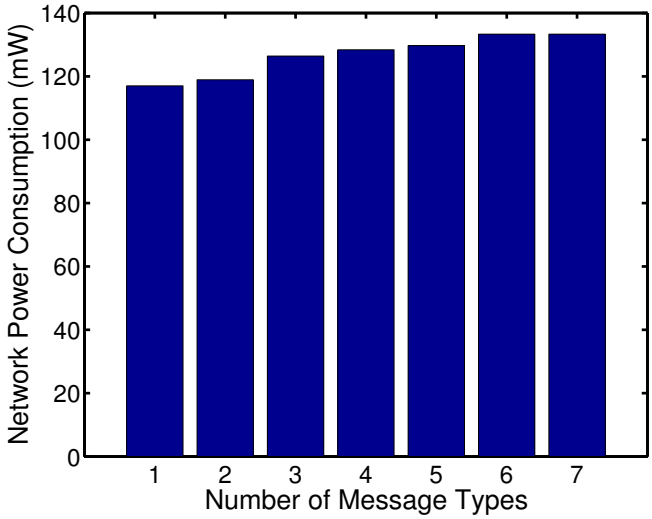


Figure 5.11: Effect of number of message types on the power consumption of a deadlock-free NoC.

efficient designs, even for a large number of message types. Moreover, the rise in power consumption with an increasing number of message types saturates (designs with 6 and 7 message types have nearly the same power consumption), as most messages are already mapped onto unique links in the network.

5.5.4 Compensation for Congestion Effects

When the designed NoC is simulated at the cycle-accurate level, there can be some mismatch between the observed traffic patterns and the initial traffic estimates. This may be either because of inaccurate traffic models or because of dynamic effects, such as congestion. It would be too time consuming to perform a detailed simulation of each topology during the synthesis process to quantify second-order effects. To bridge the gap between topology synthesis and simulation, we use a *mismatch* parameter; the input traffic rates are multiplied by the value of this parameter. The parameter is fed as an input to SunFloor. It is initially set to 1 and the user can manually tune the parameter and re-design the NoC, until the simulations satisfy the required performance level. The effect of increasing the parameter on performance for the MPEG4 NoC is presented in Figure 5.12 on the next page. Extensions of the concept to handle localized congestion effects in the NoC are currently underway.

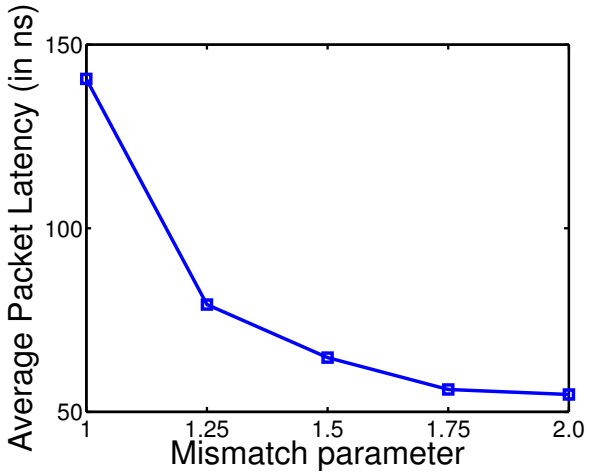


Figure 5.12: Effect of the compensation of dynamic effects, such as congestion, for MPEG4.

5.6 Conclusions

Power-, area- and latency-efficient NoC designs are crucial for industrial adoption. To achieve these results, the communication architecture should closely match the application traffic characteristics, satisfying the different design constraints. Furthermore, it should match chip floorplan constraints and avoid deadlocks. Synthesizing such NoCs is non-trivial, given the large design space that needs to be explored. We have presented a methodology that automates the process, generating efficient NoCs that satisfy the constraints of the application. To minimize respins and provide a faster time-to-market, we consider floorplan information early in the design cycle, while keeping the process fast. This leads to detecting timing violations on the NoC links during the NoC synthesis phase, thereby leading to timing closure with quicker convergence between the high level design and the physical design phases. We use accurate switch and link power models that are based on layouts of the components. We also integrate deadlock-free routing methods in the NoC synthesis process.

Experiments on several SoC benchmarks show that the synthesized topologies are much better (an average of 2.78× power reduction, 1.59× hop count reduction) than the best mesh topology and mesh-based custom topologies for our case studies.

An actual 130nm case study layout obtained from an industrial tool

(Cadence SoC Encounter [217]) of a 30-core multi-media SoC with the NoC designed using our methodology is also presented. At the layout level, the designed NoC supports the required frequency of operation (close to 900 MHz) without timing violations. We could design the NoC architecture from input specifications to layout in 4 hours, a process that used to take several weeks. A layout level comparison with a hand-designed architecture shows that our automatic design methodology produces excellent results (in terms of power consumption and performance), matching those of hand-crafted designs.

By removing message-dependent deadlocks already during the topology generation phase, we can achieve large reductions in network power consumption (38.5%) and network area (30.7%) when compared to alternative state-of-the-art approaches. The presented tool automates the entire NoC topology design process, including topology synthesis, routing and path computation, RTL code generation and floorplan generation, and seamlessly integrates in a complete flow including standard industrial back-end physical design tools, thereby bridging an important gap in the design of application-specific NoCs.

Future research can be focused on the generation of heterogeneous topologies, for example with regions operating at different frequencies or featuring different data widths.

CHAPTER 6

NoC Design Flow Back-End: Physical Implementation

This chapter[1] describes the back-end of the proposed flow (Figure 6.1 on the following page). Its goal is the streamlined physical implementation of a given NoC design, with main emphasis on ASIC targets. A number of variables come into play; among the most relevant, the technological library (*i.e.* the manufacturing process) and the choice of tooling among several available industrial alternatives have a deep impact on the quality of the final results. We apply our resulting flow to compare NoCs against bus architectures, to assess the scalability of NoCs to next-generation technology nodes, and to obtain area and power models.

6.1 Motivation and Key Challenges

Due to the quick pace of lithographic miniaturization, it is nowadays well known that a number of physical-level process issues related to deep submicron fabrication (such as wire delays and leakage power) are affecting designs. Understanding these issues is clearly key to tackling them, for example by compensating for them at the architectural level.

In the case of NoCs, the relationship among back-end flows and archiectural design is even stricter, because of several factors:

[1] This chapter is the outcome of the collaboration with many co-authors, among which the author would like to give special credit to Antonio Pullini, Paolo Meloni, Prof. Salvatore Carta, Prof. Luca Benini.

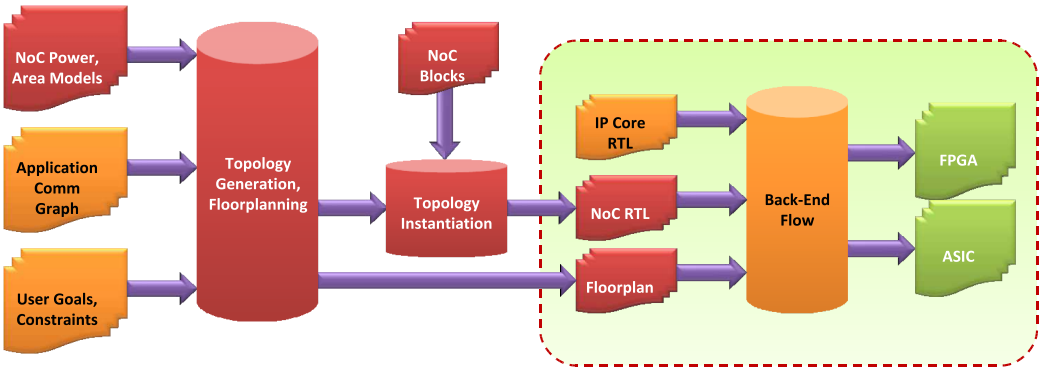


Figure 6.1: The proposed NoC design flow: back-end.

- One of the main purposes of NoCs is exactly to help in tackling wire-related physical-level issues.

- NoCs are intended to be large structures, spread across a whole chip. As such, several design issues, such as clock tree distribution, wire delays and variability, play a key role in NoCs.

- NoCs are also designed to interconnect a large number of heterogeneous components and devices, each of which could come as a pre-built, pre-characterized IP macro. Thus, it is key to be able to leverage standard back-end industrial toolchains for NoC design, else the effort of developing customized infrastructure would be impossible to afford.

As a main assumption of the reasearch activity described in this dissertation, we focus on standard cell-based physical implementations. While full custom design does certainly improve results, it does also greatly decrease flexibility and increase design time. The development of custom blocks to improve the efficiency of some specific pieces of a NoC can, however, certainly be seen as a direction for future research.

In the following, we will first of all discuss a traditional back-end flow to bring a circuit description to a chip implementation. We will then discuss why this is insufficient to cope with today's technology, how we tackle the resulting challenges in our NoC flow, and what insight we gain from this activity. We will move on to presenting a cross-benchmarking study we performed in 130nm technology by comparing a NoC- and a hierarchical bus-based implementation of the interconnect of a multimedia system. As will be seen, being able to rely on proper back-end assessments

is crucial in order to get a complete picture. We will then show how to extract area and power models for NoCs for use by SunFloor (Section 5.2 on page 121). We will eventually show some of the insight we gained by implementing NoCs in state-of-the-art 65nm technology. It is also worth stating that, while we will not present those results here, ×pipes has also been ported to FPGA for fast emulation purposes [101].

6.2 A Synthesis Flow

6.2.1 A Traditional View of the Back-End Design Flow

A traditional back-end design flow based on standard cells is depicted in Figure 6.2 on the following page. This kind of flow features a streamlined sequence of steps, which are ideally as decoupled as possible.

- Starting from a description of the circuit in some RTL language, such as VHDL or Verilog, logic synthesis is initially performed; this translates RTL descriptions into a so-called netlist, *i.e.* a connected network of basic gates belonging to a technology library. The technology library is an abstracted view of the underlying foundry process, and describes the basic gates in terms of function (such as boolean gate, flip-flop, *etc.*), propagation delay, capacitive load, *etc.*. Based on this information an on user constraints, a main task of logic synthesis is to make sure that the netlist fulfills speed, area and power consumption goals.
- The gates of the netlist are subsequently placed, *i.e.* mapped onto a canvas representing the geometrical shape of the final device - this is typically a rectangle. Placement involves both a high-level arrangement of the main functional blocks of the chip (a step often called *floorplanning*) and a low-level arrangement of each single gate (*detailed placement*).
- Finally, the routing step takes care of laying metal lines to attach the placed gates to each other, so that the circuit can function. During this stage, some signals (typically, the power supply and the clock) play a special role, since they must be distributed to a large number of gates spread all over the chip. Special attention is paid, for example, to the minimization of the skew in the clock distribution network.

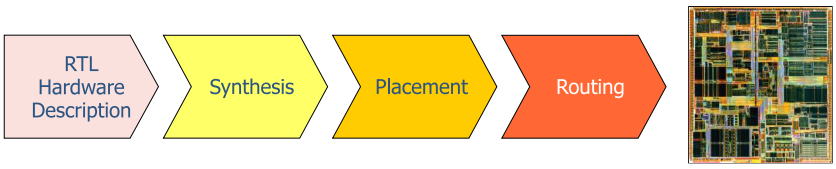


Figure 6.2: A schematic view of a traditional design flow.

Of course, in this reference flow, any constraint violation - such as the impossibility to route the wires to connect the gates of the placed netlist, or an unexpected violation of the required circuit speed - can only be tackled by feedback loops where one or more steps are repeated again, under different assumptions.

However, this basic flow is not sufficient any more to deal with today's technology, for reasons that will become more clear in the following. A crucial point of failure is that it becomes increasingly time-consuming, complex, and potentially even unfeasible, to maintain the strict separation among the steps of the traditional flow sequence; routing issues are nowadays setting an increasing amount of constraints on feasible placements, and this applies, in turn, to all the upstream steps of the flow. Therefore, the number of detected violations and of required feedback loops in physical implementation would become too large for the traditional flow paradigm to hold without changes. In response to this, new solutions must be found, either by proactively tackling issues (and NoCs at large are in some sense doing this, *e.g.* by simplifying routing through an architectural breakthrough), or by simultaneously performing multiple steps at once, with wider constraint visibility.

In the following, we will present an outline of our backend flow, subsequently focusing our attention on specific portions of the flow which have particular relevance.

6.2.2 The ×pipes Back-End Infrastructure

In the proposed NoC design and synthesis framework for ×pipes, we provide a complete back-end flow based on standard cell synthesis (see Figure 6.3 on the next page). Without any loss of generality in our conclusions, we focus on standard cell-based physical implementations. In fact, although full custom design does certainly improve results, it does also greatly decrease flexibility and largely increases design time; thus, it is not a desirable practice for the design of current, and especially forthcoming, MPSoCs interconnects.

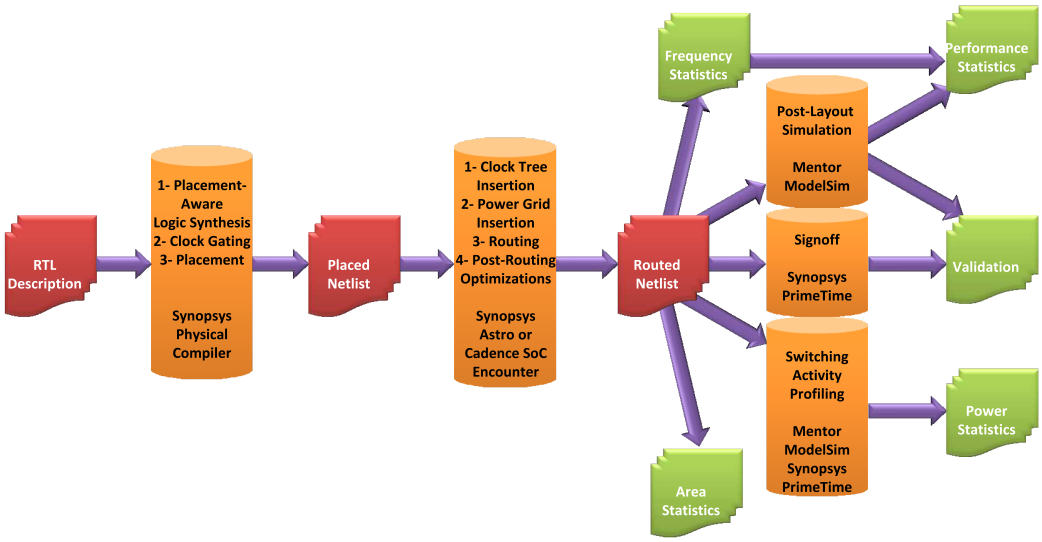


Figure 6.3: The synthesis flow for ×pipes.

First, we perform logic synthesis by utilizing standard Synopsys tools; depending on the underlying technology library, this step may need be augmented with placement awareness, as will be discussed in Section 6.2.3 on page 151. We support this procedure on 130nm, 90nm and 65nm technology libraries by partner foundries, tuned for different performance/power tradeoffs, with different threshold and supply voltages.

During synthesis, we can optionally instruct the tools to save power when buffers are inactive by applying clock gating to NoC blocks. The gating logic can be instantiated only for sequential cells that feature an *input enable* pin, which are a large majority of the datapath flip-flops of ×pipes.

We subsequently perform the detailed placement&routing step within either Synopsys Astro [218] or Cadence SoC Encounter [217] (in the following, we will sometimes refer to either of them for the sake of brevity). First, we feed Astro with a coarse floorplan, generated either manually or by SunFloor. This floorplan contains *hard macros* and *soft macros*, separated by fences (Figure 6.4 on the next page). The hard macros represent cores and memories, and are modeled as black boxes. Hard macros are defined with a LIBRARY EXCHANGE FORMAT (LEF) file and a Verilog Interface Logical Model, and obstruct an area of choice. These boxes also obstruct some of the metal layers laying directly above; the exact number of obstructed levels is configurable, depending on how many metal layers the cores are supposed to require and on whether over-the-cell routing

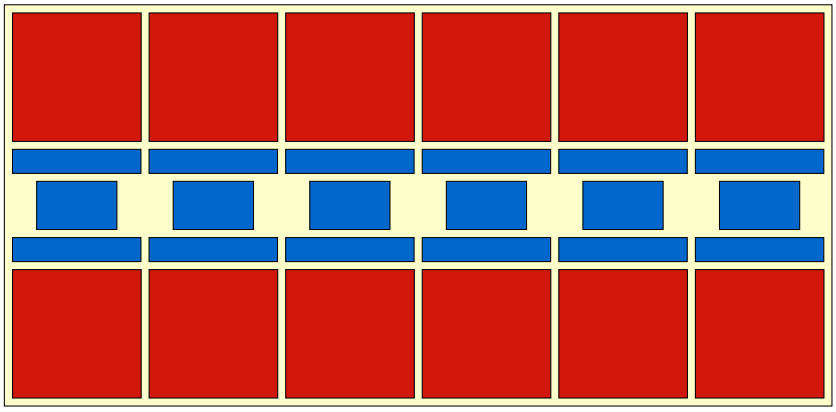


Figure 6.4: Example of the usage of fences in placement. Yellow area: floorplan; red areas: hard macros for IP cores; blue areas: soft macros for NoC components.

should be allowed for the NoC wires *vs.* between-the-cell. Soft macros are also boxes; they enclose the modules of ×pipes, and the placement tool is allowed to operate within them as long as the fences are not trespassed. By constraining the placement tool to operate on a "tile" at a time, the solution space is dramatically pruned, and relatively fast runtimes can be achieved. For proper results, however, it becomes necessary to specify rough timing constraints at the soft macro boundaries; we achieve this by pre-characterization of the links (Section 6.5.2 on page 192).

The next step in the flow is clock tree insertion. We instantiate a clock tree within each soft macro, to minimize the memory requirements and runtime of this operation; the clock trees are then attached to a common source and balanced at the global level. The clock tree can leverage *clock borrowing* algorithms in the tools. In other words, instead of trying to fully erase clock skews (an impossible task anyway), the skews are exploited to accommodate the delay properties of the circuits, by supplying wider clock periods where the logic paths are most critical. Once the clock tree has been generated, its wires are kept untouched within the tool, to prevent further skews from appearing.

At this point, the power supply nets are added. Two main schemes are available. Traditionally, *power rings* (metal lines carrying the power supply voltages) are laid around the die; as an alternative, a *power grid* can be laid across the chip in the topmost metal layers. The latter choice requires more metal resources, but minimizes *IR drops* (voltage drops and fluctuations due to resistive effects in the supply networks and to the current draw). Therefore, we choose power grids, so as to maximize voltage stability.

Next, the routing tool begins to route the logic wires. An initial heuristic mapping lays the wires; this initial solution is semi-random and almost certainly violates essential constraints, such as that of not shorting different wires. Therefore, SEARCH&REPAIR (SR) loops are executed to fix any violations, including those regarding excessive propagation delays.

Post-routing optimizations are then performed. This stage includes crosstalk minimization, antenna effect minimization, and insertion of filler cells. Finally, a *sign-off* procedure can be run by using Synopsys PrimeTime [219] to accurately validate the timing properties of the resulting design.

Post-layout verification and power estimation is achieved as follows. First, the netlist representing the final placed&routed topology, including accurate delay models, is simulated by injecting functional traffic through the OCP ports of the NIs. This simulation is aimed both at verifying the functionality of the placed fabric and at collecting a switching activity report. At this point, accurate wire capacitance and resistance information, as back-annotated from the placed&routed layout, is combined with the switching activity report using Synopsys PrimeTime [219]. The output is a layout-aware power/energy estimation of the simulation.

6.2.3 Placement-Aware Logic Synthesis

As mentioned above, the traditional flow for standard cell design features logic synthesis and placement as two clearly decoupled stages. Our experience [220] shows that this flow achieves reasonable results for 130nm and 90nm NoC designs, but we have found the situation to be substantially different at the 65nm node.

The origin of the problem lies in the decoupling of the two steps. Synthesis and placement could be considered as independent when wire delays were negligible; this is unfortunately not the case anymore [3]. Since wire delays can be comparable to logic delays, if not larger, it is crucial to be able to estimate wire delays already during synthesis. Since wire delays depend directly on wire length, it is clear that placement algorithms are also unfortunately affecting the solution space of synthesis algorithms.

To alleviate the problem, *wireload models* have been introduced. Wireload models are pre-characterized equations, supplied within technology libraries, that attempt to predict the capacitive load that a gate will have to drive based on its fan-out and on the overall design area. Unfortunately, wireload models remain a statistical representation of the physical reality, and are therefore an inaccurate tool to predict delays on a single net basis, given that each net could exhibit a different behavior. In our

65nm tests, we experience unacceptable performance degradation due to either under- or over-estimations of wire loads. Even when synthesizing single NoC modules (*i.e.*, even without considering long links), the logic synthesis tools generate a netlist with the expectation of some operating frequency; however, after placement, the actually reachable frequency is often up to 30% worse (and even lower after the routing phase). Furthermore, sometimes placement and routing tools simply do not converge towards any solution at all, trying in vain to match the expectations set by the logic synthesis step.

To address this issue NoC synthesis in 65nm requires *placement-aware* logic synthesis tools, such as Synopsys Physical Compiler [221]. Therefore, in the proposed NoC back-end flow, after a very quick initial logic synthesis based on wireload models, the tool internally attempts a coarse placement of the current netlist. Next, it iteratively optimizes the netlist and the placement, based on the actual wire loads implied by the current candidate placement. The outcome is a placed netlist that is optimized also accounting for wire delays.

We also observe in our study of NoC synthesis that other issues may arise when placing gates into soft macros. For example, in our test designs, placement tools perform poorly when modules have to be placed within fences which are either too small or too wide. While the former case is clearly understandable, we attribute the unexpected latter effect to the placement heuristics, which are probably performing worse when the solution space becomes very large. The problem must be solved by proper tuning of the spacing among the soft macro fences and, consequently, accurate area models of the NoC modules are required to avoid very time-consuming iterations.

6.3 Cross-Benchmarking: NoCs Against Buses

In this section, we focus on a detailed comparison among NoCs and (hierarchical) buses, considering performance, power, area, and ease of design. The comparison is made at the 130nm technology node and leverages an older, less efficient version of the ×pipes NoC than the one described in Chapter 4 on page 95. We expect NoCs to fare even better in a future study on state-of-the-art technologies.

6.3.1 The Fabrics Under Test

Choosing a test environment to compare such different architectures as a NoC and a bus/crossbar is a difficult task, since communication fabrics can be heavily tuned to optimally fit a target benchmark - but the resulting figures would not be representative of real-world performance under a different test load. For this reason, we do not attempt to fully optimize our evaluation platforms and we choose to present relatively regular mappings which should be suitable for multiple applications. For the ×pipes NoC, we also include in our analysis an irregular topology, optimized by SunFloor to better match the target application. The topology is optimized according to criteria of low area occupation and low power consumption, and is built for deadlock freedom.

We conceive four test platforms, namely an AMBA AHB shared bus (Section 2.2 on page 32), an AMBA AHB ML system containing a crossbar element, a ×pipes mesh, and a ×pipes custom topology generated by SunFloor (Figure 6.5 on the next page). All fabrics allow for attaching up to 30 IP cores, of which 15 masters and 15 slaves (typically memory banks). This amount is justified considering that, already at the 130nm lithography node, simple processor elements and 32 kB memory banks can be expected to require just about 1 mm^2 of die area. In fact, this number of IP cores can well be surpassed in some current and next-generation MPSoCs.

The ML AMBA topology is not a full crossbar. A full 15x15 crossbar would be prohibitively expensive in terms of area and wiring. In fact, the IP library we use to synthesize this fabric (see Section 6.3.3 on page 158) only allows instantiation of up to 8x8 components. Our ML AMBA test fabric contains a mid-sized 5x5 crossbar. For both the ML and shared bus AMBA designs, the canonical data width of 32 bits is chosen, since it represents the best match for ARM7 cores.

For the ×pipes NoCs, we instantiate non-pipelined links in the assumption that the nature of the topologies should provide enough wire segmentation to guarantee single-cycle propagation on all links, at least in 130nm technology. Experimental results (Section 6.3.4 on page 160) will confirm this assumption. The NoC mesh is configured with two different flit sizes, namely 21 and 38 bits, to explore the dependency of area, power consumption and performance on this parameter. These numbers are chosen taking into account the length of each possible packet type and trying to optimize the resulting flit decomposition. The OCP pinout is configured with 32 bit data ports. The custom NoC topology is configured with 21-bit flits to compare it against the mesh. For all our experiments, the NoC components (switches and NIs) are always configured with FIFO buffers having

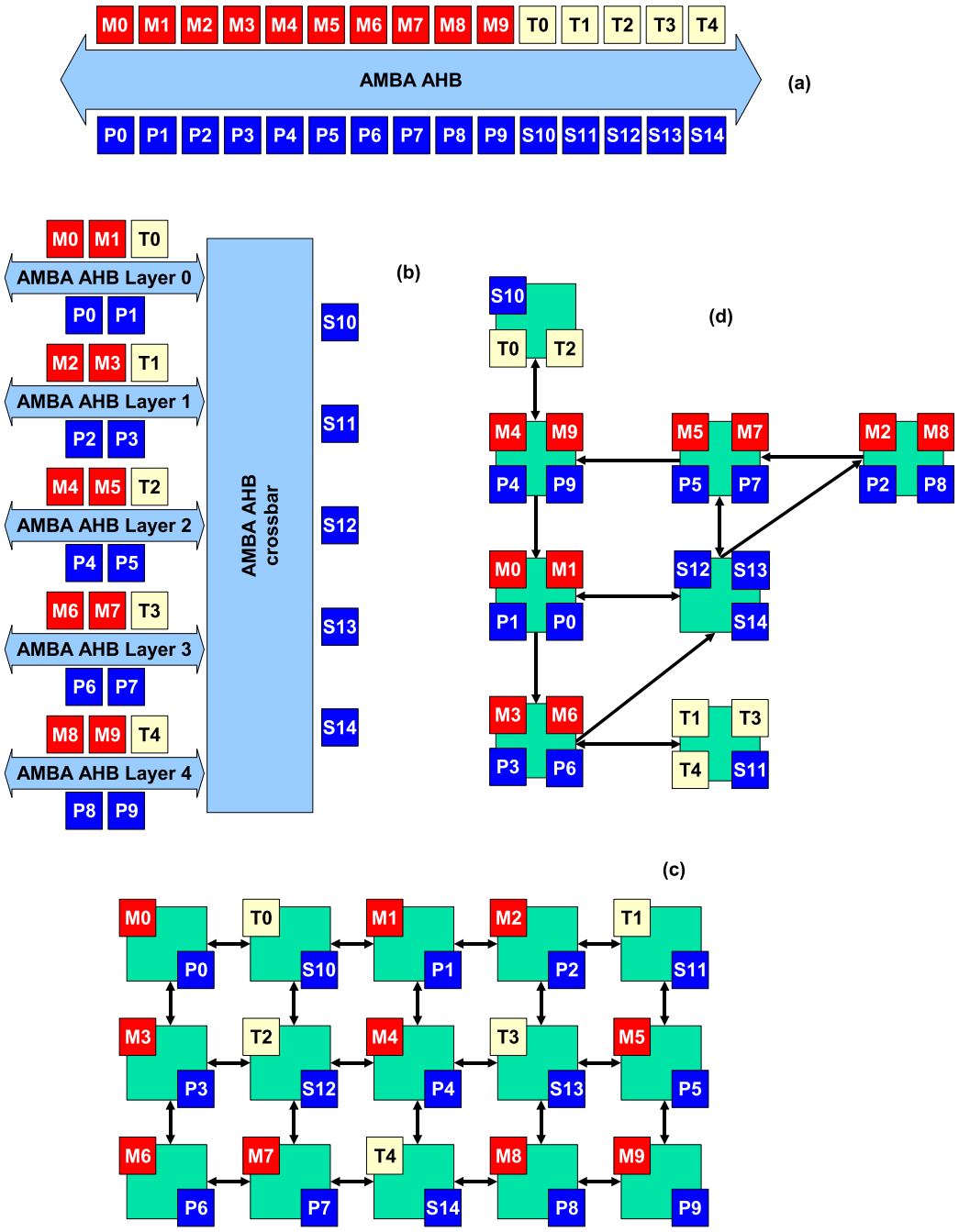


Figure 6.5: The platform fabrics under test. (a) shared bus AMBA AHB; (b) ML AMBA AHB; (c) ×pipes mesh; (d) ×pipes custom topology. **M**: ARM7 masters; **T**: traffic generators; **P**: privately accessed slaves; **S**: shared slaves.

a depth of three flits. In our testing, this value proved to be a good tradeoff between performance and area/power cost.

6.3.2 The Test Applications

We study the performance of the interconnects under two main scenarios, a multimedia processing application and a DATA ENCRYPTION STANDARD (DES) encryption algorithm. Both applications are parallelized to be suitable for multiprocessor computation. The communication graphs for both can be seen in Figure 6.6 on the following page. As can be noticed, the multimedia application is fundamentally a pipeline of computation tasks; the encryption application features a producer task (to split an incoming data stream into chunks of data), a consumer task (to reassemble the outputs) and a set of "worker" tasks which operate in parallel to perform the actual encryption. In our testing, every task is mapped onto a single processor. We let both applications run for several iterations by feeding them with a stream of input data, and capture performance statistics only during the execution of the application kernel, *i.e.* skipping the boot stage and properly handling initialization or shutdown periods where some of the tasks may be running while some others may be idle. This guarantees proper handling of cache-related effects.

We implement the multimedia application as a standalone program, which can directly run on ARM CPUs (M0 to M9 in Figure 6.5 on the preceding page), while the encryption algorithm is an example of a software running on top of the RTEMS [188] operating system.

In both benchmarks, communication between nodes is handled by means of a shared memory buffer, while synchronization is achieved via polling of hardware semaphores. The shared memory and the hardware semaphore device are labeled S12 and S13 in Figure 6.5 on the facing page. To avoid the shared memory to become a huge system bottleneck, processors are assigned private cacheable memory buffers (P0 to P9 in Figure 6.5 on the preceding page), while the shared components are non-cacheable to avoid coherency issues. Therefore, the inter-processor communication paradigm is as follows:

- Producers prepare a data set in their private memory space. In the meanwhile, consumers operate on the previous chunk of data in their private memory space.

- When ready, producers copy the new data set to shared memory. This may need semaphore polling if the shared memory buffer is

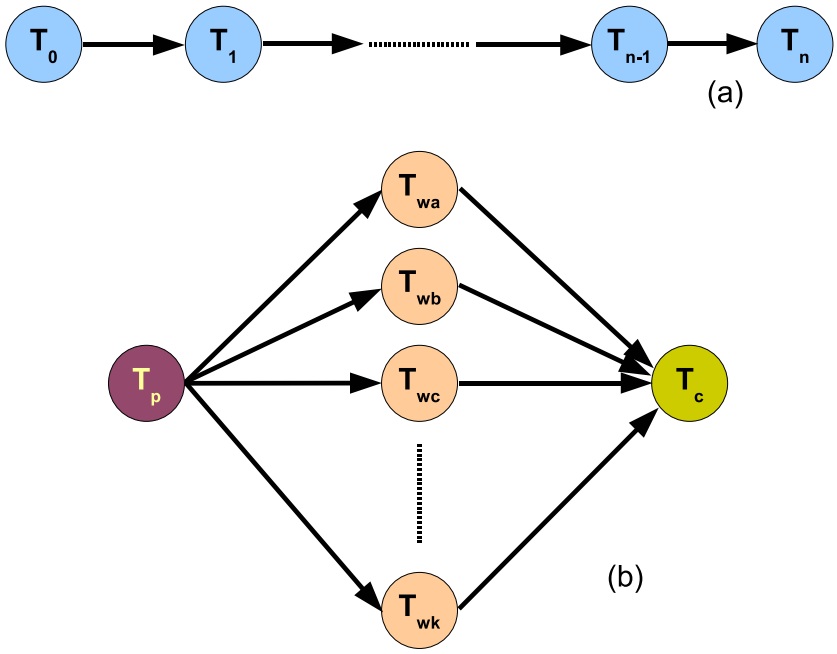


Figure 6.6: Communication graphs for the two applications under test: (a) multimedia processing application, (b) DES application.

still busy with the previous transaction. As soon as data is copied to shared memory, the producer begins preparing the new message.

- When ready, consumers acquire the new data set from shared memory. This may need semaphore polling if the shared memory buffer does not contain new data yet. As soon as data is copied from shared memory, the consumer begins computation on it.

This communication paradigm is just one of almost endless alternative possibilities. We feel that, since it features both distributed (private memories) and centralized (single shared memory and semaphore device) elements, it represents a fair comparison ground for such diverse communication fabric topologies such as a shared bus and a NoC. It may be assumed that an approach based on a fully shared memory subsystem would improve the relative performance of a bus-based fabric, while a message passing paradigm would be more suitable for distributed architectures such as NoCs. However, this analysis is beyond the scope of this research.

To verify whether the computation/communication ratio of the applications is a critical factor, we implement two variants of the multimedia

application benchmark, with different degrees of computational requirements: the low-computation variant is performing roughly eight times less mathematical operations. Please note, however, that while the ratio of computation to *explicit inter-processor* communication can be easily tuned in this way, the ratio of computation to *overall* communication requirements depends on several additional factors. For example, unless an ideal cache is available, changing the computation patterns also *implicitly* results in different bandwidth demands (for cache refills and write-backs). This will be further discussed in Section 6.3.4 on page 160.

In the following, for the sake of brevity, we will call the benchmarks **multi-high**, **multi-low** (high-computation and low-computation variants of the multimedia benchmark, respectively) and **des**.

Since real-life MPSoCs are not likely to only feature general purpose processing cores, we also deploy traffic generators (T0 to T4 in Figure 6.5 on page 154) to model additional hardware IP blocks which may be present in the platform. While this choice is not in any way supposed to model on-chip coprocessors in a general fashion, we feel that it adds extra realism. Therefore, we include two different types of traffic generation patterns: DSP-like (streams of accesses to a memory bank) and I/O controller-like (a rotating pattern of accesses towards neighbouring devices). DSP-like traffic generators are each programmed to fetch 128 or 256 bits of data from one of the shared memory banks or devices (indicated with S in Figure 6.5 on page 154), compute for 10 clock cycles, and repeat. I/O controller-like traffic generators are instead programmed to query three shared devices in a rotating pattern, by reading 256 bits from each. We program the generators to issue functional traffic such as data transactions for consistency reasons; adopting a lower-level approach, such as the injection of packets in the NoC, would make the comparison with AMBA very unintuitive.

On both ML AMBA and the NoCs, the location of the various devices within the topology is key to good performance. For example, private memories exhibit optimal latency only if located next to their master (*i.e.*, on the same AHB layer or attached to the same ×pipes switch). The placement of shared slaves must comply with functional constraints: for example, in a ML AMBA topology, shared slaves must be put beyond the crossbar component, otherwise only local masters will be able to access them.

6.3.3 Fabric Synthesis

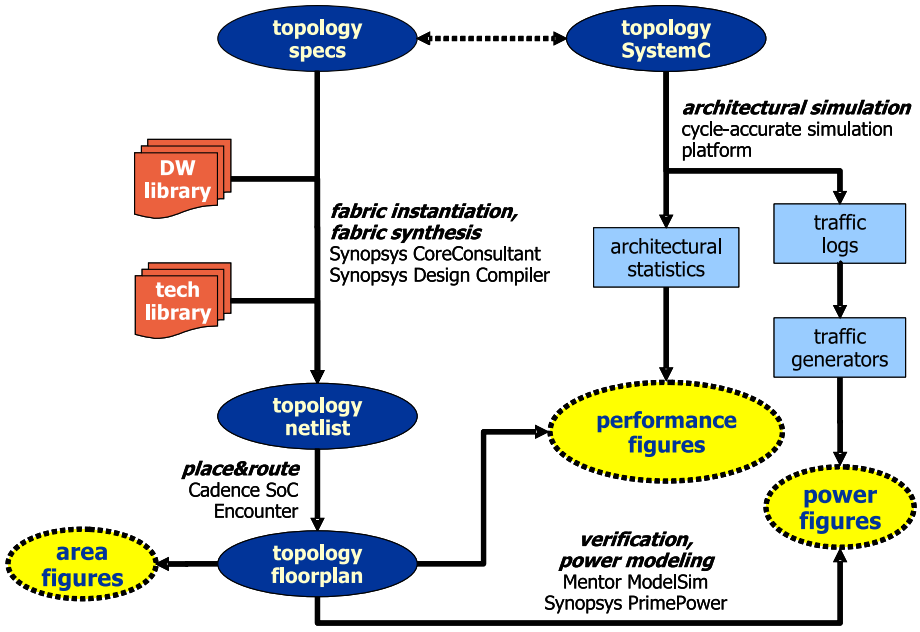
The tool flow used for this study, summarized in Figure 6.7 on the next page, closely resembles the flow described in the rest of this dissertation (Figure 1.5 on page 27, Section 6.2 on page 147). The main noteworthy difference is that the study is performed on a 130nm technology backend, and therefore leverages non-placement-aware synthesis tools. Also, Figure 6.7 on the facing page depicts the different steps required for the AMBA physical implementation, and provides some details on the simulation and power estimation infrastructure. In the following we will just discuss the main differences and additions with respect to what we have already presented.

AMBA synthesis (Figure 6.7(a) on the next page) is performed by using the Synopsys CoreAssembler [222] tool to instantiate the IP cores included in the Synopsys AMBA DesignWare libraries, therefore composing the needed topologies. During this phase, design parameters (such as data lane width) and constraints can be defined. The final result is a low-level HDL netlist composed of technology library standard cells representing the interconnect fabric, with AMBA AHB masters and slaves instantiated as black boxes.

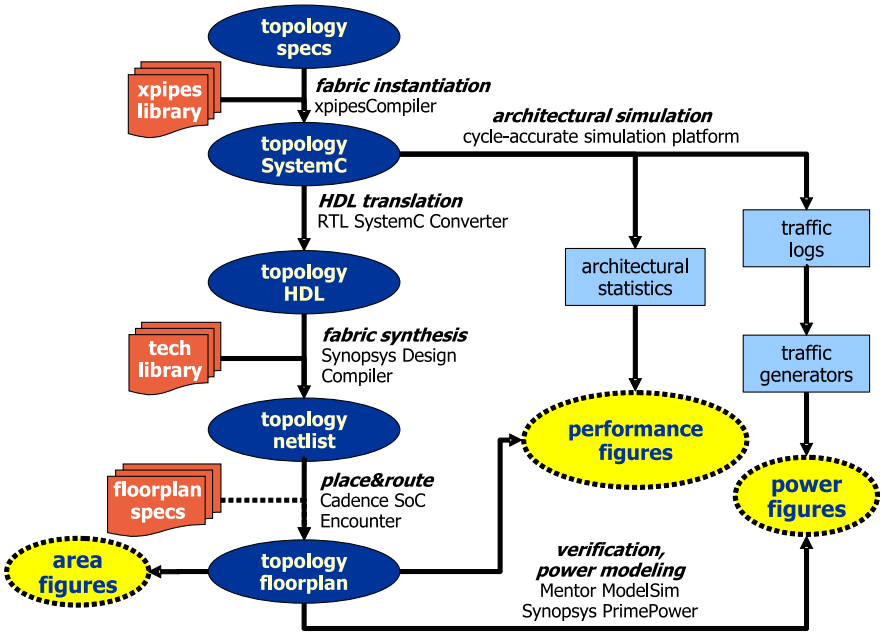
×pipes is, in this study, synthesized with Synopsys Design Compiler [223]. We instruct Design Compiler to save power when buffers are inactive by applying clock gating to the NoC blocks. If the target technology library features dedicated clock gating cells, they may be used; in our case such devices are not available, therefore we let Design Compiler implement the gating circuit by means of generic cells. This incurs a small penalty in operating frequency and power consumption, that could be avoided with a more complete technology library.

For placement&routing, which we perform with Cadence SoC Encounter [217] for all fabrics, we specify hard macros of 1 mm^2 representing cores and memories. We let the tool use over-the-cell routing, *i.e.* to route wires on top of the IP cores, which only obstruct the five bottom metal layers. Out of the eight metal layers that our technology library allows, only the top three are used for AMBA or ×pipes routing.

Post-layout verification and power estimation is achieved as follows. First, the final placed&routed topology is simulated by injecting functional traffic through the AHB (respectively, OCP) ports. This simulation is aimed both at verifying functionality and at collecting a switching activity report. At this point, accurate wire capacitance and resistance information, as backannotated from the placed&routed layout, is combined with the switching activity report using Synopsys PrimePower [224]. The



(a) AMBA.



(b) ×pipes.

Figure 6.7: The physical implementation flow for the fabrics under test.

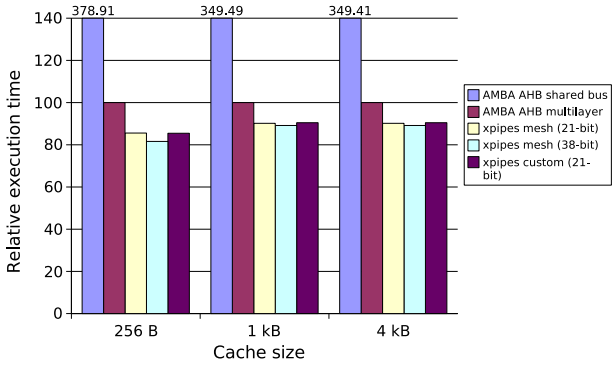
output is a layout-aware power/energy estimation of the simulation.

6.3.4 Cross-Benchmarking Results

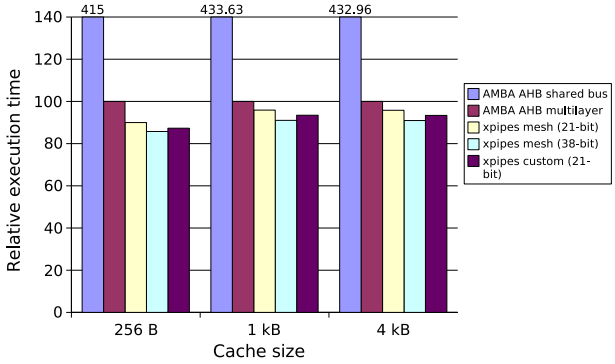
Interconnect Performance

First, we perform a cycle-accurate architectural simulation of the alternative topologies under the load of three benchmarks, as described in Section 6.3.2 on page 155, to assess their performance. The plots in Figure 6.8 on the next page summarize the global results. The vertical axis represents the relative benchmark execution time, taking as the baseline the execution on the multilayer AMBA AHB topology. Execution times are computed by first running an architectural simulation, which provides results in terms of clock cycles, and then by backannotating the clock period as resulting from the synthesis flow, as discussed in Section 6.3.3 on page 158. Frequency results will be discussed in more detail in Section 6.3.4 on page 165, but let us anticipate that we achieve 370 MHz for the AMBA topologies and 793 MHz for the NoC topologies. We realistically assume that ARM7 cores should be able to run up to a frequency of 400-500 MHz in 130nm lithography. Since the ML AMBA topology is capable of achieving 370 MHz at most, and the overhead for the usage of dual clock synchronization FIFOs would not be justified in this case, we assume the system to be fully synchronous at 370 MHz. For the NoC, we exploit the dual clock support of ×pipes to run the cores at 396.5 MHz and the NoC at its maximum frequency of 793 MHz. The 7% frequency boost we give to the cores is small and does not represent an unfair advantage; in fact, it is a byproduct of the high clock frequency achievable by NoCs, a feature that must be exploited as much as possible by NoC designers. We repeat the tests with three different cache sizes for the ARM7 processors; smaller caches translate into more cache misses and more congestion on the fabric. The ARM7 caches have 128-bit lines and feature a write-through policy.

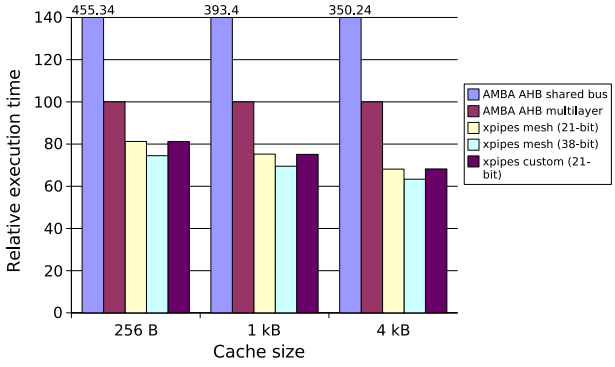
In all benchmarks, the shared bus is completely saturated and execution times are unreasonable - about four times larger than with the other interconnects. In **multi-high** and **multi-low**, the 21-bit NoC mesh exhibits a small but noticeable advantage with respect to the ML AMBA fabric of about 5% to 15%, with the largest gain being achieved in high-congestion (small cache) setups. The 38-bit NoC extends the gap to the 10% to 20% range. Even the custom NoC topology, which features much less bandwidth than the meshes, typically performs 10% better than ML AMBA. The figures represent overall benchmark execution time, therefore such improvements are remarkable. Finally, the **des** benchmark is strongly



(a) **multi-high**.



(b) **multi-low**



(c) **des**.

Figure 6.8: Relative execution times for three benchmarks for varying cache sizes. The ML AMBA AHB execution time is the baseline at 100. AMBA AHB shared bus results lay beyond the upper limit of the Y axis scale.

more bandwidth-intensive than **multi-high** and **multi-low**, and therefore the gap among AMBA and the NoC interconnects widens to the 20 to 35% range.

The results do not show a large difference between the **multi-high** and **multi-low** applications; in both cases, the gap between AMBA and NoC topologies is similar. At first sight, the benchmark with more communication and less computation demands (**multi-low**) should give a larger advantage to bandwidth-rich interconnects, such as the NoCs. The results can be better understood by noticing two things.

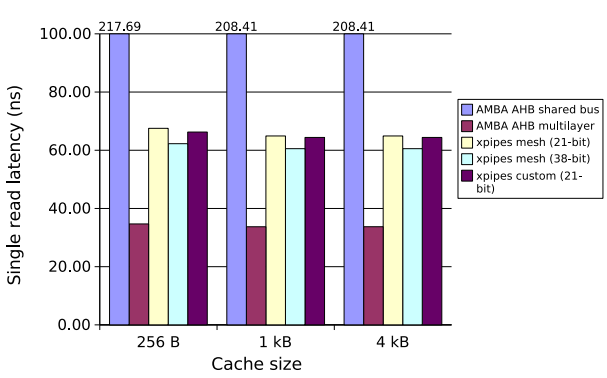
The first is that, despite a difference of a factor of eight in the performed computation, and all else being equal, this 8X difference is only translating into a gap of about 20% in the ratio of computation to *actual* communication. This happens because communication bandwidth is required not only for explicit data transfers, but also implicitly to fetch computation inputs (cache misses) and to store computation outputs (cache write-backs or write-throughs). For example, on the 21-bit NoC mesh with 1 kB caches, we observe that in the **multi-low** case, computation is typically 50 to 60% of the application kernel's simulation time, while in **multi-high** the fraction of computation time ranges between 70% to 85% - obviously larger, but less so than what could be expected.

The second key to understanding the behaviour of **multi-high** *vs.* **multi-low**, and more in general the reasons for the performance advantage of NoCs, is the difference between bandwidth and latency.

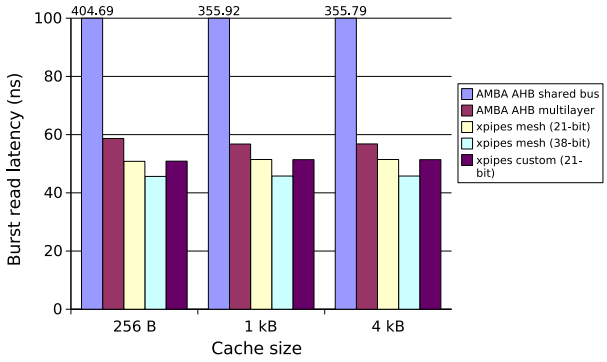
One important factor that contributes to the NoC speed results is certainly their peak bandwidth. Given the clock frequencies above, the overall bandwidth of the fabrics can be calculated. The NoC meshes have 44 links, for an aggregate bandwidth of about 87 GB/s (21-bit mesh) or even 158 GB/s (38-bit flits). The custom topology, which is specifically optimized, only features 14 links and therefore has around 28 GB/s of bandwidth. The ML AMBA topology can have at most five pending transactions at a time; considering two sets of 32-bit data wires (AMBA features dual data channels for reads and writes), 32-bit address wires and a dozen used control wires, the available bandwidth can be computed to be around 24 GB/s.

Therefore, the NoC meshes feature about 3.5 to 6.5 times more peak bandwidth than the ML AMBA topology. This seems to explain the performance gap. However, the NoC custom topology still outperforms ML AMBA with just 28 GB/s of peak bandwidth - a 15% margin.

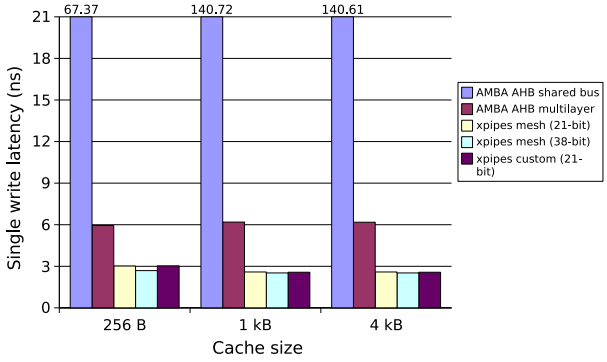
This is due to the fact that bandwidth is only an indirect clue of performance; the real metric to assess the speed of an interconnect is the latency from request to completion of a transaction, defined as the time needed to



(a) Single reads.



(b) Burst reads.



(c) Single writes.

Figure 6.9: Latency of different transfers on the interconnects. Latency measured between the issue of the transfer request and the availability of a response (for reads), or between the issue of the transfer request and the request acceptance (for writes). AMBA AHB shared bus results lay beyond the upper limit of the Y axis scale

let the requesting core resume its execution. In Figure 6.9 on the preceding page, we depict the average latencies for various types of transactions in **multi-high** (other benchmarks show similar trends) as seen by the ARM7 cores. For single reads, the packeting overhead of the NoC is clear; the ML AMBA topology is about twice as fast in returning responses. Indeed, we infer from the protocol and empirically observe that the ML AMBA design can internally complete a single read in 5 CPU clock cycles in the best case (with one-wait-state memories), while 10 CPU clock cycles are needed for the NoC in the 38-bit configuration with a 2X clock multiplier. The same does not hold for burst transfers, where the packeting overhead is only paid once, and congestion becomes the key limiter: even though the burst traffic in our case is mostly composed of short 4-beat cache refills (the traffic generators inject a smaller amount of 8-beat reads), the NoCs come out faster, with a margin of 10 to 20% (the 38-bit topology performing even better at 20 to 25%, due to lower congestion). This result strongly suggests that NoCs should try to take advantage of stream transfers. The single write latency figure is also interesting; in this case, the NoC shows, on average, less than half the latency of the ML AMBA scheme. This figure is the result of the support for posted writes in the OCP protocol, which is exploited by ×pipes. ×pipes allows streams of writes to be issued in a posted fashion, without any delays except those possibly introduced by eventual buffer saturation somewhere in the network. In contrast, the AMBA protocol forces a master to wait for the response to the previous write request and for rearbitration before issuing a new write; therefore, write streams experience continuous hiccups. This phenomenon could be bypassed by interposing data FIFOs, but this kind of optimizations is beyond the scope of this discussion.

The overwhelming bandwidth advantage of the NoC meshes is a hint to overdesign, and explains our choice of presenting a custom NoC topology that is specifically tailored for the benchmarks under scrutiny. The purpose is not to contrast this topology against the AMBA fabric, which would be unfair, but to show how significant the savings that derive from custom mapping can be. The custom topology is much less bandwidth-rich than the meshes, noticeably trimming power consumption while not affecting fabric speed nearly as much.

Coming back to the analysis of benchmark execution times, as shown above, our results show similar performance gains of NoCs *vs.* AMBA in **multi-low** and **multi-high**, despite the fact that **multi-low** spends about 20% more time in communication than **multi-high**, and should therefore exhibit larger speedups. On the other hand, **des** requires heavy communication resources and indeed strongly benefits from NoCs. Given the dis-

cussion above, the mix of transaction types can clearly explain the results. So, for example, let us consider the **multi-high** benchmark when run on the 21-bit mesh with 4 kB caches. We observe 26% of single reads, 1% of burst reads (very few cache misses) and 73% of single writes. In the same setup, **multi-low** exhibits 46%, 2% and 52%. So, while **multi-low** spends more time in communication, its communication mix is less favourable to the NoCs than that of **multi-high**, producing similar overall results. **des** not only demands a lot from the interconnect, but it is also a good match to NoC architectures; due to a much larger data set and code segment, in the same setup, we notice that the ratios are respectively 16% (few single reads), 32% (many refills) and 52%.

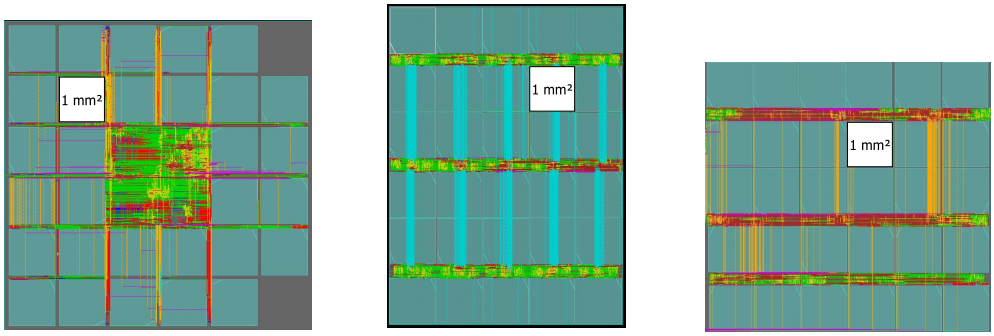
Many factors can contribute to performance results, including caching schemes (as the plots show), functional bottlenecks (one slave is heavily accessed and slows down the whole system), localized congestion (the topology suffers from overload at some location), traffic spikes over time (resources are normally underutilized, but communication spikes occur and when they occur they are poorly handled). A discussion of all these issues is beyond the scope of this dissertation. Overall, we think that our results show a noticeable performance lead of NoCs over a wide range of transaction patterns.

Interconnect Area, Frequency of Operation and Bandwidth

Screenshots of the layouts for ML AMBA AHB and for the NoC topologies are shown in Figure 6.10 on the next page. Here and elsewhere in the following paragraphs, the shared bus configuration is omitted because, as we have shown (Section 6.3.4 on page 160), the performance of the fabric is so bad as to make any comparison pointless.

We begin our analysis with area occupation of the topologies under test. As a premise, we must state that our study mostly focuses on performance and power. Thus, we do not perform any specific optimization in the synthesis flow to minimize area requirements; on the contrary, we configure the tools for maximum frequency, without area constraints. Further, we perform a placement step to derive fabric floorplans, but this is only done to get a realistic estimation of capacitive overheads due to long wires; we omit the step of tightly compacting the design, which would be needed in an industrial product but is unneeded for our characterization. For these reasons, the overall floorplan areas which can be inferred from Figure 6.10 on the next page are not meaningful, except as a way to get information about wire lengths.

Still, the cell area metric, which only takes into account the area occu-



(a) ML AMBA. (b) ×pipes meshes. (c) ×pipes custom.

Figure 6.10: Layouts of the test fabrics.

Max Frequency	ML AMBA	×pipes mesh (21-bit)	×pipes mesh (38-bit)	×pipes custom (21-bit)
After synthesis	480 MHz	847 MHz	847 MHz	847 MHz
After place&route	370 MHz	793 MHz	793 MHz	793 MHz
Frequency drop	22.9%	6.4%	6.4%	6.4%

Table 6.1: Pre- and post-placement achievable frequencies.

pation of logic cells in the design, is a useful indication about the expected silicon overhead of alternative fabrics. After placement&routing, and including the clock tree buffers, the cell area for the AMBA ML topology is 0.52 mm^2. For the ×pipes meshes, which feature 15 switches and 30 NIs, area is 1.7 mm^2 (21-bit design) to 2.1 mm^2 (38-bit design), while the custom topology comprises fewer switches and is therefore a bit less demanding at about 1.5 mm^2. While these results show a large relative overhead for the NoCs *vs.* AMBA, the overhead is in fact small when compared to the area requirements of the IP cores.

The maximum frequency results for the fabrics are as reported in Table 6.1. The ML AMBA fabric reaches at most 480 MHz before the placement stage. After placement and clock tree insertion, the actual achievable frequency decreases sharply to 370 MHz (-22.9%). This drop means that, compared to the design netlist, some unexpected capacitive loads arise in the final floorplan due to routing constraints. An explanation can be found in the purely combinational nature of the fabric, which implies long wire propagation times and compounds the delay of the crossbar block.

As can be seen, the ×pipes topologies all achieve much higher clock

frequencies. Even after placement&routing, the critical path is not on the NoC links, which confirms that the wire segmentation is highly effective. A byproduct is wire load predictability; in fact, as the table shows, the NoC fabrics suffer a minimal timing penalty of 6.4% after taking into account actual capacitive loads. These results suggest better scalability of the NoC architecture to future technology nodes.

We would like to underline the effect that clock gating and clock tree deployment have on the design of a complex architecture. Compared to results [220] where these elements were not accounted for, it is for example possible to notice that the maximum frequency achievable by NoCs drops by almost 100 MHz (885 MHz *vs.* 793 MHz). This is easily explained; signals need to travel from flip-flop to flip-flop within a time budget of one clock period, but the clock management logic adds delay and skew, both of which cut into the available timing window. We feel that this result, while certainly not novel, is further highlighting the importance of a complete modeling and synthesis flow spanning up to the lowest levels of abstraction.

Interconnect Power and Energy

To attempt a power evaluation, we first monitor activity during functional system simulations and log all source-target transaction pairs. We then inject traffic from master ports towards each of the targets which are accessed in the functional simulation.

At the 130nm node, without clock gating, sequential logic represents by far the largest fraction of power consumption in the ×pipes NoC, with flip-flops burning as much as 80% of the global power requirements (still excluding the clock tree contribution, which, as we will show, is also major). While the power cost of switching activity on global wires is expected to become more prevalent in future technologies, at the 130nm node sequential elements seem to be the prime candidates for tuning. This observation leads us to attempting to optimize the NoC by means of several strategies related to buffering elements. First, the implementation of clock gating lets us achieve about 30% power savings. Second, we keep buffering resources to a minimum across the NoC, by sizing NI and switch buffers to hold only three flits at a time. Third, we explore the flit width degree of freedom, which proves very useful: moving from 38-bit to 21-bit flits reduces buffer size almost in half, cutting power by a significant amount (see below).

The power results that we achieve for the topologies at their maximum operating frequency are reported in Table 6.2 on the next page, while en-

Power Consumption	ML AMBA	×pipes mesh (21-bit)	×pipes mesh (38-bit)	×pipes custom (21-bit)
Global power	75 mW	376 mW	473 mW	347 mW
Sequential cell power	15 mW	145 mW	187 mW	116 mW
Sequential power ratio	20.5%	38.6%	39.5%	33.4%

Table 6.2: Power consumption of the fabrics.

Energy Consumption	ML AMBA	×pipes mesh (21-bit)	×pipes mesh (38-bit)	×pipes custom (21-bit)
Energy per injectable data	3.13 mJ/GB	4.32 mJ/GB	2.99 mJ/GB	12.39 mJ/GB
Energy per benchmark run (fabric only)	0.075 mJ	0.338 mJ	0.402 mJ	0.312 mJ
Energy per benchmark run (1W system)	1.08 mJ	1.30 mJ	1.31 mJ	1.28 mJ
Energy per benchmark run (5W system)	5.08 mJ	5.17 mJ	4.96 mJ	5.14 mJ

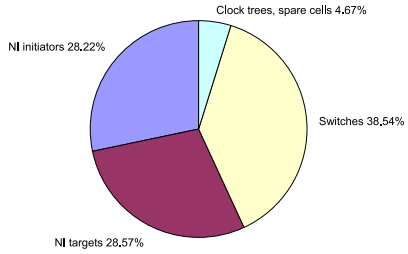
Table 6.3: Energy consumption of the fabrics.

ergy results are reported in Table 6.3. ×pipes figures are for designs with clock tree and clock gating, while in the case of AMBA, we only insert a clock tree; given the low amount of sequential logic that AMBA contains (see Table 6.2), clock gating would offer negligible benefits and unnecessarily add design flow complexity and frequency penalties.

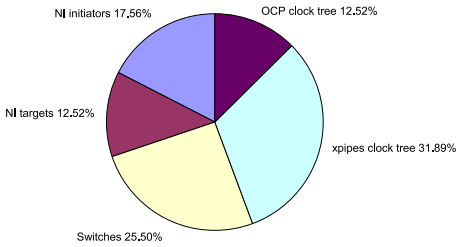
The ML AMBA crossbar is clearly the winner in terms of pure power consumption. The power consumption of the NoC meshes is 5.6 to 7.5 times higher. Keeping the flit width of the NoC mesh low is however helpful, as power savings of 25% can be noticed, with a much lower impact on overall performance (Section 6.3.4 on page 160). Thanks to clock gating, the fraction of power consumption due to sequential logic drops significantly, from an initial value of around 80% [220] to around 35%. This drop is due to the compound effect of clock gating (which cuts the sequen-

tial power requirements by 30%) and of the clock tree insertion, which is implemented by means of combinational cells, thus lowering the relative fraction of sequential power. The custom NoC topology, thanks both to its lower switch count, is able to shave about 8% off the power of the 21-bit mesh. When considering the power density of the interconnects, AMBA has an advantage of roughly a factor of two; we mostly attribute this to the difference in clock speeds.

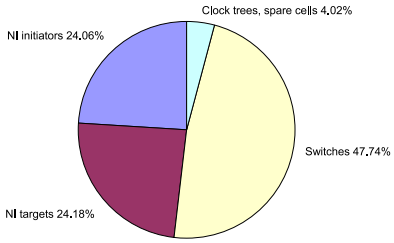
In terms of energy, the advantage of ML AMBA is less clear. When considering the ratio among power consumption and available bandwidth (mW over GB/s, or mJ over GB of injectable data), ML AMBA and the NoC meshes look much closer. However, this figure is a bit misleading; using all of the available bandwidth, the meshes would indeed consume much more energy. Further, the custom NoC, which is designed around providing bandwidth only where necessary, but utilizing it efficiently, is unreasonably penalized by this analysis. Therefore, we attempt to use a more meaningful metric: power over benchmark execution time, *i.e.* the energy required to complete a benchmark. Given the performance figures shown by our experiments, we conservatively assume an execution time advantage as shown by **multi-high** or **multi-low**; in a **des**-like scenario, of course, the results of NoCs would look better. Thus, we set an execution time gain of 10% for the 21-bit NoCs (mesh and custom) against the ML AMBA fabric, and of 15% in the case of the 38-bit NoC mesh. Calculating the energy consumption over an execution time which is 1 ms for the baseline ML AMBA case, we observe the results reported in the second row of Table 6.3 on the preceding page. To derive an even more useful metric, however, the energy consumption of the whole system should be taken into account. To this effect, the power consumption of other parts of the system must be modeled. This is a very difficult task, as it greatly depends on the specific components at hand. We could very conservatively assume a power consumption of just 1 W at 370 MHz for all of the 15 cores, caches and memory blocks. Further, we could assume a 370 MHz working frequency for the cores in the ML AMBA case and a 396.5 MHz frequency for the NoCs (Section 6.3.4 on page 160). The overall power consumption of the systems would therefore be 1.075 W for ML AMBA, 1.449 W for the 21-bit NoC, and so on. The corresponding energy is reported in the third row of Table 6.3 on the preceding page; the three NoCs are giving almost identical results, about 20% worse than ML AMBA. With system components requiring 5 W, however, the NoCs become strongly competitive, as the table shows; the 21-bit NoCs get almost on par with ML AMBA, while the 38-bit topology actually offsets its higher power requirements with its performance results, coming out as the most energy-efficient by a slight



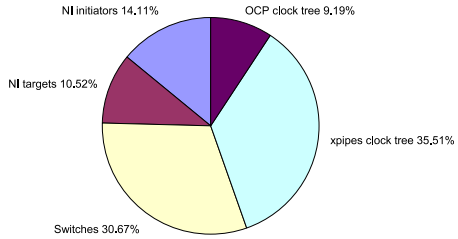
(a) Area of the 21-bit mesh.



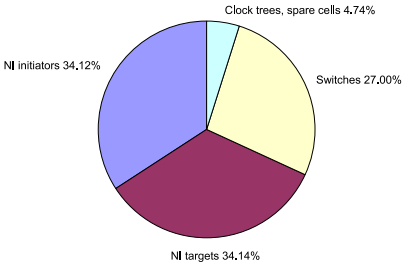
(b) Power of the 21-bit mesh.



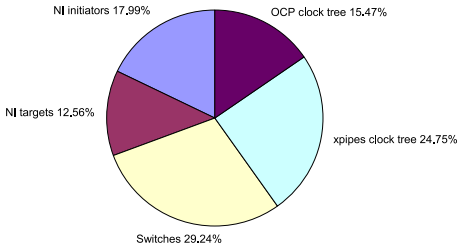
(c) Area of the 38-bit mesh.



(d) Power of the 38-bit mesh.



(e) Area of the 21-bit custom topology.



(f) Power of the 21-bit custom topology.

Figure 6.11: Split report for area and power of three ×pipes topologies.

margin.

Split Analysis of Area and Power Contributions

In Figure 6.11, we present a split report of area occupation and power consumption for the three NoCs. In terms of area (Figure 6.11(a) and following), at first glance, three main contributions can be noticed: switches, initiator NIs and target NIs. However, the ratios between them can shift in a relevant fashion. To understand the figures, please remember that the mesh topologies feature 15 instances of each type of component, while the custom NoC has 15 of each type of NIs but only 8 switches. The absolute contribution of the NIs to the NoC area remains roughly constant across the topologies; NI area is found to be dominated by the OCP front-end,

which remains unchanged regardless of the topologies. The main differences are therefore due to switch area. Taking the 21-bit mesh as a baseline, we measure switches to require 38.5% of the NoC area; we can notice that in the 38-bit mesh (where switches are larger due to the increased flit width) the percentage rises to 47.7%, while in the 21-bit custom topology (where there are fewer switches) it falls to 27.0%.

In terms of power (Figure 6.11(b) on the facing page and following), the major contribution, as expected, is due to the clock distribution network. Two clocks are actually being distributed, a fast one for the network and a slower one for the OCP front end of the NIs. The two clock trees together burn 40% or more of the overall power. We observe several interesting trends in this split analysis. For example, since the absolute power consumption of the NIs remains more or less constant across the topologies, their relative consumption is determined by the other components. The ×pipes clock tree, *i.e.* the fast one, has a consumption which is very directly correlated to the amount of flip-flops it must drive; therefore, it takes the smallest fraction of the power budget in the 21-bit custom topology (where there are fewer switches to clock), a larger one in the 21-bit mesh, and the largest one in the 38-bit mesh. Switches themselves exhibit a more complex trend. They already burn a significant amount of power in the 21-bit mesh, and this budget increases even more in the 38-bit mesh and in the custom topology. The reasons are different; in the former case, there are simply more gates (38-bit switches have datapaths which are almost twice as wide), while in the latter, the amount of gates is actually lower (8 switches instead of 15) but traffic is still efficiently processed, which points to a much higher switching activity.

Finally, it is interesting to note that, in all cases, when comparing the switches to the NIs, switches take a larger fraction of the power budget than they do of the area budget. For example, in the most extreme case, the custom topology, the switches require less area than either the initiator or target NIs, but burn as much power as both types of NIs together. We mostly attribute this fact to NI front ends working at the OCP clock frequency, *i.e.* at half the frequency of the rest of the network (Section 6.3.4 on page 160), while switches uniformly run at the higher frequency. Further, switches experience a higher average activity level, since for example a single incoming packet forces all output ports of a switch to evaluate a new request - even if a single output port will eventually let flits through.

6.4

NoC Area and Power Modeling

In this section, we focus on devising area and power models for the NoC switches. These models, as seen in Section 5.2 on page 121, are essential for (i) getting a better understanding of the NoC optimization opportunities, (ii) the effectiveness of SunFloor. As discussed, accurate switch models are more useful than NI models, since the latter do not have any relevant impact in devising the optimal NoC topology for a given application.

We propose a NoC switch modeling methodology which takes advantage of the designer's knowledge of the target architecture and synthesis library. It is of course impossible to devise an accurate yet fully generic model for the hardware cost, in power and area, of any given NoC. Our focus is instead on how such a model can be built for a specific NoC instance. Key properties of our approach include accuracy and explicit modeling on several parameters of the design, like switch cardinality, flit width, buffering, traffic and synthesis parameters. These properties make the approach suitable for fast exploration of large parts of the fabric design space, flexible and applicable in real life, for example by accounting for the behaviour of the synthesis tools when the target operating frequency approaches the limits of the design. Model coefficients can be made even more accurate by using a placed and routed training set for characterization, albeit at a modeling effort cost.

6.4.1 Proposed Modeling Methodology

Our modeling activity starts from an existing RTL description of the NoC components, and is composed of five main phases.

1. We devise a set of parameters that are relevant to the accuracy of any model which aims at practical applicability.

2. We define a general model formula for area and for power, relying on the knowledge of the target switch architecture.

3. We synthesize several configurations (*training set*) of the target switch architecture, and measure the corresponding area and power consumption. The configurations are chosen so as to uniformly but sparsely cover the design space of interest, therefore allowing for an accurate yet quick construction of the model. The synthesis process can optionally include the placement&routing step for maximum thoroughness of the assessment.

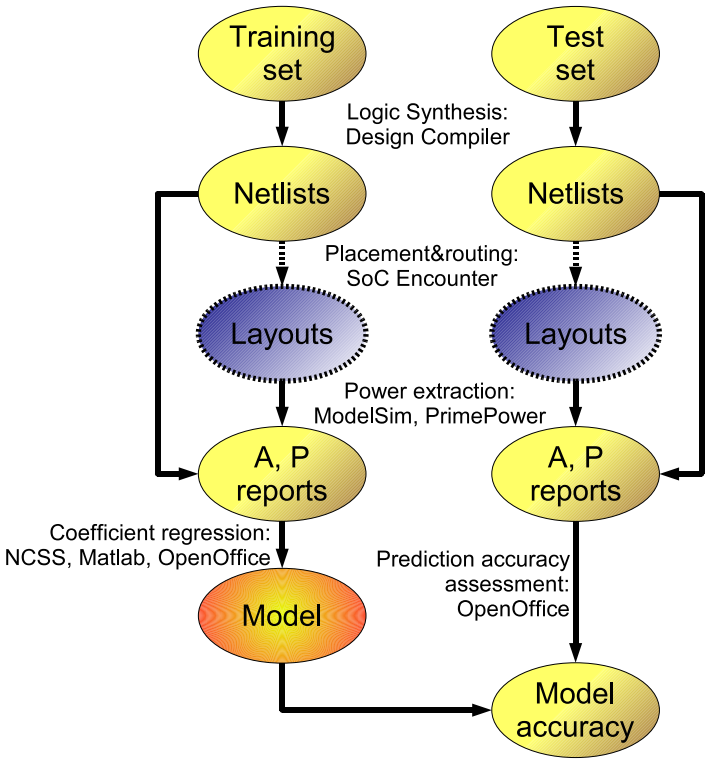


Figure 6.12: Outline of our characterization activity. The placement and routing step is optional both for the training and the test set.

4. We use experimental results to numerically quantify the coefficients of the model. As outlined later, we propose two different ways of performing this step, with varying accuracy/effort tradeoffs.

5. We assess the quality of our models against configurations (*test set*) outside of the training set.

The first four steps will be covered in Section 6.4.2 through Section 6.4.6 on page 184. An outline of how we handle steps 3 to 5 is provided in Figure 6.12.

6.4.2 Parameters of Interest

A key phase of the approach is devising a model that matches the architecture under consideration and its properties. However, considering the architecture alone does not guarantee that the model will be applicable and accurate enough in practice. For example, synthesis tools play a pri-

mary role in defining the area and power efficiency of a component. Therefore, we first summarize the parameters of interest when assembling our model. Without loss of generality, we analyze the ACK/NACK implementation of ×pipes; equivalent models can be devised for STALL/GO, just by accounting for the differences in buffering and possible traffic conditions. In fact, these models would be simpler due to the lower number of possible transmission states in STALL/GO.

Architectural Parameters

The main parameters are:

- Switch cardinality (radix, number of ports). To account for rectangular switches, we separately consider the amount of input ports (np_i) and output ports (np_o).
- Amount of buffering devoted to flow control handling and performance optimization, also called buffer depth (bd) (expressed in terms of single-flit buffering elements).
- Number of bits of the incoming and outgoing elementary data blocks, also called flit width (fw).

Implementation Flow Parameters

It is possible to tune synthesis tools, among other things, for:

- Target frequency of operation.
- Target area.
- Target power consumption.

Tuning these parameters differently in the synthesis tools yields, as expected, a widely different quality of results. For example, when performance demands are extreme, synthesis tools are forced to create netlists containing large amounts of buffers and fast gates, which are not area- and power-efficient. To mimic a typical industrial flow, where an application performance constraint must be satisfied, we impose as the primary objective a certain target operating frequency (which is a parameter of our model), while area and power minimization are given to the tool as secondary optimization objectives. As a result, area and power requirements,

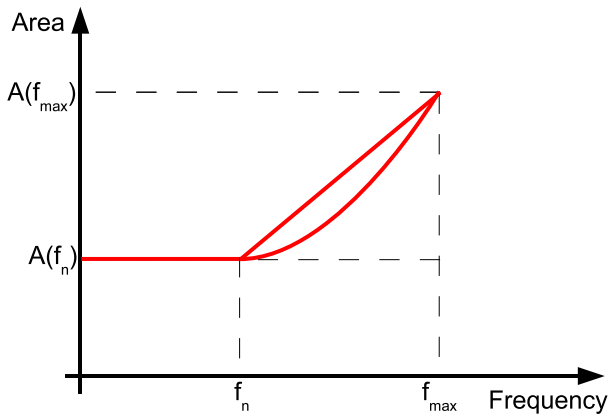


Figure 6.13: Area requirements *vs.* target operating frequency.

expressed as a function of the target operating frequency, exhibit a characteristic flat behaviour followed by a steeply rising trend after an inflection point. This trend is well known [225], and can be explained by the fact that, above some target operating frequency which can be achieved with minimal circuitry, synthesis backends are forced to insert extra gates to comply with increasing performance demands.

Figure 6.13 shows a linearized and a parabolic approximation of this trend, and at the same time summarizes the ways we modeled this effect. For each device configuration (*e.g.* 4x4 32-bit switches with 6-deep FIFO buffers), a "native" frequency f_n can be identified. This frequency is that achieved by the synthesizer with relaxed timing constraints. Under this condition, the tool is free to fully pursue its secondary objectives, hence creating minimum area ($A(f_n)$) and power ($P(f_n)$) netlists. Configuring the tools for target frequencies lower than f_n does not result in further decreases of area or power dissipation. For each switch instance, it is also possible to find a frequency f_{max}, that corresponds to the fastest achievable synthesis result. Under this timing constraint, the module has $A(f_{max})$ area and $P(f_{max})$ power consumption. We approximate the dependency of area and power overheads as linear or parabolic in the range $(f_n; f_{max})$. This assumption allows us to characterize devices only twice, at f_n and f_{max} (under various combinations of the other architectural parameters), while being able to estimate results over the whole range of frequencies achievable by the module. Since this analysis is not correlated to other model parameters, in the following, for simplicity of notation, we will not explicitly mention the dependency of coefficients on the synthesis target frequency; the characterization of this parameter will be implicitly

assumed.

The linearized or parabolic approximation is a way of abstracting away from low-level details of the logic synthesis process, which are impossible to capture in a high-level model. The experimental results that will be shown in Section 6.4.6 on page 184 will be based on a test set which is also spread in terms of target operating frequency, therefore providing a metric of the accuracy of such a model. Section 6.4.6 on page 191 will compare the accuracy of the linear *vs.* the parabolic models.

Please note that developing area and power models which are a function of the target frequency of operation up to f_{max} also implies making available a model of the timing properties of the switches.

Traffic Condition Parameters

These parameters are only relevant to power models, since area models are clearly static. They include downstream congestion and internal congestion (*i.e.* arbitration conflicts). They will be explained in more detail in Section 6.4.3 on the next page.

6.4.3 Area and Power Models

Area Model

In general, the area equation must be of the form of Equation (6.1):

$$A = f(bd, fw, np_o, np_i) \tag{6.1}$$

We identify as suitable the area model expressed in Equation (6.2):

$$\begin{gathered} A(fw, bd, np) = A_1 \cdot np_o \cdot fw \cdot bd + \\ + A_2 \cdot np_i \cdot fw + A_3 \cdot np_o \cdot np_i + A_4 \cdot fw \cdot np_o \cdot np_i \end{gathered} \tag{6.2}$$

The rationale of this formula is that the area of the target switch can be rendered as the sum of four contributions (Section 4.3.2 on page 100): (i) output buffers, (ii) input buffers, (iii) arbitration and flow control logic, (iv) crossbar. Each contribution strongly depends on a known combination of architectural parameters:

- Output buffers, which are dominated by flip-flop area, can be supposed to depend linearly on flit width fw and buffer depth bd (×pipes switches are output-buffered), which respectively represent the width and depth of the buffer (Figure 6.14 on the next page). There are np_o such buffers.

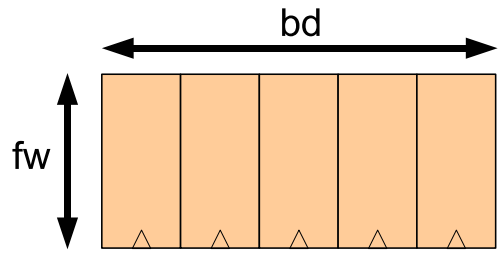


Figure 6.14: Dependency of the output buffer area on fw, bd.

- Input buffers are similar to the case above, but since they have a constant depth, they do not depend on bd. Obviously np_i is used in place of np_o.
- Since a distributed arbitration technique is used in the target switch, one arbiter is instantiated at each output port. Each arbiter has a complexity proportional to the number of candidate input ports np_i, therefore the overall contribution is the product of the input and output cardinalities. The arbiter logic is clearly independent of datapath parameters such as flit width and buffer depth.
- The area overhead due to the crossbar must have a linear dependency on flit width, must be independent of the buffering resources and must have a linear dependency on the product of input and output cardinalities.

Power Model

The power consumption of a module depends on the switching activity of the cells, so, to express the power consumption of a NoC switch, a term that accounts for traffic conditions must be present. The most general way to model the power consumption thus becomes:

$$P = f(bd, fw, np_o, np_i, T) \tag{6.3}$$

with T being a generic variable that summarizes the traffic conditions. Since sequential components exhibit a power consumption even if they are not performing computation, due to the clock switching, a static (traffic-independent) term must appear. After analyzing the possible traffic flows in the ×pipes router, we propose Equation (6.4) on the following page as a general power model:

$$\begin{array}{c} P(bd, fw, np_o, np_i, T) = P_A(...)+ \\ +\sum_{j=1}^{np_o}[P_B(...) \cdot T_{Oj}]+ \\ +\sum_{j=1}^{np_o}[P_C(...) \cdot T_{OCj}]+ \\ +\sum_{j=1}^{np_i}[P_D(...) \cdot T_{ICj}] \end{array} \quad (6.4)$$

where the dots express dependencies on bd, fw, np_o, np_i which will be analyzed in more depth in the following. The first term models the power dissipated by inactive, but still clocked, registers. The remaining terms depend on traffic conditions. An accurate representation of the traffic conditions requires a separate analysis of the state of each input and output port. Therefore, we define np_o traffic variables T_{Oj} and T_{OCj}, to model the lack or presence of external congestion, and np_i traffic variables T_{ICj}, to model internal contention for resources. More specifically, we define:

- T_{Oj}: Percentage of time during which the output port j is successfully transmitting flits. This coefficient models traffic in absence of congestion.
- T_{OCj}: Percentage of time during which the output port j is trying to transmit, but flits are rejected. This coefficient models external congestion due to traffic spikes.
- T_{ICj}: Percentage of time during which the input port j of the switch is trying to transmit flits through one of the output ports, but arbitration is denied by the switch logic. This coefficient models the contention for the same output port inside of the switch.

This set of traffic percentages is linearly independent, since the complex arbitration and flow control patterns within a NoC switch make it very easy for some of these time windows to overlap. Please consider the following:

Example 5 *A 4x2 switch (see Figure 6.15 on the next page) may feature one established input-to-output connection where traffic is freely flowing (which is expressed by the condition T_{O1}), another established input-to-output connection which is stuck due to congestion in the downstream switch (modeled within T_{OC2}), while the third input port is unsuccessfully trying to transmit to one of the two output ports, which in this example are already busy (T_{IC3}), and the fourth is simply idle (this contribution is therefore included in the coefficient P_A).*

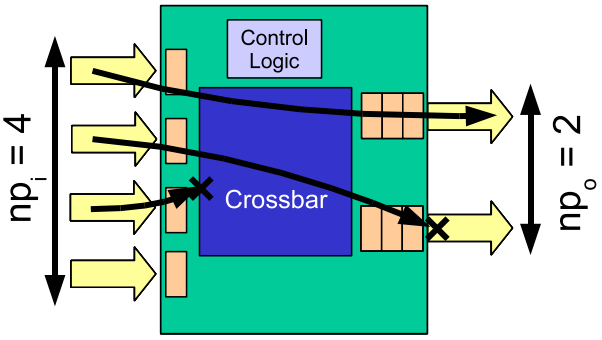


Figure 6.15: Example traffic in a 4x2 switch.

The coefficients P_A, P_B, P_C, P_D depend on architectural parameters, as for the area model. They account for the power consumption in the traffic states described above, as follows:

- P_A accounts for the static power dissipated by the switch and it is due to the non-combinational logic in the design. Therefore, it simply depends linearly on the number of flip-flops in the design, which are:
  - input buffers
  - output buffers
  - state registers in the control logic

  whose dependencies on architectural parameters are summarized in Table 6.4.

Contribution	**Dep. on** fw	**Dep. on** bd	**Dep. on** np_i	**Dep. on** np_o
output buffering	linear	linear	none	linear
input buffering	linear	none	linear	none
spare registers	none	none	linear	linear

Table 6.4: Dependency on architectural parameters of the static power coefficient P_A.

- P_B accounts for the dynamic power dissipated by flowing packet streams, due to the enabled registers and to the switching activity of combinational logic. We identify four contributions to the power dissipation:

  - output buffers. In these buffers, during every cycle, one of the flit registers (fw bits wide) samples a new piece of data; a $bd \times 1$ multiplexer then brings a flit to the output port. Therefore, this contribution is itself the sum of two terms.
  - input buffers
  - control logic
  - selected crossbar branch

  The dependencies of these contributions on the architectural parameters are summarized in Table 6.5.

Contribution	Dep. on fw	Dep. on bd	Dep. on np_i	Dep. on np_o
output buffer (reg.)	linear	none	none	none
output buffer (mux)	linear	linear	none	none
input buffer	linear	none	none	none
control logic	none	none	linear	none
crossbar branch	linear	none	linear	none

Table 6.5: Dependency on architectural parameters of the dynamic power coefficients P_B, P_C.

- P_C accounts for the dynamic power dissipated by the switch under a scenario where downstream congestion is preventing a free flow of packets. Although numerically different, the P_C coefficient is similar to P_B, in that it still involves an established input-to-output channel, and therefore its dependency on architectural parameters is the same (see Table 6.5).

- P_D accounts for the power dissipated by the switch when an incoming stream requires the access to an output port, but the arbitration is denied. The contributions to this portion of the power consumption are related to the following logic blocks:

  - input buffers
  - control logic

  We can summarize the dependencies on this contributions as shown in Table 6.6 on the next page.

Contribution	Dep. on fw	Dep. on bd	Dep. on np_i	Dep. on np_o
input buffer	linear	none	none	none
control logic	none	none	linear	linear

Table 6.6: Dependency on architectural parameters of the dynamic power coefficient P_D

Model Coefficient	Dep. on fw	Dep. on bd	Dep. on np_i	Dep. on np_o
P_A	linear	linear	linear	linear
P_B	linear	linear	linear	none
P_C	linear	linear	linear	none
P_D	linear	none	linear	linear

Table 6.7: Dependency of power coefficients on architectural parameters.

The dependencies of the power coefficients are thus summarized in Table 6.7.

We would like to stress that some coefficients, which could be intuitively expected to quadratically depend on parameters, are instead linearly dependent, because they characterize a single input or output port. The quadratic behaviour is indirectly restored by the summation symbols in Equation (6.4) on page 178.

6.4.4 Choice of a Relevant Training Set

To characterize the coefficients of our area and power models, we define a training set, composed of switch configurations chosen in such a way as to uniformly cover the relevant design space for the particular NoC under study. In the case of the ×pipes NoC, which is focused on the highest customizability of topologies, we choose to study a design space spanning over a large variety of cardinalities (np_i and np_o of 4, 10, 16 and 20). Since ×pipes is also focused on the best performance/overhead tradeoff point, and therefore on low hardware cost, we consider moderate buffer depths bd of 5 and 7 FIFO locations and flit widths fw of 21, 28 and 38 bits.

In the modeling approach called Full Factorial Design, all the possible permutations of the values of the independent design parameters should be studied to create the training set. This is often impractical due to the

quick rise in the number of instances as soon as new design knobs are added, leading to approaches to select only a subspace of the characterization set (Fractional Factorial Design [226]). In our case, based on the knowledge of the target architecture, we choose a very simple way of pruning the training set. The rationale is based on the observation that rectangular switches add a smaller amount of information to the training set; for example, when studying the power consumption, a rectangular switch is by design unable to simultaneously feature traffic flows on all of its input and output ports (see Figure 6.15 on page 179), and is therefore behaving similarly to a square switch of smaller cardinality. Our preliminary internal testing confirms this property, at least for the ×pipes NoC. Therefore, we simply choose to coalesce the np_i and np_o axes for the generation of the training set, and only include 4x4, 10x10, 16x16 and 20x20 instances.

We finally permutate all the possible parameter values, resulting in 24 (4 cardinalities times 2 buffer depths times 3 flit widths) configurations being synthesized.

6.4.5 Fitting Model Coefficients

Fitting Area Model Coefficients

To estimate A_1, A_2, A_3, A_4, we propose two different methods:

- Methodology 1: Coefficients can be derived directly from synthesis reports, which hierarchically list every switch sub-block. For example, once the area cost of an output buffer which is bd_0 flits deep and fw_0 bits wide is gathered from one report, it can be called $A_{obuf}|_{bd_0,fw_0}$. Since A_1 is expected to increase linearly with both bd and fw, it can be approximately derived as in Equation (6.5):

$$A_1 = \frac{A_{obuf}|_{bd_0,fw_0}}{bd_0 \cdot fw_0} \tag{6.5}$$

  Other coefficients can be similarly computed.

  Advantages: With this methodology, each contribution in the formula keeps a strict physical meaning. Only one synthesis run is needed to extrapolate coefficients for any switch instance; we arbitrarily choose a 10x10, 28-bit switch as a reference. This instance is close to the center of the design space of interest (see the previous paragraphs); its choice will be further discussed in Section 6.4.6 on page 184.

Disadvantages: This simplified approximation discards any constant offset that may be present in the coefficients. Further, the nature of synthesis tools introduces unpredictable fluctuations in the netlist area and power trends under different architectural configurations. This noise does not have any easily characterizable property. Thus, the model incurs a non-negligible error when compared against actual switch instances. Moreover, the choice of the specific switch instance for characterization might skew the computed coefficient values.

- Methodology 2: Coefficients can be derived by leveraging the multivariate non-linear regression algorithms natively provided by several mathematical and statistical packages. In this case, the input is a set of characterization syntheses (the training set described in the previous Subsection). The target polynomial for the regression is chosen based on insight of the dependency of area on the architectural design parameters (see Equation (6.2) on page 176).

  Advantages: The model fits actual synthesis results better.

  Disadvantages: Longer characterization time; with a thorough characterization set like that chosen in Section 6.4.4 on page 181, experiments must be performed in 24 device instances, against just one. The actual improvement in accuracy depends on the smoothness of the native behaviour of the synthesis tools. Some coefficients may lose their physical meaning (*e.g.*, they may become negative).

Both methodologies can be readily adapted to any parameterizable NoC architecture.

Fitting Power Model Coefficients

To characterize the P_A, P_B, P_C, P_D coefficients, we first inject traffic into the switch netlists under test, one at a time. This is achieved by ModelSim [227] simulation of the Verilog netlists (please refer to Figure 6.12 on page 173), to which traffic generators are attached. The traffic generators are configured to inject into the switch one of the four patterns described above (idle, free flow, downstream congestion, internal contention) at a time. The switching activity is logged and fed as an input to Synopsys PrimePower [224], which provides a hierarchical report of the power consumption of the switch sub-blocks. For each netlist of the training set, four hierarchical reports are therefore generated.

At this point, the power model coefficients are determined by using either of the techniques just outlined for the area models. The P_A, P_B, P_C, P_D scenarios are separately accounted for; the fitting polynomials are directly derived from Table 6.7 on page 181. For each of them, either by direct derivation or by non-linear regression, we extract the coefficients modeling the dependency on architectural parameters.

6.4.6 Experimental Results

We run the proposed characterization and modeling flow on a 130nm technology library. Therefore, we leverage Synopsys Design Compiler [223] as a logic synthesis tool.

To evaluate the accuracy of the proposed techniques, we first randomly choose a test set of 70 switch configurations spread across the design space of interest (both in terms of architectural parameters and target synthesis frequencies), and not overlapping with the training set previously used for characterization. Each switch is synthesized with Design Compiler to extract its area requirements, then stimulated with traffic streams within ModelSim and studied in PrimePower to evaluate its power consumption (Figure 6.12 on page 173). A reference set of experimental results is therefore collected. The area and power consumption of the same set of switches is then estimated according to our methodology, and the statistical distribution of the resulting error is plotted to study the behaviour of both coefficient fitting strategies.

Netlists can be generated in a relatively short time by logic synthesis, but they do not include any information about the placement of the cells, and thus do not give any information about the length of the wires needed for the interconnections. This key missing piece of information is approximated by inaccurate wireload models (Section 6.2.1 on page 147). On the other hand, creating the layout of a complex circuit provides more accurate estimations of its area and power cost, but this extra step is at least as time-consuming as the initial logic synthesis. Therefore, designers would clearly like to avoid performing this extra phase repeatedly during a modeling activity, if at all possible.

To assess the usefulness of our models, we investigate their inference and their application to both netlists and layouts. This can be seen in Figure 6.12 on page 173, where the placement and routing step is optional.

Experiments with Netlist-Based Models and a Netlist-Level Test Set

In this section, we generate our models starting from synthesized (but not placed and routed) switch instances, and check their accuracy against a test set which is also at the netlist level. The results are depicted in Figure 6.16 on page 187, where the vertical axis reports the number of occurrences of inaccuracies comprised in the ranges listed on the horizontal axis. As can be seen, in around 80% of the cases, our models result in an error margin smaller then 10% of the actual value. Sporadically, relatively high error rates of up to 20% are detected; however, as can be seen for example in Figure 6.17 on page 188, the distribution of the errors is quite randomly spread over the design space, and comprises both under- and overestimations. The figure reports modeling inaccuracy for a subspace having as axes the flit width and the switch cardinality; these numbers are thus only a subset of the whole test set. Similar plots can be derived for varying buffer depths and target synthesis frequencies, and we omit them due to space constraints. Therefore, we can attribute inaccuracies to the unpredictability which is intrinsic in the behaviour of synthesis tools, and not to a problem of our modeling approach.

Comparing the results of the two techniques for coefficient fitting presented in Section 6.4.5 on page 182, we see that the tails of the inaccuracy distributions drop more sharply for Methodology 2, indicating a lower chance of large modeling errors. However, Methodology 1 exhibits just marginally worse average inaccuracy rates: 6.26% against 5.30% for power models and 5.97% against 5.45% for area models. In terms of characterization effort, in our experience, we can roughly assume that one hour may be needed in average for the analysis of an instance of the training set; therefore, Methodology 1 requires one hour of runtime, while Methodology 2 needs 24 hours to provide numerical values of coefficients (the actual time depends on how thoroughly the design space is covered). Due to the drastically lower effort, Methodology 1 becomes a natural candidate for fast yet accurate modeling. However, this approach leverages upon a single switch instance to characterize all the coefficients. The choice of the reference switch configuration is therefore key, and may impact the robustness of the flow. Internal testing, that we omit due to space constraints, shows that coefficients are quite accurately rendered under a wide range of possible choices of the reference switch. However, when manually picking an "outlier" instance as the reference, errors over the whole design space turn out to be large. As a possible workaround, Methodology 1 could be applied to multiple switch instances to minimize the chance of choosing bad references; outliers could be effectively discarded. This hybrid ap-

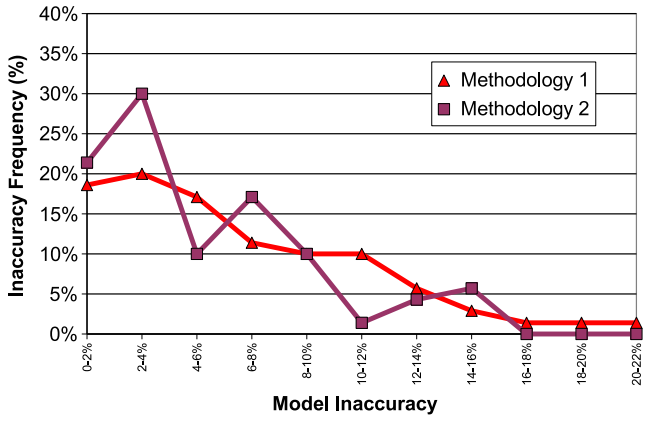
proach provides better reliability, but requires a modeling effort which is progressively closer to that of Methodology 2 as its robustness is increased. Methodology 2 remains the most accurate and reliable, and its characterization time can still be assumed to be fully acceptable for both academic and industrial environments.

Test Case: a Complete NoC Topology

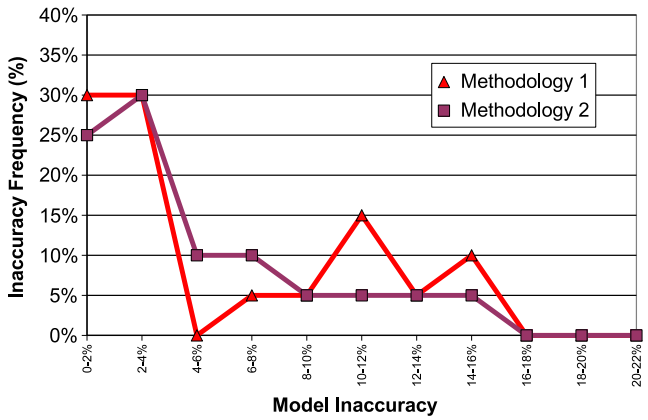
To further validate the most complex part of our methodology, *i.e.* the power modeling, we study a whole NoC topology, such as a 5x3 mesh. The mesh includes switches with three different cardinalities of 4x4, 5x5 and 6x6. We then inject functional traffic, namely that required to drive a multimedia application, in the topology, and compare the resulting power consumption against that predicted by our model (characterized with Methodology 2). Traffic patterns in the mesh are irregular, due to application needs, causing the switches to spend variable amounts of time in each possible state. The results are plotted in Figure 6.18 on page 188. The average inaccuracy is 5%, with only two switches out of fifteen (about 13%) exhibiting inaccuracies greater than 10%. Since the power consumption of some switches is overestimated while that of others is underestimated, the margin of error on the consumption of the whole mesh is as low as 1.3%. This result confirms the usefulness of our modeling strategy for integration within a CAD mapping and design space exploration flow.

Experiments with Netlist-Based Models and a Layout-Level Test Set

We try to apply the previously mentioned models, which are based on netlist-level analyses, to a layout-level test set, by placing and routing the test set described above. This activity generates a very realistic test set, and is a demanding metric for the accuracy of the models, since extra unpredictable noise is added. The results we get are presented in Figure 6.19 on page 189, which should be compared to Figure 6.16(b) on the next page. The two plots exhibit a comparable trend and errors of roughly the same magnitude, even though the average modeling error for the layout-level test set is about 3% higher. This means that models developed by only taking netlists into account still show good accuracy even with respect to layout-level power evaluation. The added noise also blurs the accuracy difference between Methodology 1 and Methodology 2, both in maximum error (26% *vs.* 23%) and average error (8.8% *vs.* 8.35%). While Methodology 2 remains marginally more accurate, these results seem to suggest that the unpredictability introduced by the logic synthesis process is somewhat



(a) Area.



(b) Power.

Figure 6.16: Modeling inaccuracy (percentage deviation among predicted and actual area/power values for the test set) under two different characterization policies for the coefficients. Vertical axis represents the occurrence frequency of a given inaccuracy range. Models and test set are at the netlist level.

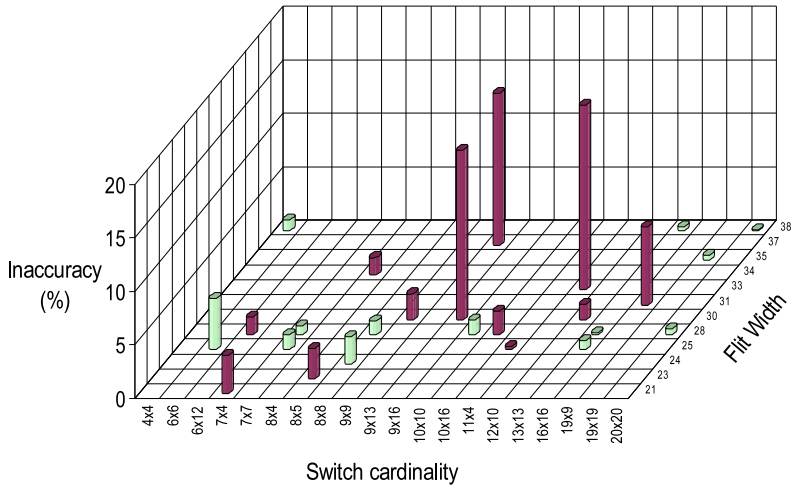


Figure 6.17: Distribution of the area modeling inaccuracy over a subset of the design space for Methodology 2. Dark colour: underestimations; light colour: overestimations.

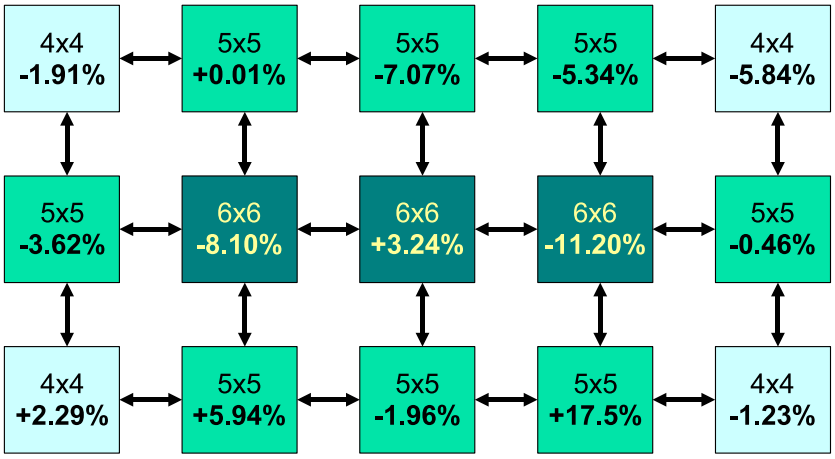


Figure 6.18: Distribution of the power modeling inaccuracy for the switches of a 5x3 NoC mesh.

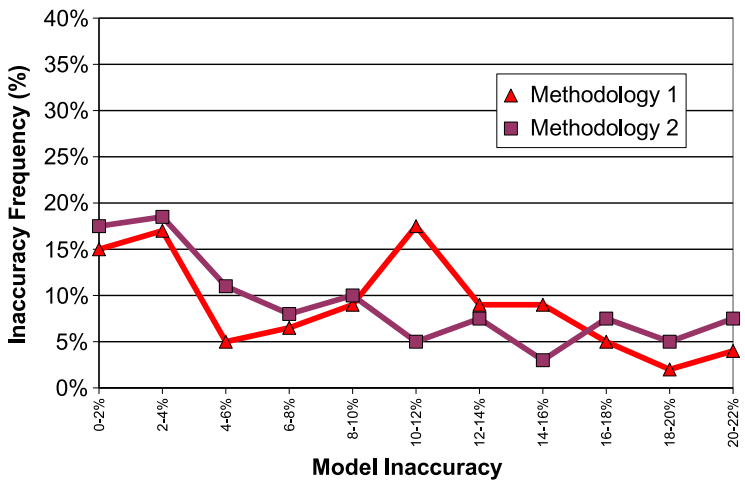


Figure 6.19: Modeling inaccuracy (percentage deviation among predicted and actual values of power for the test set) under two different characterization policies for the coefficients. Vertical axis represents the occurrence frequency of a given inaccuracy range. Models are at the netlist level, test set is at the layout level.

unrelated to that introduced by the placement and routing phase. In other words, even though Methodology 2, thanks to its interpolation of results, can compensate for some of the non-idealities of the logic synthesis process better than Methodology 1, this compensation is less effective when trying to predict the power consumption after placement and routing.

Experiments with Layout-Based Models and a Layout-Level Test Set

In an attempt to check whether more accurate models can be built, we recompute the numerical coefficients starting from a layout-level version of the training set and applying Methodology 2. This model is very close to an ideal reference point, since it is derived from a regression on experimental results which already encompass most of the unpredictable elements of the synthesis flow. However, the time required to build the model coefficients is noticeably longer. In our experience, both logic synthesis and placement steps require a computation time which is not easy to predict, as it largely depends on many factors, such as the switch cardinality and the target operating frequency. However, as a rule of thumb, the two steps are about equally time consuming; therefore, the modeling time is approximately doubled.

The error distribution resulting from the usage of the layout-level test

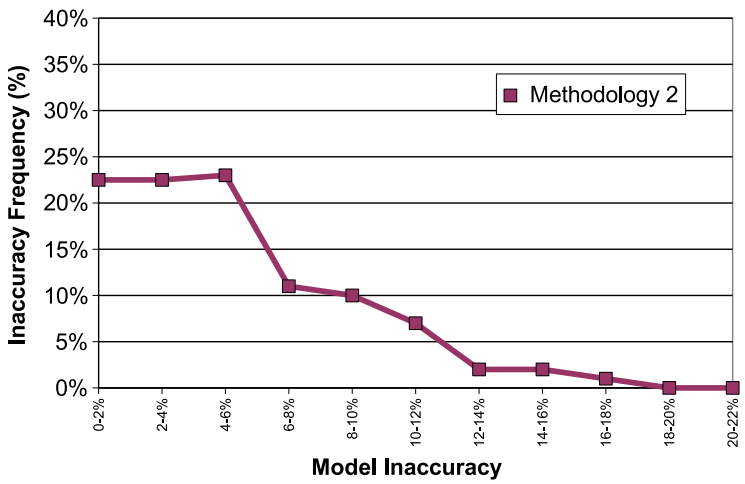


Figure 6.20: Modeling inaccuracy (percentage deviation among predicted and actual values of power for the test set) with Methodology 2 characterization policy for the coefficients. Vertical axis represents the occurrence frequency of a given inaccuracy range. Models and test set are at the layout level.

set when validating the model coefficients achieved from a layout-level training set is shown in Figure 6.20.

As can be noticed, and as expected, the average error and the maximum error values both noticeably decrease when compared to Figure 6.19 on the previous page. However, the decrease is not huge. We attribute the remaining inaccuracies in Figure 6.20 to the intrinsic unpredictability of the synthesis tools. Even after taking into account all the systematic behaviours in the synthesis flow, the trend is the result of residual instance-to-instance variations due to heuristics in the CAD tools and to degrees of freedom which can only vary in a discrete fashion.

The accuracy improvement guaranteed by a layout-level characterization is associated to a doubling of the runtime overhead, and still does not completely eliminate the presence of some "outlier" instances. The designer may certainly choose to adopt our methodology to characterize devices at the layout level for maximum accuracy. However, we feel that a result that can be derived from our experiments is that, at least at the 130nm technology node, it is still feasible to use accurate netlist-based models in order to save characterization time.

Experiments with a Parabolic Model for the Dependency on the Target Synthesis Frequency

We conclude our experiments by checking whether a linear model is accurate enough to characterize the dependency of synthesis results on the target synthesis frequency (see Figure 6.13 on page 175). We leverage a parabolic model as a potentially more accurate approximation of the actual dependency of model coefficients on the target frequency, then recheck the model accuracy on the test set. The results are reported in Table 6.8.

Experiment		Linear approx.	Parabolic approx.
Netlist training set, Netlist test set	Average error	5.19%	6.32%
	Maximum error	14.61%	15.27%
Netlist training set Layout test set	Average error	8.23%	6.57%
	Maximum error	22.88%	19.94%
Layout training set, Layout test set	Average error	5.04%	9.23%
	Maximum error	16.10%	22.23%

Table 6.8: Accuracy of the linear *vs.* parabolic models for the dependency of synthesis results on the target synthesis frequency. Coefficients derived with Methodology 2.

These results do not seem to indicate a strong bias towards any of the alternatives. The linear approximation seems to cope much better with a netlist-level or layout-level test set when the model is derived from experiments on a training set at the same level, but the parabolic model is quite a bit better at predicting layout-level results starting from netlist-level models. We attribute this behaviour to the impact of noise. In other words, although synthesis results do clearly change depending on the target frequency, the choice of a linear or parabolic model to describe this trend does not matter much, since the non-idealities introduced by the synthesis flow induce enough noise to blur the distinction. Overall, the usage of the linear model, which is simpler, seems to be justified.

6.5

Bringing NoCs to 65nm

In this section, we focus on results for the implementation of NoCs in state-of-the-art technologies, such as 65nm libraries, and on some scaling experiments when porting the same NoC from one technology to the other.

6.5.1 65nm Technology Libraries: Degrees of Freedom

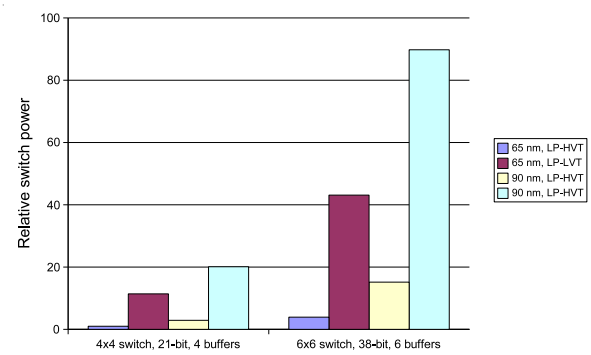
Figure 6.21 on the next page shows how the power, speed and area of a reference ×pipes NoC switch vary, when synthesized based on different technology libraries. The experiment utilizes two 65nm and two 90nm libraries, labeled LP-HVT and LP-LVT; while all of these libraries belong to the LOW POWER (LP) family, the HIGH V_T (HVT) variant strives for absolute minimum consumption, while the LOW V_T (LVT) variant offers a more performance-oriented setup. The switches are fully placed&routed, including the addition of a clock tree.

A first observation is that, as hoped, synthesis in 65nm technologies indeed offers huge benefits compared to 90nm; both area and power experience savings around 50% among comparable libraries, while the frequency of the 65nm design is higher (at least if the LP-LVT library variant is chosen).

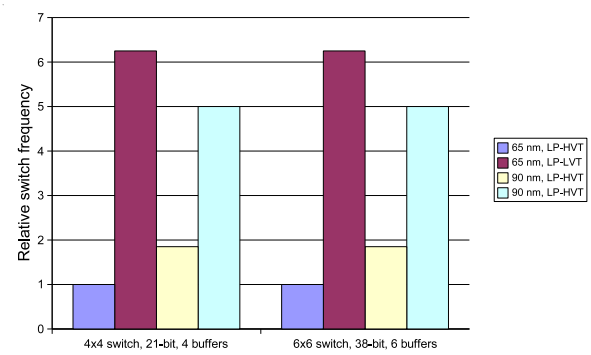
In addition, it is also relevant to observe that, considering for example the power results, already in 90nm technology, there is a factor of six difference among the power consumption of the LP-LVT and LP-HVT implementations; in 65nm, this gap increases to 11×. Similarly for frequency, a gap of 3× in 90nm becomes a gap of 6× in 65nm. Therefore, when designing for next-generation technologies, it is in fact impossible to identify a single technological target. In fact, a very large set of tradeoffs is available, where, by several metrics, results can be up to one order of magnitude different from one another. It is then the designer's responsibility to identify the best set of technological choices in NoC synthesis for the given project.

6.5.2 Link Delay and Power

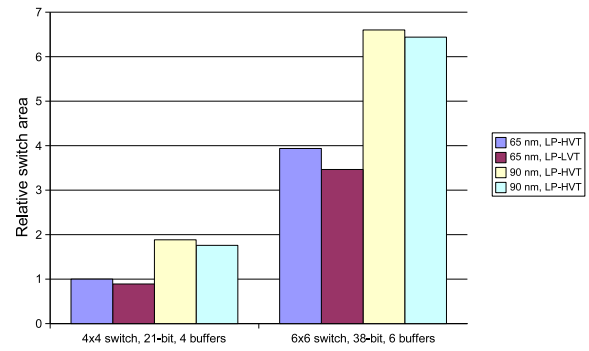
To assess the impact of global wires, we studied 65nm NoC links in isolation from the NoC modules. An overview of the results is shown in Figure 6.22 on page 195. Several factors have to be considered in link design, including obviously length and desired clock frequency. Short or slow-clocked links do not pose problems. However, as either length or target frequency are increased, an undesired rise of power consumption



(a) Power comparison.



(b) Operating frequency comparison.



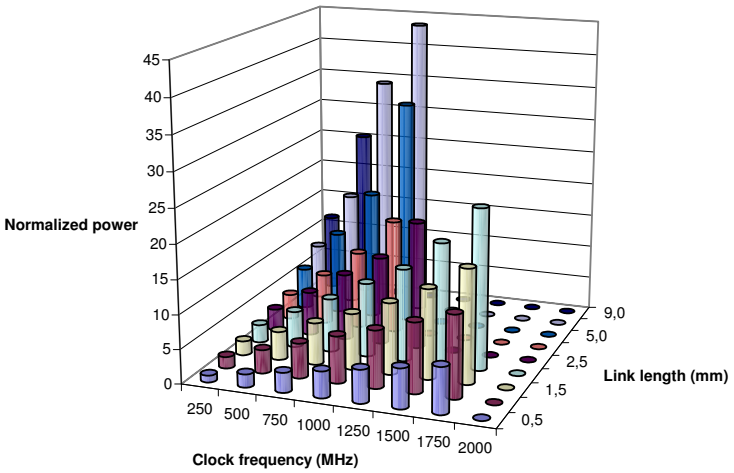
(c) Area comparison.

Figure 6.21: Analysis of two representative ×pipes switches in different technology libraries. Figures normalized to the 4x4 switch in the LP-HVT library.

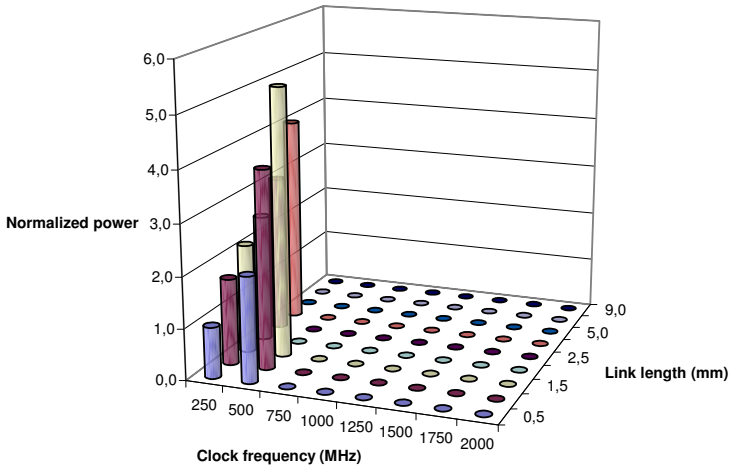
is also observed. The reason is that when links are pushed for high performance, back-end tools automatically insert large amounts of buffering gates, dramatically increasing the energy cost of the links. If link frequency or length are pushed even further, the link becomes infeasible, either because of timing violations, or because of crosstalk concerns, *i.e.*, the added buffers would be too large to be deployed without affecting nearby wires. This kind of tradeoff among link performance, feasibility and power consumption is crucial to the NoC designer.

Another extremely important dependency we observe is on the specific technology library used. As seen in Section 6.5.1 on page 192, especially at the 65nm node, a single "technology library" is no longer realistic for NoC designs based on standard cells. In fact, manufacturing technologies are spreading across a variety of processes optimized for specific uses (*e.g.* low power or high performance), with several intermediate levels featuring, for example, different threshold voltage values. In this case, if very low power libraries are used, the size and speed of the buffers interleaved along wires become dramatically inferior, which results in much tighter constraints on frequency of operation or length. Figure 6.22(a) on the facing page reports power consumption for the 65nm LP-LVT library, while Figure 6.22(b) on the next page describes the LP-HVT variant. These results show that NoC links implemented using the LP-HVT library are substantially more power-effective, but puts much tighter constraints on link feasibility. Hence, the availability of floorplan-aware and technology-aware high-level design automation tools becomes key to pruning the NoC-based design space and to identifying the best libraries for each design according to its particular constraints.

A way to tackle the timing violations on long NoC links, other than just inserting electrical buffers, is link pipelining. Pipeline stages are clocked registers interleaved along the links. By providing one or more extra clock periods to traverse long distances, they solve the link infeasibility problem at a much lower cost than, *e.g.*, by deploying whole NoC switches in the middle of the links. In some cases, pipelining may even produce more power-effective solutions than regular wire buffering along particularly critical links. However, it incurs a performance cost of one extra cycle of latency. Another major drawback is that NoC flow control must be extended to account for the fact that feedback signals are now coming back after multiple clock cycles instead of in the same clock period. This can be tackled by either deploying deeper buffers at the link endpoints, and using plain registers as pipeline elements, or by pipelining the link with flow control-aware elements, without touching the buffers and logic at the endpoints. The latter approach proves better in our experience (Sec-



(a) Performance/power oriented 65nm library (LP-LVT).



(b) Very low-power 65nm library (LP-HVT).

Figure 6.22: Power consumption of 38-bit links of varying lengths at different operating frequencies. Values normalized to shortest link at slowest frequency for confidentiality reasons. Missing columns represent infeasible length/frequency combinations.

tion 4.4 on page 101). In all cases, since link pipelining affects both the RTL description of the architecture and its latency, the need for higher-level (but technology-aware) CAD tools able to pro-actively accounting for them arises clearly, as discussed in Section 5.1.3 on page 120.

6.5.3 Wire Routability Issues in NoCs

All the issues (*e.g.*, crosstalk) applying to global wires in NoCs also apply, to a smaller extent, to local wires. This means that local wires are increasingly critical too in latest and forthcoming technology nodes. As a result, in wire-intensive components, such as, NoC switches, which are essentially crossbars, it becomes difficult to simultaneously achieve signal integrity, timing closure, and routability (*i.e.*, finding a wire layout in such a way that design rules are respected). As tools automatically try to make wires as straight and short as possible to improve timing, and insert spacing among them to avoid crosstalk, a number of DESIGN RULE CHECK (DRC) violations may occur, including overlapping/shorted wires. Routing tools automatically try to remove DRC violations, for example, by means of SR iterations; the design is virtually split into sub-blocks, and the tools begin trying to resolve routing violations one block at a time. If many violations occur, it is unlikely that all will be automatically fixed in the NoC synthesis flow, so designers have to resort to alternate ways, including:

- Manual intervention on the layout, as in full custom design. Of course this is extremely time-consuming and non-reusable, and is normally only undertaken when the violations in the NoC design are very few.

- Decreasing the row utilization, *i.e.*, spreading the module out into a larger area. Ideally this leaves more space for wire routing, but since it may also affect the output of placement (possibly causing the placement algorithm to diverge from timing closure, as discussed above), this alternative must be experimentally explored in future research. In any case, this approach implies at least an area cost.

- Decreasing the target frequency. Wires are allowed to take less straight paths to their destinations without violating timing constraints, and crosstalk is less of an issue, allowing for tighter wire packing. This strategy is very effective in removing DRC violations in NoC synthesis, but its obvious cost is lower performance.

	HVT	MVT	MVT	LVT
Frequency goal	Max	300 MHz	Max	Max
Clock gating	Enabled	Enabled	Enabled	Disabled
Frequency (MHz)	142	300	714	952
Bandwidth (GB/s)	27	57	137	183
Power (mW)	11	25	88	145
Power/bandwidth (mJ/GB)	0.41	0.44	0.64	0.79

Table 6.9: Post-routing performance/power comparison of meshes in different variants of a 65nm technology library.

- Hierarchical floorplanning. This approach tries to better direct the algorithms of the routing tool, by allowing for pre-optimizations and by splitting the problem complexity. Our experience shows that its effectiveness in NoC synthesis depends on the specific module at hand, and must be weighted against the extra design effort at the tool scripting level (usually considerable). Furthermore, hierarchical floorplanning prevents several optimizations that tools can perform on flattened designs. Thus, in the case of NoC switches, this strategy seems to be of limited use in our experience. In fact, if the designer has to manually position even the sub-blocks of switches, just deploying more, smaller switches would require much less effort.

6.5.4 Design Space in 65nm Technology

As already mentioned, at the 65nm node, multiple libraries are available, optimized for performance or power, featuring different supply and threshold voltage levels, *etc.*. To investigate this aspect, we implement the same 4x4 mesh design (see Figure 6.23 on the following page) with different library choices: LVT (fast), HVT (low-power), and MULTI V_T (MVT). The latter option is based on picking gates from multiple libraries at different threshold voltages, and allows for an ideal mix: while gates in the critical path are chosen from the fastest library, the other gates are optimized for power. For the MVT case, we study two configurations: in one of them we aim for a high frequency in order to show the advantages compared to the plain LVT library; in the other, we study a power/performance tradeoff. To make the experiment more accurate, we normally enable clock gating. Since clock gating implies a slight performance penalty, we make an exception for the LVT scenario, where performance is of paramount importance. We choose nominal operating conditions for all the instances. Table 6.9 summarizes our findings.

As can be seen, there is almost an order of magnitude difference in the power/performance ratios achievable by selecting LVT or HVT libraries.

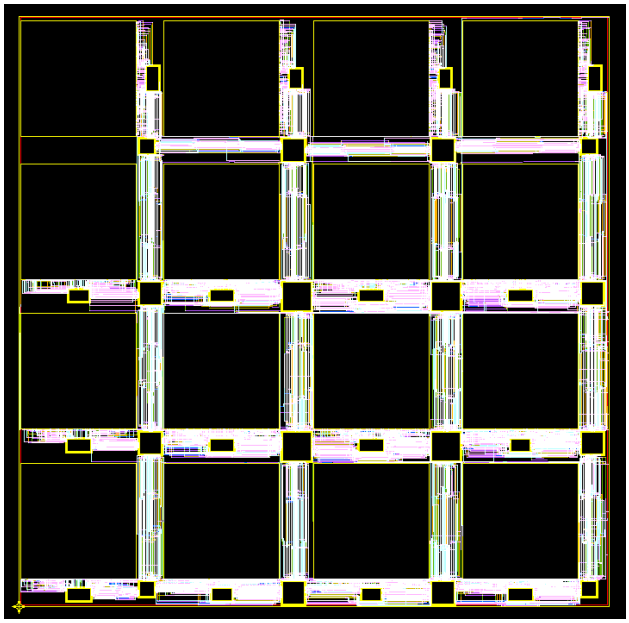
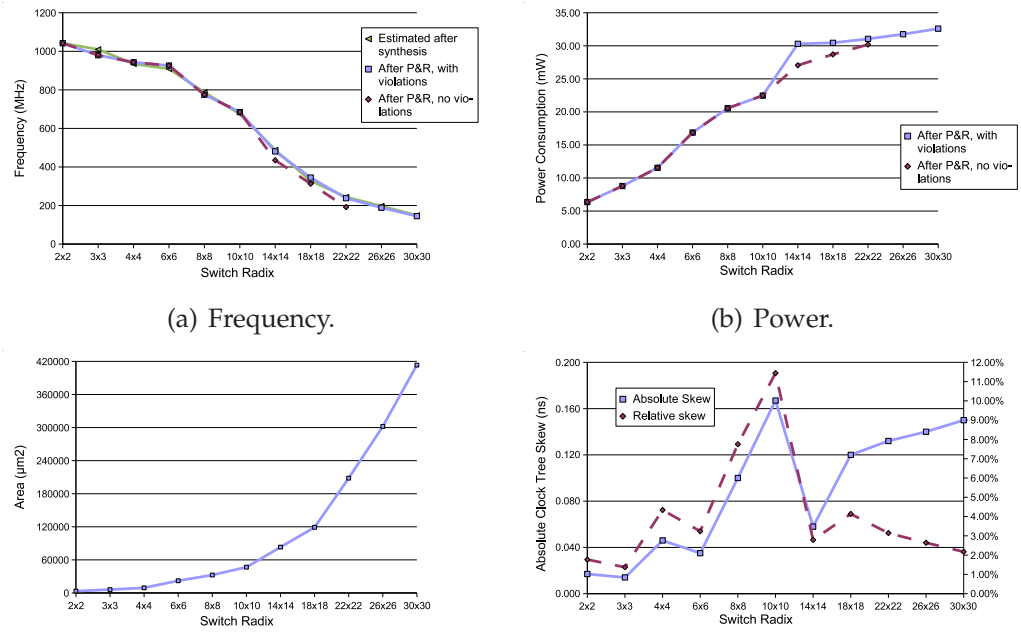


Figure 6.23: A 65nm 4x4 ×pipes mesh.

System architects should take this into account when choosing the ideal NoC configuration. The MVT scenario proves to be a particularly attractive option, with performance approaching the LVT library (the difference in the table is due to the addition of clock gating) at a better power consumption. HVT proves to be the most effective in terms of power per available bandwidth; the MVT design at 300 MHz does almost as well, since it is very far from the maximum frequency point and therefore features a large majority of HVT gates.

6.5.5 Tradeoffs in the Design of Large Switches

In Figure 6.24 on the facing page we show how area, frequency and power scale when implementing ×pipes switches of increasing cardinality in a 65nm MVT 1.2V technology, with clock gating. The area metric is the size of the whole box in which the switch logic is enclosed; cell area itself is smaller, as discussed further down. We characterize power with two simulations on the post-routing netlist annotated with parasitics; the first simulation is in complete idleness, the second features worst-case traffic, with all input ports injecting flits and the maximum number of bits switching (each flit payload flipping all the bits of the previous one). We then simply average the two resulting power figures. While we believe this to be



(a) Frequency.

(b) Power.

(c) Macro Area.

(d) Intra-block clock tree skew.

Figure 6.24: Physical-level metrics for switches of increasing cardinality.

a quite pessimistic metric, it is much more accurate than tool-generated power estimates, since we inject real functional traffic, while tools just assume a certain switching activity value. We typically observe a mismatch of about a factor of two between our results and the tools' automatically generated outputs, with the latter being overly pessimistic.

As expected, area and power increase with the switch radix, while frequency goes down dramatically. The first observation we can draw is that placement-aware synthesis is working as expected; there are no significant gaps among the timing predictions of Physical Compiler and the timing actually reached by Astro after placement and routing (Figure 6.24(a)).

The most interesting result that we observe, however, concerns large switches. The logic synthesis tools are now aware of placement, but not yet of routing. Starting from 14x14, the wire density in the switch crossbars becomes just too high to simultaneously comply with timing objectives, guarantee crosstalk freedom, and resolve DRC violations. Due to the goal priorities we set in our scripts, we achieve the former two, but get an increasing amount of the third, ranging from hundreds (14x14) to tens of thousands (30x30). This number of DRC violations is clearly unacceptable for manual fixing, and must be tackled automatically. As discussed in Section 6.5.3 on page 196, two possible options for fixing are

increasing the switch area or decreasing the switch frequency. The former option proves only partially effective. Typical industrial rates are somewhere close to 85% utilization. The 85% goal can be reproduced without issues in our tests until 10x10 cardinality; at this point, some widening of the target fences proves necessary. For example, the 14x14 switch can only be properly routed once its row utilization is tweaked to be close to 70%; the remaining "unused" space is in fact required to route resources. However, in the 30x30 case, the violations are not fixed even with a final row utilization of 50% (*i.e.* by leaving half of the switch floorplan unfilled). This result is clearly unacceptable due to its cost overhead.

The alternate option of trying to fix DRC issues by decreasing the switch frequency returns somewhat better results, making 14x14 and 18x18 switches routable at a 25-30% frequency cost, but still fails on larger switches. Similarly to the above results, even after more than halving the frequency targets, 30x30 switches remain unroutable.

Even in cases where DRC violations can be fixed by some means, our results suggest that avoiding too large switches may be the best option. This is also due to system-level effects that would result from using large centralized blocks, which are not immediately apparent from the plots reported here. For example, the many cores connected to such a switch would ideally need to be physically placed just around it, causing obvious congestion in the floorplan. Alternatively, they could be spread around, but then several long links would be needed to connect remote cores to the switch. These links would require pipelining, as discussed above, bringing further latency, area and power costs.

6.5.6 Clock Tree Insertion

The last plot in Figure 6.24 on the previous page reports the clock tree skew inside of each of the switches under test. Our clock tree insertion policy, for a whole topology, is as follows. First, we hierarchically synthesize and place each sub-block. Second, we connect all the blocks together in a topology. Third, the clock tree is inserted either globally, after system assembly, or within each of the blocks, before assembly. Fourth, all remaining nets are routed.

The two alternate clock tree insertion approaches have different tradeoffs. In the former, which theoretically guarantees the minimum possible skew since all the design is visible at once, runtime is heavily affected. In fact, we find this strategy to be almost unusable for large 65nm topologies, due to the need for minimum skews over long distances; runtimes would be of many hours and memory usage would be of several gigabytes. The

	Active switch	Idle switch
Dynamic power	52.12 mW	9.46 mW
Leakage power	0.24 mW	0.23 mW
Relative leakage power	0.46%	2.42%

Table 6.10: Dynamic and leakage power in a 22x22 switch in active and idle state.

latter approach, on the other hand, proves to be more efficient. By synthesizing clock trees locally within each sub-block, the local skew can be better controlled (Figure 6.24(d) on page 199); the absolute skew does not increase significantly, while the clock period does, so that the relative clock skew remains constant. When creating a complete topology, the clock trees can then be joined to a single common root, compensating (if necessary) for the different delay of each clock tree. The routing tools are easily able to minimize the frequency loss brought by a skew of less than 10% to a negligible drop.

6.5.7 Leakage Power

Leakage power is often mentioned as one of the major issues in deep submicron design. Our experiences with a 65nm NoC switch tend to mitigate this assumption, as shown by Table 6.10 for a representative MVT case at 1.2V supply voltage. According to our results, leakage represents as little as 0.46% of the total power consumption when the circuit is active; in idle, leakage remains roughly constant, and even though dynamic power decreases by a factor of five (only the clock tree is switching, and it is gated) the leakage power is still below 3%.

However, our results may change under some different scenarios: (i) non-nominal conditions (either high operating temperatures or lower-than-expected transistor threshold voltage), (ii) ability to completely stall the system clock, (iii) need for the use of LVT libraries due to demanding performance requirements. A more thorough assessment of these trade-offs is scheduled as future work.

6.5.8 Test Design: a Multimedia Benchmark

In this test study, we consider a 30-core multimedia benchmark [220], consisting of 10 ARM7 processors with caches, 10 private memories (a separate memory for each processor), 5 custom traffic generators, 5 shared memories and devices to support inter-processor communication. Using

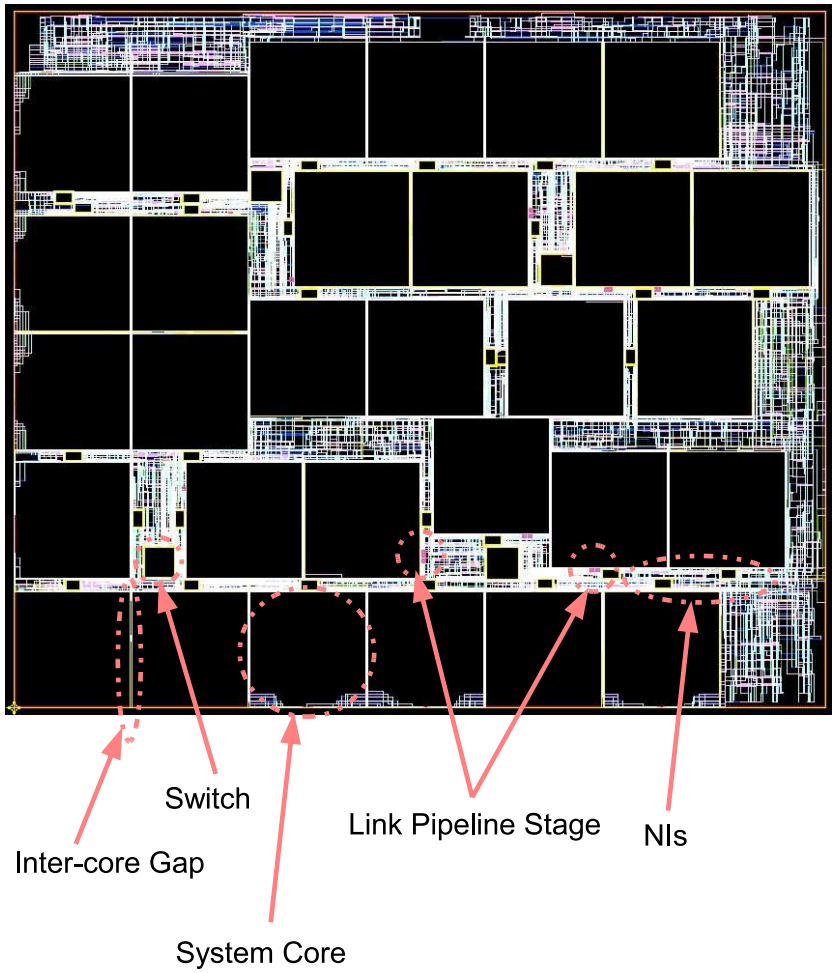


Figure 6.25: The 30-core multimedia benchmark with link pipeline stages automatically placed by SunFloor.

SunFloor, we synthesize the most power-efficient topology for the application, satisfying the application bandwidth and latency requirements. The flit width of the NoC is set to 32 bits, and the target network operating frequency is tuned by the tool to 230 MHz.

In this experiment, we use the 65nm MVT technology libraries, and leverage area, timing and power consumption models of the switches, NIs and links. The floorplan is automatically generated by SunFloor accounting for the repeaters. The sizes of the processor/memory cores are obtained as inputs to the tool flow, and are assumed to be 1mm × 1mm. During the floorplanning process, the NIs are placed together with the cores by using a combined bounding box for the two, as each core com-

municates with its NI through point-to-point wires.

As the physical length of the NoC links can be obtained only after floorplanning, the number and location of the pipeline stages needed along the links cannot be determined beforehand, and must be assessed once the floorplan is known. Pipeline registers are first mapped in the geometrically ideal position, but if another component is already occupying that area, the overlap is resolved by moving the register to the boundary of the nearest core. The bounding boxes of the cores are surrounded by thin gaps 30 μm wide; this space is utilized to place the pipeline flip-flops and to have room to route the wires in the design. The resulting floorplan of the design with pipeline flip-flops is presented in Figure 6.25 on the facing page. Thanks to the accurate pre-characterization of the network components and the fact that the physical design issues are taken into account during the topology design phase itself, the final layout can be achieved with little manual intervention. From our experiments, we find that building accurate models of the components and bridging the gap across the different design phases are critical to achieving a working NoC design in a reasonable time.

6.6 Conclusions

For NoCs to be successful, it is imperative to assess their performance at the physical implementation level. Achievability of design closure, wire predictability and routability, area cost and power budget are all key metrics that can only be assessed once a clear path to physical implementation is laid.

As a contribution of this dissertation, we presented a complete backend flow and a thorough exploration of NoC physical properties. The proposed flow is very rich, and in addition to spanning several levels of abstraction (from RTL level to fully placed&routed layout ready for the foundry), it includes several optimizations (such as clock gating) and facilities for simulation, power characterization and validation of the resulting NoC.

We performed a cross-benchmarking study, whereby we pitted ×pipes NoC instances against hierarchical bus fabrics. The outcome, despite the analysis being performed at the 130nm node and with a NoC library that has been improved in the meanwhile, is very positive for NoCs. Shared buses prove totally unable to cope with current- and next-generation

workloads. Even compared to hierarchical buses, NoCs provide better performance and are far easier to place&route. While they are worse in power, they are actually on par in energy. Further, the scalability of NoCs to future technologies and applications can only be better than that of buses. This work shed light on some questions which were previously unanswered or only partially answered. For example, the NoC handles wiring so well that, at least in 130nm, most of its area and power overhead is concentrated in buffers - which is contrary to expectations in some previous literature. We leveraged this insight to suggest and implement several power optimizations. We also showed that custom NoC topology design, where a NoC is tailored to fit the target application, has noticeable potential benefits. The improvements in power and area are in the 10% range, and are visible even despite being masked by large system-level overheads, such as the resources required by the clock tree and the cores themselves.

We then leveraged the back-end flow to extract flexible, parametric area and power models for NoC switches. The models are detailed enough to guarantee excellent applicability within a NoC CAD flow for topology mapping and/or design space exploration. The area and power models for the ×pipes case study turn out to be very accurate within the limits allowed by the non-idealities of synthesis tools; different tradeoffs among characterization speed and accuracy are possible.

We proceeded to show how NoC performance and power scale to forthcoming technology nodes, thoroughly studying the trends imposed by deep submicron manufacturing processes in NoC designs. Some of our salient results are:

- Designers should leverage the degrees of freedom supplied by the large variety of available technology libraries;
- Synchronous design is still feasible at the 65nm node, even for distributed components such as NoCs, if the clock distribution infrastructure is properly designed;
- High-radix switches are feasible until maybe 10x10 or 14x14, after which their overhead in area and frequency becomes too severe at the 65nm node;
- Link pipelining allows for maximum flexibility in topology design at a relatively low cost, and is becoming a necessity to comply with timing and signal integrity constraints;

- Leakage is not yet critical at the 65nm node as long as the device is operating in normal conditions.

Many opportunities for future research are available. Among some of the most interesting, we would like to mention the possibility of developing some NoC components in full custom design style, to optimize the NoC efficiency; the possibility of studying asynchronous, mesochronous, or GLOBALLY ASYNCHRONOUS LOCALLY SYNCHRONOUS (GALS) synchronization for improved variability tolerance and power consumption; the development of DYNAMIC VOLTAGE AND FREQUENCY SCALING (DVFS) support and policies.

CHAPTER 7

NoC Traffic Handling: Fault Tolerance, Performance, Power

This chapter describes a strategy for packet routing across a NoC. This technique provides the foundation for a number of positive outcomes, such as the ability to better tolerate faults in the memory subsystem, to perform load balancing (resulting in better performance), and to better manage the power drain of the system.

7.1 Motivation and Key Challenges

As the geometries of transistors reach the physical limits of operation, one of the main design challenges for MPSoCs will be to provide dynamic (run-time) support against faults that can occur in the system. The variability in process technology, the issue of thermal hotspots and the effect of various noise sources, such as power supply fluctuations, pose major challenges for the reliable operation of current and future MPSoCs [228, 141]. Failures may be temporary (for example if due to thermal effects) or permanent. One of the most critical elements that affect the correct behavior of MPSoCs is the unreliable operation of on-chip memories [228], where errors can flip the stored bits, possibly resulting in a complete system failure. Current memories already include extensive mechanisms to tolerate single-bit errors, *e.g.* error-correcting codes such as Hamming codes [229, 148]. Memory cores with built-in self-test logic and spare storage resources have also been developed [150, 151, 152]. However these

mechanisms are expensive and the overhead in area, power and delay to recover from multi-bit errors would be very high [228].

Hence, with the increasing uncertainty of device operation, an effective system-level support to memory fault tolerance will be mandatory to ensure proper functionality at a reasonable cost. We propose a novel solution to enable fault tolerant on-chip memory design at the system level for multimedia applications, based on the NoC infrastructure. The main idea is to transparently keep backup copies of critical data on a reliable memory; upon a fault event, data can then be fetched from the backup copy in hardware, without any software intervention. The use of a NoC backbone enables an efficient design which is modular, scalable and efficient; in particular, for example, the flexibility and scalability of NoCs allows for the addition of redundant cores in the same chip (*e.g.* backup memories) without largely increasing the design complexity and without performance penalties. Furthermore, a NoC makes it very easy to place main and backup memories far away in the chip floorplan; this is a key point to counter failures due to phenomena such as thermal hot-spots. The fault tolerance of NoCs themselves has been tackled by several previous papers. For example, noise and coupling phenomena on the links are faced in [153] and [49], where mechanisms for tolerating such interference issues are thoroughly presented. Soft errors can happen, but can be fixed by retransmission of corrupted packets; to this effect, error detection circuits can be coupled to schemes such as [154]. We assume these works as complementary, and leverage upon them.

To understand how to cope with increasing physical-level unreliability, the characteristics of the target MPSoC software applications need of course to be studied in detail. Key drivers in this respect will be various multimedia services, such as scalable video rendering, videogames, *etc*. For large classes of these applications, many types of data corruptions can be tolerated without perceivable service degradation (Section 7.1.1 on the next page), while only some small parts of the memory storage (which include the code segments) are really critical enough to require additional safeguards. At the software level, we thus characterize the application memory footprint into two different types (**critical** and **non-critical**) and focus on the first category.

As a major contribution, we address the design of a reliable integrated memory subsystem for a NoC-based chip. The key idea is to automatically keep backup copies of critical data on a reliable memory; upon a fault event, data can be transparently fetched from the backup copy in hardware, without any software intervention, but purely through the NoC backbone. We handle intermittent and permanent memory faults in the

main memory; upon any occurrence of them, the NoC is dynamically reconfigured to switch all critical transactions to the backup memory. For transient failures, when the main memory recovers, the NoC switches back to the default mode of operation. ×pipes natively enables the decoupling of the frequency of the NoC from those of the attached cores (Section 4.3.1 on page 98), allowing for clocking backup memories at a lower frequency, which improves their reliability and power consumption without the need for any additional clock conversion logic. Since we propose to first split the application data traffic into logically distinct flows, and subsequently to back up only the critical portion of data, which is expected to be comparatively small, our solution does not demand a large overhead, and can even be used in association with existing techniques for memory fault tolerance.

We will proceed to demonstrating the effectiveness with two real-life application case studies, and explore the performance under varying architectural configurations. The overhead to support the proposed approach is very small compared to non-fault tolerant systems, *i.e.* no negative performance impact and an area increase dominated by that of just the backup storage itself.

While the motivating reason of this research is to increase the system fault tolerance, the proposed technique, as will be shown (Section 7.4 on page 224), can also be deployed to improve other properties of the system. Namely, the data flow redirection we propose can be exploited to balance traffic loads among a set of slaves, thus improving performance; it can also be used to transparently divert traffic from one core to another, letting the former be shut down to save power.

7.1.1 Case Study: MPEG4 Video Texture Coder

New multimedia applications cover a wide range of functionality (video processing, video conferencing, games, *etc.*); one of their main common features is that they process large amounts of incoming data in a streaming-based way (*e.g.* a continuous flow of frames). We can observe that certain parts of these streams are essential to produce a correct output, while others are not so critical and only partially affect the user-perceived quality. In many multimedia applications, it is possible to distinguish critical from non-critical data because each type is stored in different data structures within the applications. Let us briefly illustrate these characteristics in the implementation of a real-life multimedia application that is used as one of our case studies in Section 7.3 on page 217, *i.e.* an MPEG4 VIDEO TEXTURE CODER (VTC). VTC is the part of the MPEG4 standard

Figure 7.1: Complete 2D wavelet decomposition in VTC for one image encoded with DC and 3 AC levels.

that deals with still texture object decoding. It is a wavelet transform coder, which can be seen as a set of filter-banks [230] sent in a stream of packets. Each packet represents a portion of an image in different sub-bands, *i.e.* at different resolutions. The first packet of the stream includes the basic elements of the image, but at low resolution. This part is called the *DC sub-band* of the wavelet. If the data that represents the DC sub-band is lost, the image cannot be reconstructed. As typical of critical data in streaming applications, it is very small in size (few kBytes for 800x640 images) and is stored in a dedicated variable and class within the VTC code. The following packets of the stream are called *AC or Spatial Levels* and contain additional details about the image. They have a much larger size than the DC sub-band, but they only refine the image represented by the DC sub-band. If data representing these levels is lost, the user still sees an image, just at a lower resolution. Moreover, whenever a new frame arrives, the previous (faulty) picture is to be updated with the newly received information. Hence, any low resolution output only lasts a very limited amount of time.

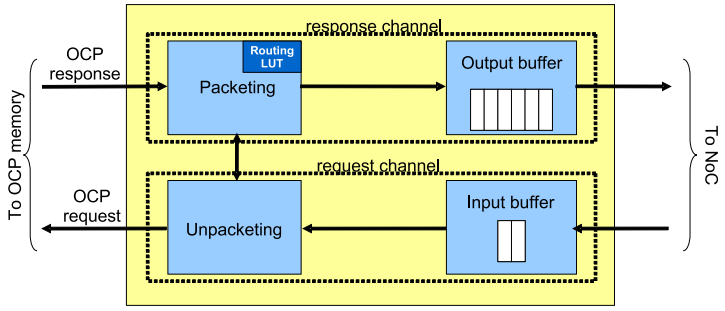
From this example, we can derive fault tolerance requirements for typ-

ical multimedia applications. Only a small part of the data set is critical to the quality of output as perceived by the user, while most of the data to be processed is actually of little importance in this respect. Therefore, it is essential to preserve correct copies only of the former structures, while faults in the latter may be safely accepted.

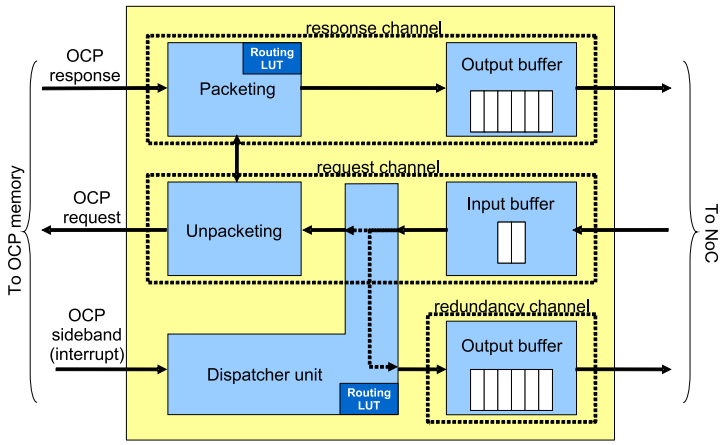
7.1.2 Assumptions on Underlying System

For our reference system, we assume the availability of two classes of memories: "error-detecting" and "reliable". Error-detecting memories, which can be commonly found today, are not capable of error correction but are at least capable of detecting faults, for example by Cyclic Redundancy Check (CRC) codes. We also postulate the availability of memories with much higher reliability for backing up critical data. This assumption is motivated by ad-hoc circuit level solutions and strengthened by three design choices we enable for these memories: (i) small capacity, (ii) lower-than-usual clock frequency (in the experiments in the following we assume one half that of regular memories), (iii) during typical system operation, smaller workload than regular memories. We assume the existence of *main* memories having error detection capability; normal SoC operation leverages upon them, including storage of critical and non-critical data. We add smaller spare *backup* memories, featuring higher reliability, to hold shadow copies of critical data only. Each main memory requires the existence of one such backup, although a single storage device can hold backups for multiple main memories.

To identify the critical data set, we assume that the programmer defines the set of variables to be backed up, and maps them to a specific memory address range. This address range is then used to configure our NoC, either at design time or at runtime during the boot of the system. The accesses to this particular memory region are thereafter handled with our proposed schemes, improving the fault tolerance of the MPSoC design. Application code is assumed to be a vital resource too. Therefore, instructions are always treated in the same way as the critical data; in the following, we will not mention this distinction for the sake of simplicity. Note that the classification of data into critical and non-critical can also be done using efficient compiler support. In this case, the user can mark critical data using special macros and the compiler can map the data to a specific address range. The size of the critical set will depend on the application at hand, and is impossible to predict in general. We aim this work at streaming applications, mostly in the multimedia field, for which the amount of critical information can be safely assumed to be small in



(a) Plain target NI architecture.



(b) Extended target NI architecture.

Figure 7.2: Target NI extensions to support traffic rerouting.

percentage. These applications do represent a significant slice of the embedded device market.

7.1.3 Proposed Hardware Extensions

To implement our approach, we perform changes to the NoC building blocks. The flexible packet-switching design of NoCs ensures that these changes are transparent to the transport layer (switches and links), but NIs need to be made aware of fault events. Two NIs exist natively in ×pipes (Section 4.3.1 on page 98): *initiator NI* (attached to a system master, such as a processor) and *target NI* (attached to a system slave, such as a memory). Both perform source routing by checking the target of the transaction against a routing LookUp Table (LUT).

The changes to the target NI can be seen in Figure 7.2. The original target NI is still plugged to backup memories, while the extended version

is used for main memories. A plain target NI features a request channel, where requests from system masters are conveyed, and a response channel, through which memory responses are packeted and pushed towards the NoC. A third channel (redundancy channel) is now added to the extended target NI; this channel is an output, and re-injects some of the request packets back again into the NoC. By this arrangement, critical-data accesses to the memory (*i.e.* within a predefined address range) can be forwarded to the backup storage element. Not all packets are forwarded; during normal operation, that is before a fault detection, only writes to critical address regions follow this path. This ensures that the backup memory is kept up to date with changes in critical data, but minimizes the network traffic overhead and increases the reliability of the backup memory, which faces a smaller workload. Since the backup memory only receives write commands, it remains silent, *i.e.* it does not send packets onto the NoC. This prevents conflicts such as two memories responding to the same processor request. The resulting flow of packets is depicted in Figure 7.3(a) on page 215. The forwarding behavior is controlled by a *Dispatcher* NI block, which supervises input and output packet flows. An extra routing LUT directs forwarded packets; the LUT normally consists of just a single entry, since there is only one backup memory per each main memory (furter extensions could be possible, though).

The extended target NI also features an extra interrupt interface by the memory side. Whenever a fault is detected, the memory can issue an interrupt. This triggers a reaction by the dispatcher, which responds by beginning to also forward critical read packets to the backup memory according to the extra routing table entry. In this way, reads that would fail due to data corruption are instead transparently forwarded to the backup memory and safely handled (see Section 7.2 on the following page for more details). Critical writes continue to be forwarded as before.

The initiator NI is also extended. First, it checks all outgoing requests for their target address. If the address falls in the specific range provided by the application designer as storage of critical data, then a flag bit is set in the packet header. This allows the dispatcher in the extended target NI to very easily decide whether to forward packets or not. A second change in this NI involves an extra entry in its routing LUT, and a very small amount of extra logic that checks the `SourceID` field in the header of response packets. The initiator NI can thus detect whether a read request it sent got a response from the intended slave or from a different one. As we will show, in our approach, upon a fault, critical reads receive responses from the backup memory instead of the main one. Therefore, noticing a mismatch is an indirect indicator of whether there was a fault in the main

memory. This can trigger different actions depending on the type of error that needs to be handled, as described in Section 7.2.

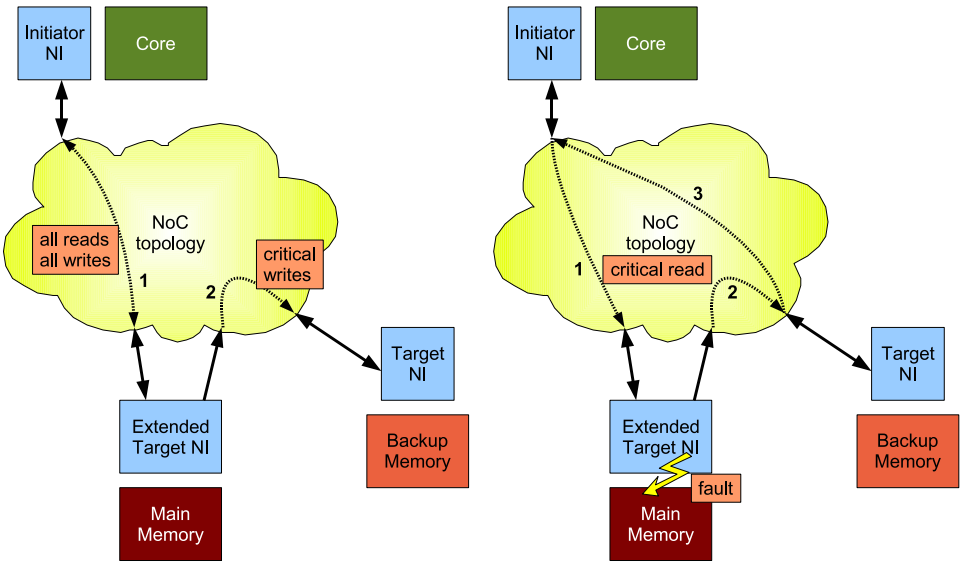
7.2 Run-Time Fault Tolerant NoC-Based Schemes

Two types of errors can occur in on-chip memories of MPSoC designs, namely, transient or permanent. We assume that the system is able to recognize transient errors by detecting some known combination of parameters, either upon the error event itself or even before any error appears. For example, a thermal sensor detecting that a threshold overheating temperature has been surpassed may signal a "transient error" condition before any real fault is observed. The "transient error" condition would be deasserted once the temperature returns to acceptable levels. The same prevention or detection principle could be applied to other electrical or functional parameters that may indicate that a critical point of operation is being approached. In highly fault tolerant systems, the main memory is itself equipped with error correction (not only detection) logic; any internal correction event could then be pessimistically assumed as a hint of an imminent failure. This hypothesis could be reversed after a configurable period of time, once the isolated correction event can be safely assumed to be an occasional glitch, or maybe after a (self-)testing routine. Any known-critical or unexpected events should however be treated by the system as permanent faults, and accordingly handled.

In the following subsections we describe how the proposed extensions can be used to design schemes capable of handling both transient and permanent failures. In both cases, the backup memories do not contain any data upon boot, but are kept synchronized with the main memory at run-time.

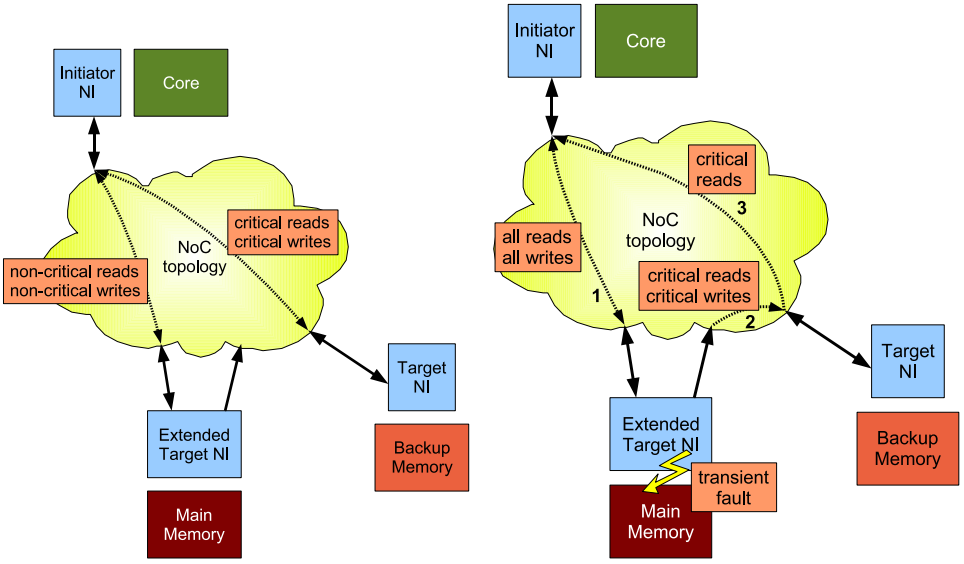
7.2.1 Permanent Error Recovery Support

As soon as a permanent error is identified, the recovery process begins. First, critical-region operations continue to be issued by the processors to the main memory as normal (see Section 7.1.3 on page 212), but the extended target NI starts diverting both read and write requests to the backup memory. Therefore, the backup memory, which had been silent, begins to generate responses as a reaction to the read requests, while the main memory becomes silent for accesses into the critical address range.



(a) Normal operation with backup.

(b) First phase of recovery for permanent and transient failures: read transaction handling upon fault occurrence.



(c) Final operation mode after recovery from permanent failure.

(d) Operation mode while a transient failure is pending.

Figure 7.3: Handling of packet flows in the system.

The `SourceID` field of request packets is kept unchanged, so that the backup memory automatically sends its reply to the system master that had originally asked for it without any lookup conversion. Figure 7.3(b) on the previous page shows the handling mechanism of critical reads upon a fault.

Since going through the main memory and then the backup memory to fetch data is time consuming, the second phase of our recovery process for permanent faults tries to minimize the performance impact of this three-way handling of critical reads. The extended initiator NI (Section 7.1.3 on page 212) is able to identify whether the source of read responses is the main or the backup memory. The first critical read after the fault occurrence triggers a mismatch detection, which in turn forces the initiator NI to access a different entry within its routing lookup table. Hence, all following memory reads within the critical address range are directly sent to the backup memory after the fault. This clearly improves latencies for the remaining operations. The resulting flow of packets is shown in Figure 7.3(c) on the previous page.

The approach does not introduce any data coherency issue. During normal operation, the forwarding of write transactions guarantees that critical data is always consistent among the main and backup memories. Writes are forwarded just before hitting the main memory bank, not after having been performed; in this way, a faulty main memory has no chance of polluting the backup copy of the data. The contents of the backup memory are updated after a slight delay, but this causes no issue as the sequence of packets is strictly maintained. Upon a fault occurrence, transactions are initially directed to the main memory, and only afterwards, when needed, are routed to the backup device; this arrangement avoids skipping transactions and guarantees that all pending transactions (reads and/or writes) are completed on the correct copy of the data. Therefore, proper functionality is strictly maintained when introducing the extra storage bank.

Similarly, when adding the backup memory to the NoC, deadlock issues do not arise given a proper design of the NoC routing scheme. In this respect, the NoC designer must accommodate for one extra IP core and some extra routing paths during the deadlock-free NoC mapping stage. We provide a streamlined way of handling the issue by integrating the discussed reliability enhancements within the SunFloor flow Chapter 5 on page 115.

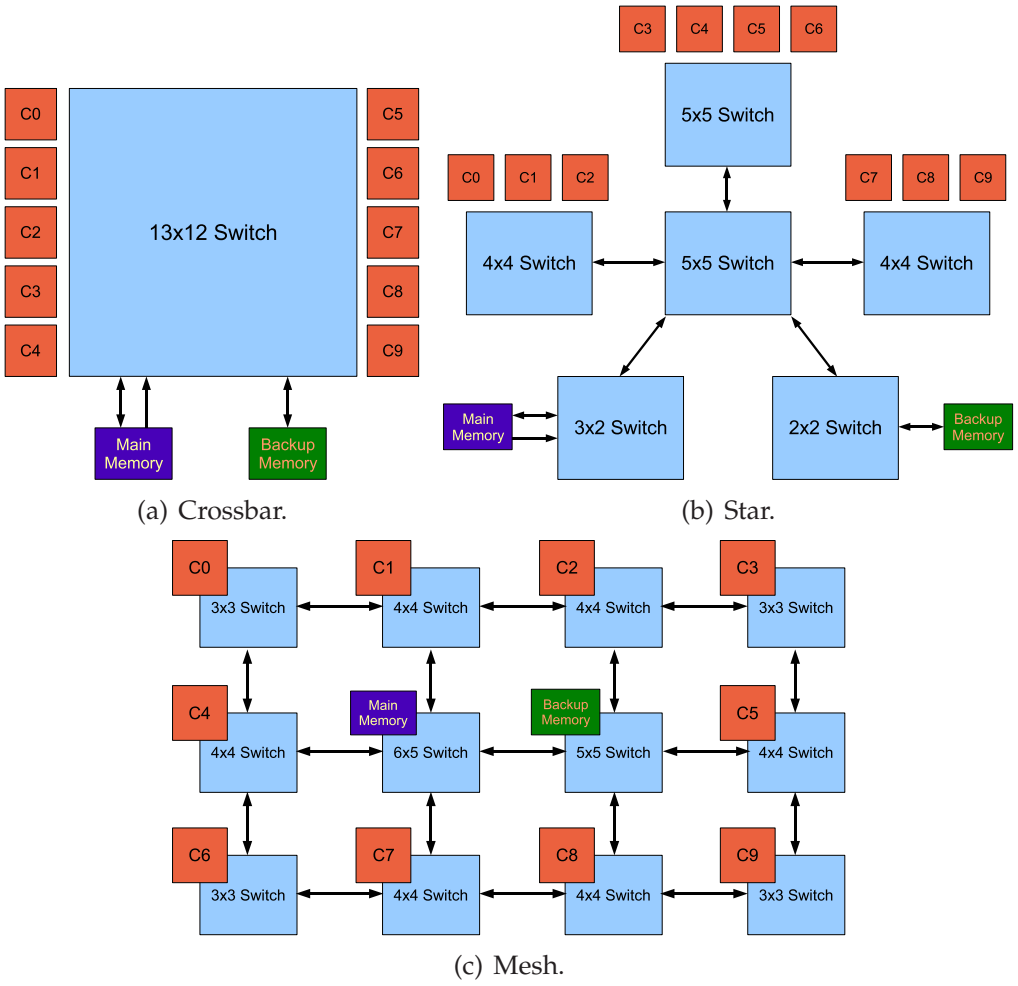
7.2.2 Intermittent Error Recovery Support

In the case of transient errors (*e.g.* due to overheating detection), the first phase of the recovery process is as seen above; critical-region read transactions are automatically forwarded to the backup memory, which automatically responds to the initiator. However, the second phase differs due to the nature of transient failures, where the main memory is supposed to recover complete functionality at a certain moment in time. All traffic, including the critical one, continues to be sent from the processor to the main memory. The extended target NI, being aware that a fault condition is pending, diverts all critical reads towards the backup memory, but lets critical writes be performed towards both the main and backup locations. When the main memory detects that it is able to return to normal operation (*e.g.* after a temperature decrease), it is allowed to issue a different interrupt to indicate so. The extended target NI then resumes normal operation.

The main assumption is that updates to the critical data set in main memory can be successfully performed even during the "transient fault" state. This might be allowed, *e.g.* by choosing conservative temperature thresholds to assert the fault warning. If this solution is not acceptable and the designer does not want to consider the fault permanent, we assume that a higher-level protocol will transfer the safe backup copies of critical data back to the main memory after its return to full functionality.

7.3 Experimental Results

To assess the validity of our approach, we employ two different benchmarks from the multimedia domain. The first one is the MPEG4 VTC application already described in Section 7.1.1 on page 209. As a second test, we use one of the sub-algorithms of a 3-DIMENSIONAL IMAGE RECONSTRUCTION (3DR) algorithm [231] (see [232] for the full code of the algorithm, 1.75 million lines of C++ code), where the relative displacement between every two frames is used to reconstruct the third dimension. Similarly to the VTC benchmark, the amount of critical data that stores control information about the matching process (*e.g.* 160 kB for images of 640x480 pixels) is much smaller than the overall input data per each 2-frame matching process (2 MB at the same resolution), and is stored in two data structures which are easily identifiable by the application designer.



(a) Crossbar. (b) Star.

(c) Mesh.

Figure 7.4: The three topologies under test.

In our experiments, we run the 3DR and the VTC benchmarks on top of three reliability-enhanced topologies, as shown in Figure 7.4. Both benchmarks are implemented using 10 processing cores and a single main memory. The first topology is a NoC crossbar, the second is a star, and the third is a mesh. The topologies and benchmarks are chosen to illustrate different situations of performance penalty for adding reliability support, since the applications demand different features. In fact, 3DR tends to saturate the main memory bandwidth, while VTC is less demanding. The NoC is simulated with a cycle-true simulation environment (Section 3.2 on page 49). We clock the NoCs at 900 MHz (a realistic value, as seen in Chapter 6 on page 145), twice the frequency of the cores and memories.

7.3.1 Performance Studies

We run the benchmarks in five different setups. The first two are reference baselines, the remaining ones represent our proposed scheme.

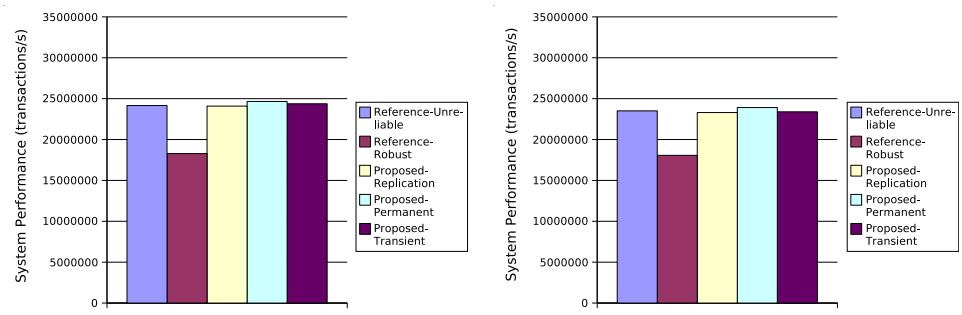
- *Reference-Unreliable*: our reference run is a system without reliability support at all, where accesses are to a fast (450 MHz) main memory. No faults are injected.
- *Reference-Robust*: we model the same system with a reliable main memory running at a lower frequency, therefore minimizing error occurrences [145] and accounting for the overhead of extra circuitry. System performance is obviously impacted, but robust operation can be assumed.
- *Proposed-Replication*: we create a system with a fast main memory and deploy a slow backup memory, but we do not yet inject any fault in the system. As a result, the overhead for the backup of critical data can be observed. We assume the backup memory to be clocked at half the clock speed of regular memories, for the same reasons outlined in the previous setup.
- *Proposed-Permanent*: we create a system with a fast main memory and deploy a slow backup memory, then inject a permanent fault right at the beginning of the simulation. This enables the evaluation of the impact of accessing the backup copy of critical data.
- *Proposed-Transient*: we create a system with a fast main memory and deploy a slow backup memory, then inject a transient fault right at the beginning of the simulation, and never recover from it. This analysis helps to understand what happens to system performance during the period where the main memory is accessed first, but critical traffic needs to be rerouted to the backup memory.

Figure 7.5 on page 221 reports performance, measured in completed transactions per second, for our test setups. The system throughput of most of the scenarios is close, with *Reference-Robust* being much worse than average and *Proposed-Permanent* performing much better, at least in the 3DR case, than even *Reference-Unreliable*. We explain these major effects by observing that both benchmarks, like most multimedia applications, place heavy demands in terms of memory bandwidth; this is a logical consequence of parallel computing on a 10-core system. In *Reference-Robust* the available memory bandwidth is decreased to provide more reliability, which causes performance to worsen dramatically: throughput

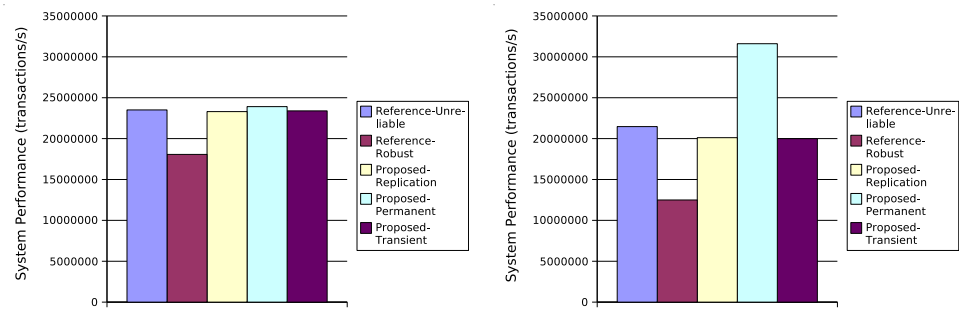
drops by about 24% in VTC and by as much as 43% in 3DR, which is even more demanding. For the same reasons, the *Proposed-Permanent* scenario, where critical data is stored in a separate device, actually guarantees a performance boost related to load balancing among the two memories; the boost is up to 40% for 3DR. Under less demanding applications, we expect both scenarios to perform more similarly to the baseline. The *Proposed-Replication* scenario exhibits a minimal penalty compared to the unreliable case, since the traffic overhead is well handled by the NoC. VTC rarely accesses critical regions, so no penalty is noticeable; in 3DR the throughput decreases 1% to 9%. Finally, the *Proposed-Transient* case exhibits a performance level close to *Reference-Unreliable*, because non-critical traffic behaves exactly as in the base scenario, but several effects related to critical traffic have to be accounted for. On the one side, critical traffic creates NoC congestion and incurs a latency overhead. On the other hand, the main memory does not have to process critical reads, therefore the non-critical transactions can be executed with less delay. In VTC, the overall balance is roughly even. In 3DR, where a larger amount of critical reads (*e.g.* instruction cache refills) takes place, the main memory benefits from large latency gains.

Experimental results show that, in order to improve system reliability, deploying a single highly fault tolerant main memory (*Reference-Robust*) may not be a wise choice in terms of performance within complex multimedia systems. In our proposed architecture, the main memory is left running at a high frequency, and a slower secondary memory bank is added. This choice incurs minor throughput overheads both during normal operation and after fault occurrences. These results justify the feasibility of deploying our architecture even in throughput-constrained environments.

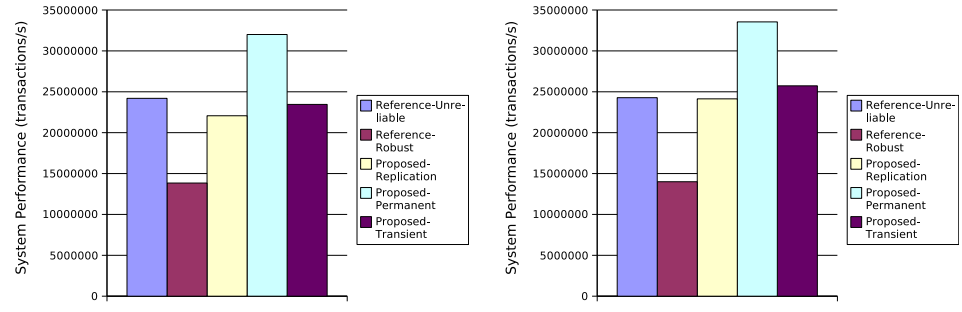
The gains we outline for the *Proposed-Permanent* scenario suggest that always mapping critical information to a separate reliable memory, without inter-memory transactions, may be a simpler yet efficient approach, due to load balancing. However, such a choice does not improve reliability as much as our backup mechanism. First, having two copies of critical data is certainly more reliable than having a single one. Second, using the main memory as the default resource permits a lower workload for the backup memory during normal operation (only write transactions need to be processed), which further increases its reliability. Since the focus of this work is high fault tolerance, we feel that a redundant data mapping is justified, and our aim is simply to verify that performance is not seriously impacted as a result. Performance optimizations through reduction of local congestion can always be achieved by the system designer by tuning the memory hierarchy, which includes deploying multiple storage elements;



(a) VTC benchmark on crossbar. (b) VTC benchmark on star.



(c) VTC benchmark on mesh. (d) 3DR benchmark on crossbar.



(e) 3DR benchmark on crossbar. (f) 3DR benchmark on crossbar.

Figure 7.5: Comparative performance cost of adding reliability support for VTC and 3DR on crossbar, star and mesh topologies.

these steps can be taken in combination with our proposed approach.

7.3.2 Architectural Exploration of NoC Features

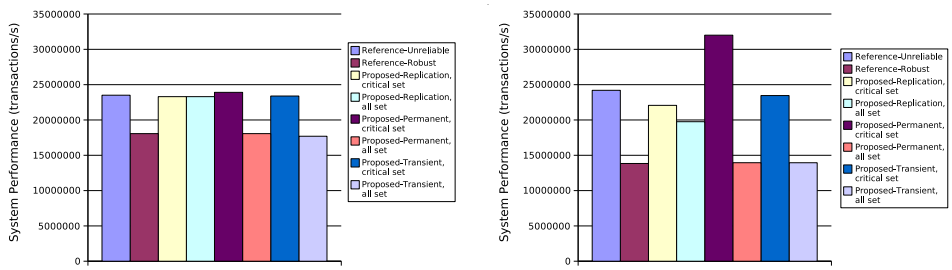
We extend our analysis to different NoC-based hardware architectures using the same NoC backbone. We vary some parameters of our baseline topologies. First, we modify the star topology of Figure 7.4(b) on page 218 by attaching the backup memory beyond a further dedicated switch. The total distance from the central hub is therefore of two hops instead of one. In this way we model backup memories further apart from main memories in the chip floorplan, which improves the tolerance to local overheating. Performance is unchanged under the *Reference* scenarios, where the backup memory is never accessed. In *Proposed* scenarios, where the backup storage is in fact accessed, throughput worsens by less than 0.3%. This is because the latency to go through an extra hop in the NoC is very small, provided there is limited congestion. If the latency to reach the backup memory becomes too large, the topology designer may want to add dedicated NoC links.

To test the dependency of performance on the buffer depth of the redundancy channel, we try a sweep by setting this parameter within the extended target NI from 3 to 6 stages. Our results indicate that, both in VTC and 3DR, deep FIFOs only improve system performance by less than 2%, which indicates that large buffering is not mandatory in the extended target NI.

To validate the effectiveness of the routing shortcut that is enabled in the initiator NI after permanent faults, we measure the latency of two different transactions on the star topology: (1) a critical read going from the core to a faulty main memory, bouncing towards the backup memory, and from there to the processor again and (2) a read directly towards the backup memory after the processor has updated its internal lookup tables. The minimum latency is cut from 78 to 68 (-13%) clock cycles, and the average one goes down from 103 to 95 (-8%). This metric, while topology-dependent, shows the advantage of updating the routing decisions of the initiator upon permanent faults.

7.3.3 Effects of Varying Percentages of Critical Data

It is important to explore different reliability/performance tradeoffs according to the amount of variables that are considered critical: the more data needs to be backed up, the larger the safe backup memories need to



(a) VTC. (b) 3DR.

Figure 7.6: Impact of adding reliability support on the star, with different sizes of the critical data set.

be. Since backup memories are supposed to be reliable also thanks to being smaller, slower and relatively little accessed, the effect of having large backups upon reliability is unclear. To shed some light onto the performance side of the issue, we analyze the behavior under different rates of possible critical *vs.* non-critical data in Figure 7.6. The star topology is taken as an example. In the plots, the *Reference-Unreliable* bar can be assumed to represent an ideal case where no data is critical. For the *Proposed* cases we protect against faults two different memory area: the actual critical set of the benchmark (the same of the studies in Figure 7.5 on page 221, labeled "critical set"), and as an extreme bound, the whole address space ("all set"). The first interesting remark is that the *Proposed-Replication* performance, *i.e.* the system throughput before any fault, but in presence of the backup overhead, is only moderately impacted by the size of the critical data set. In the worst case of 3DR, which is severely bandwidth-limited, even backing up the whole address space incurs a penalty of just 18%. On the other hand, in case of a fault, the size of the protected memory space is a key performance parameter. While choosing a small critical set allows for very good throughput, extending the fault tolerance to the whole main memory content incurs a large penalty. This is in agreement with expectations; in both the *Proposed-Permanent* and the *Proposed-Transient* cases, all traffic is ultimately redirected to the backup memory, which is running at a lower frequency: therefore, throughput becomes similar to the *Reference-Robust* baseline.

This bracket of results frames the applicability of our approach. If the critical set of the application can be kept small, throughput penalties are minimal and advantages are clear. Otherwise, performance degrades up to a worst case equivalent to a system with a single reliable memory.

7.3.4 Synthesis Results

Regarding the modifications in the NoC to support a backup memory, four changes are needed: (i) the NI associated to the main memory must be augmented, (ii) the backup memory needs an extra (plain) target NI device, (iii) the initiator NI becomes a bit more complex, (iv) extra links and switch ports may be needed for routing data to the backup memory.

To assess the silicon cost of the proposed extensions, we synthesize the original and extended NIs with a 130nm UMC technology library. Initiator NIs experience no operating frequency penalty to support the extra functionality, while area increases by about 7% (0.031 mm^2 against 0.029 mm^2). We also study extended target NIs, having 4-slot buffers in the response channel and 3- to 9-slot buffers in the extra redundancy channel. The impact on maximum achievable frequency is just of 2% to 6%, negligible in a NoC where the clock frequency is limited by the switches (Chapter 6 on page 145). By adopting a 4-slot buffer identical to that of the response channel, area increases from 0.032 mm^2 to 0.039 mm^2.

As a result, the area cost due to NI changes is 0.041 mm^2. Overall, even including other possible overheads in the NoC (*i.e.* extra ports in switches and extra links), the final overhead is still small in comparison to the area of the extra backup memory bank itself, which can take on average 1 mm^2 of area for a 32 kB on-die SRAM in 130nm technology.

7.4 Additional Applications of the Proposed Methodology

The same NoC extensions that allow for the fault tolerance mechanism described up to now can also be used for other purposes. The main idea is to exploit the presence of multiple (identical or similar) instances of the same type of core attached directly to the NoC (a *pool* of cores). For instance, a pool can consist of a set of accelerators, a set of memories, *etc.*. This parallel arrangement is a very common property of multicore computation systems, either to comply with performance requirements or to improve reliability via redundancy. By extending the NIs attached to such a pool of cores as discussed above, they acquire the capability of redirecting packets towards alternate cores in the pool. In addition to enabling fault tolerance, this capability could also be exploited to balance the communication load in the pool, improving performance. Alternatively, it allows for some of

the devices in the pool to be switched off to save power, while still being able to transparently divert incoming requests to other, active devices. Many policies can be conceived to coordinate these facilities. For example, the load balancing or power management decisions could be taken either centrally, or in a distributed manner directly by the devices in the pool. The rerouting of packets in the pool could follow priority chains or be based on broadcasts.

7.5 Conclusions

With the growing complexity in consumer products, a generation of MP-SoC architectures with extreme interconnection fabric demands is being envisioned. One of the main challenges for designers will be the deployment of fault tolerant architectures. We have presented an approach to countering transient and permanent failures in on-chip memories, by taking advantage of the communication infrastructure provided by reliable Network-on-Chip (NoC) backbones. Our design is based on modular extensions of the network interfaces of the cores, and is completely transparent to the software designer. The only activity required by the programmer is minimal code annotation to tell the compiler which parts of the data set are critical. The extensions are also integrated within our NoC design flow, therefore transparently handling instantiation issues. Our experimental results on two applications and three NoC topologies show that the proposed approach has a very limited area overhead compared to non-reliable designs, while being scalable for any number of cores.

The proposed hardware extensions enable capabilities which are beyond the domain of fault tolerance, and exploiting them is an area of future research.

CHAPTER 8

Looking Forward: NoCs for 3D Chips

This chapter[1] provides some groundwork for what could possibly become one of the most fascinating future applications of NoCs: 3D stacked chips. These devices represent, in several respects, an ideal application field for NoCs. Even though the 3D manufacturing technology is not fully developed yet, thus making it difficult to thoroughly assess the viability of specific techniques, we will present initial exploration and implementation results, aimed at getting NoCs ready for 3D integration.

8.1 Motivation and Key Challenges

Over the years, the MPSoC fabrication trend has been towards the integration of larger and larger amounts of processing elements and memories. More specifically, there has also been a strong push towards the mixing of functional blocks which may require a variety of processing steps, such as plain CMOS, DRAM, MEMS, passive and active analog circuitry, opto-electronic elements, chemical sensors, actuators, *etc.*. Unfortunately, each extra manufacturing step increases costs and decreases yield, imposing a limit on the heterogeneity of each silicon die. Vertically stacking multiple layers of silicon is an attractive way of sustaining the pace of improvement in functionality, thanks to advantages such as:

[1]The author would like to thank Igor Loi for his contribution to this chapter.

- Ability to provide more complexity than planar technologies in the same package footprint.
- Modularity. A plug&play approach can now be envisioned not just at the IP core level, but at the die level.
- A potentially increased amount of bandwidth, thanks to the possibilities brought by vertical wiring.
- Potentially lower manufacturing costs at the die level, since each heterogeneous die in a stack can be manufactured with the optimal mask set for its field of application.

3D stacking provides novel physical means of interconnection along the vertical axis, opening many research opportunities. First of all, the yield (and thus the cost) of the vertical links is still not well known. Depending on this crucial parameter, an extremely broad range of architectural solutions may be adopted in order to provide 3D communication. Second, the performance of vertical connections has not been clearly assessed for MPSoC applications, leading to uncertainties with respect to the optimal system design strategies. Third, crucial system-level issues are pending; for example, how to design a skew-free 3D clock tree is currently a question without good answers.

We believe that NoCs represent a synergistic match for 3D chip stacking, because they respond perfectly to three major needs:

- Scalability. If 2D systems are already becoming complex to interconnect, 3D designs can only be more challenging. NoCs provide the headroom to tackle systems with multiple layers and many tens or hundreds of IP cores.
- Modularity. Regardless of the heterogeneity in a 3D stack, both at the manufacturing level and at the circuit architecture level, NoCs provide more flexibility to cope with a varied set of IP cores than anything else available today.
- Serialization. At present, and to some degree for the foreseeable future, each single vertical connection in a stacked chip will reduce the chip yield. Thanks to packet serialization, NoCs can achieve the same performance as buses, or better, with a dramatically reduced number of wires, making them the ideal candidate for wire-constrained stacks.

The development of 3D circuits in general, and 3D NoCs in particular, is still at an early stage [164, 165, 233, 163, 234], also due to the rapid evolution that assembly techniques are undergoing. We try to face some of the many unknowns and challenges of 3D NoC development by presenting:

- A circuit-level model for a particular kind of vertical interconnects: THROUGH-SILICON VIAS (TSVS) [160, 161, 162]. The model is based on accurate three-dimensional parasitic extraction. Comparative analyses demonstrate that not only vertical interconnects are usable, but that they are highly competitive with horizontal wires in terms of delay and power, with a reasonable area overhead.
- An extension of a two-dimensional NoC switch architecture to deal with vertical links.
- An extension to our NoC design flow (Chapter 6 on page 145) for semi-automatic instantiation of three-dimensional NoCs.
- A case study where a planar NoC topology is folded and implemented across two chip layers.
- The design of a mesochronous synchronizer, aimed at coping with the issues in distributing skew-free clock trees across stacked chips.

8.2 Physical Modeling of Vertical TSVs

To be useful for a NoC infrastructure, a vertical wire should not be used in isolation; instead, to simplify routing, it is better to create buses of such wires. The geometry of a TSV bus connecting adjacent stacked wafers is shown schematically in Figure 8.1 on the next page for two manufacturing scenarios: Silicon on Insulator (SOI) and bulk-silicon technologies. Given the physical proximity of the TSVs, concerns related to capacitive coupling within such buses may arise. In this section, we quantify the delay in a bus formed by vertical TSVs for both the SOI and bulk-silicon cases.

TSV models are obtained with the Ansoft Q3D extractor [235], a quasi-static electromagnetic-field simulation for parasitic extraction of electronic components, which utilizes finite element algorithms and the Method of Moments to compute the RLC parameters of a 3D structure. This makes the study of signal integrity (crosstalk, ground bounce) and delay possible.

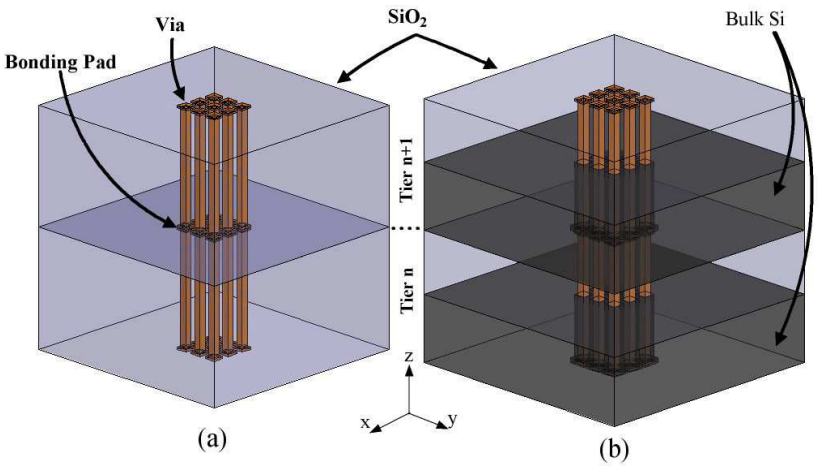


Figure 8.1: Through-Silicon Vias in (a) SOI and (b) bulk-silicon technologies.

The starting point of our analysis is a simple configuration composed of nine TSVs placed in a 3x3 grid structure. The baseline configuration we study (see Figure 8.2 on the facing page) derives from published literature [161, 162] and can be summarized as:

- Copper vias
- $4\mu m \times 4\mu m$ via cross-section ($W \times L$)
- $5\mu m \times 5\mu m$ pads at via extremities
- $8\mu m$ via pitch
- $1\mu m$ oxide thickness (t_{OX}) (only for bulk silicon)
- $50\mu m$ layer thickness ($25\mu m$ bulk silicon and $25\mu m$ SiO_2)

Delay is a function of resistance and capacitance. Resistance can be described with a single parameter as a function of via length ℓ, cross-section σ and resistivity ρ:

$$R = \frac{\rho \times \ell}{\sigma} \tag{8.1}$$

For example, copper TSVs with $4 \times 4\mu m$ diameter show a resistance around $1.18m\Omega$ per μm. The skin effect, at these sizes, is negligible at frequencies of few GHz, and a comparison between vias and top metal wires

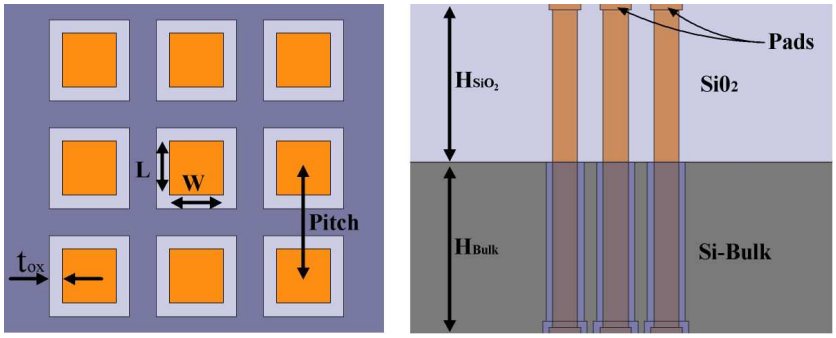


Figure 8.2: Schematic representation of a bundle of 3D vias.

(Metal 8, 130nm technology node) having $0.4\mu \times 0.8\mu m$ cross section shows that the TSV resistance per unit of length is fifty times smaller.

Capacitance, on the other hand, due to coupling effects, poses several more issues. Therefore, we resort to a capacitance matrix $\overline{\overline{C}}$ (Equation (8.2)):

$$\overline{\overline{C}} = \begin{pmatrix} C_{1,1} & -C_{1,2} & \dots & -C_{1,n} \\ -C_{2,1} & C_{2,2} & \dots & -C_{1,n} \\ \dots & \dots & \dots & \dots \\ -C_{n,1} & -C_{n,2} & \dots & C_{n,n} \end{pmatrix} \tag{8.2}$$

In this matrix, the elements outside of the diagonal represent inter-via coupling, with inverted sign, while the ones along the diagonal are the sum of the capacitances towards the ground plane ($C_{i,0}$ - not explicitly reported in the matrix) plus the coupling capacitances:

$$C_{ii} = C_{i,0} + C_{i,1} + \dots + C_{i,i-1} + C_{i,i+1} + \dots + C_{i,n} \tag{8.3}$$

In Table 8.1 on the following page and Table 8.2 on the next page we report extraction results for the capacitance of vias in SOI and bulk-silicon TSVs, respectively, for the reference case. The capacitance towards the ground plane is negligible in the SOI case, since the whole structure is "floating", but it is the dominant element in bulk-silicon technology. On the other hand, due to the presence of a passivation coating around the TSVs in the bulk-silicon case, the SOI scenario exhibits much larger coupling capacitances among the vias.

We can analyze the behavior of TSVs in different geometries using our geometric model. In Figure 8.3 on page 233 we sweep the TSV diameter, from $0.5\mu m$ to $6\mu m$, while keeping the TSV pitch constant at $8\mu m$. Capacitance in the bulk-silicon case increases linearly with the diameter, while

C [fF]	Ground	M	N	S	W	E	SW	NW	NE	SE
M	0.00	11.41	-2.43	-2.43	-2.43	-2.43	-0.41	-0.42	-0.41	-0.42
N	0.00	-2.43	10.13	-0.03	-0.47	-0.47	-0.07	-3.19	-3.18	-0.07
S	0.00	-2.43	-0.03	10.13	-0.47	-0.47	-3.19	-0.07	-0.08	-3.18
W	0.00	-2.43	-0.47	-0.47	10.13	-0.03	-3.18	-3.19	-0.07	-0.07
E	0.00	-2.43	-0.47	-0.47	-0.03	10.13	-0.08	-0.07	-3.19	-3.18
SW	0.00	-0.41	-0.07	-3.19	-3.18	-0.08	8.32	-0.40	-0.11	-0.41
NW	0.00	-0.42	-3.19	-0.07	-3.19	-0.07	-0.40	8.31	-0.40	-0.12
NE	0.00	-0.41	-3.18	-0.08	-0.07	-3.19	-0.11	-0.40	8.32	-0.41
SE	0.00	-0.42	-0.07	-3.18	-0.07	-3.18	-0.41	-0.12	-0.41	8.31

Table 8.1: Capacitance matrix of TSVs in SOI technology. M = middle via; the other vias are labeled according to their positioning with respect to it (N = north, *etc.*). "Ground" refers to the ground plane ($C_{i,0}$). The capacitive load of an inverter in this technology is about 2 fF.

C [fF]	Ground	M	N	S	W	E	SW	NW	NE	SE
M	-17.7	23.89	-1.20	-1.21	-1.20	-1.20	-0.33	-0.33	-0.36	-0.36
N	-18.1	-1.20	23.26	-0.09	-0.39	-0.34	-0.05	-1.58	-1.52	-0.05
S	-18.3	-1.21	-0.09	23.39	-0.35	-0.33	-1.47	-0.06	-0.05	-1.56
W	-18.1	-1.20	-0.39	-0.35	23.25	-0.09	-1.57	-1.48	-0.05	-0.05
E	-18.3	-1.20	-0.34	-0.33	-0.09	23.42	-0.05	-0.06	-1.55	-1.52
SW	-18.6	-0.33	-0.05	-1.47	-1.57	-0.05	22.23	-0.11	0.00	-0.13
NW	-18.5	-0.33	-1.58	-0.06	-1.48	-0.06	-0.11	22.16	-0.11	-0.01
NE	-18.5	-0.36	-1.52	-0.05	-0.05	-1.55	0.00	-0.11	22.24	-0.13
SE	-18.3	-0.36	-0.05	-1.56	-0.05	-1.52	-0.13	-0.01	-0.13	22.07

Table 8.2: Capacitance matrix of TSVs in bulk-silicon technology. M = middle via; the other vias are labeled according to their positioning with respect to it (N = north, *etc.*). "Ground" refers to the ground plane ($C_{i,0}$). The capacitive load of an inverter in this technology is about 2 fF.

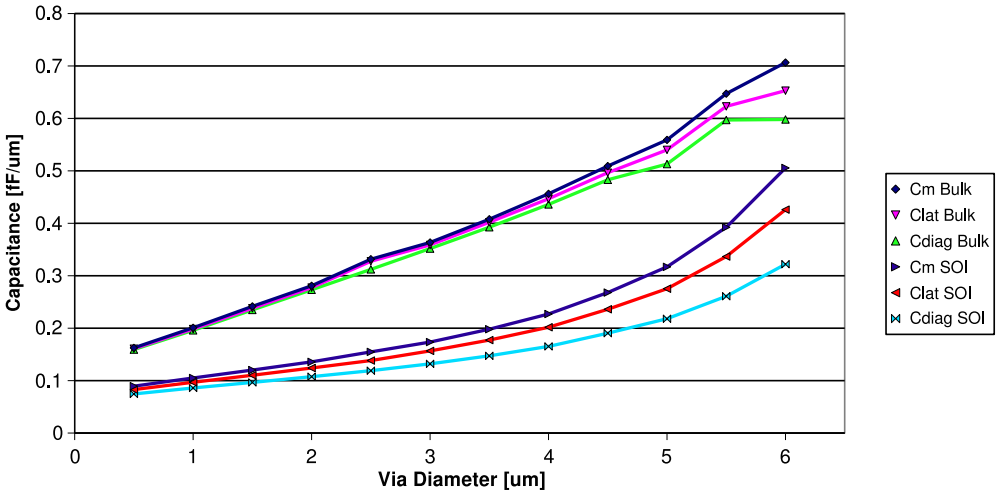


Figure 8.3: Capacitance trend when sweeping the diameter of vias having a constant pitch. Figures are reported for SOI and bulk-silicon. C_m: $C_{1,1}$; C_{lat}: average of $C_{2,2}$ to $C_{5,5}$ (N, S, W, E vias); C_{diag}: average of $C_{6,6}$ to $C_{9,9}$ (SW, NW, NE, SE vias).

the increase is steeper for SOI. This is due to the fact that, in both technologies, the lateral via surface, which determines the coupling, is becoming larger. Further, the distance among the lateral surfaces decreases, since the pitch is constant. However this effect is most relevant in the SOI scenario, whereas, in bulk-silicon, the passivation layer surrounding each TSV (t_{OX} thickness) dampens the increase in coupling.

It is also interesting to sweep via pitch while keeping the TSV diameter constant (*e.g.*, at $4\mu m$). The curves are dual with respect to the previous plot, since increasing via diameters has a similar effect as decreasing via pitches. The most interesting property to be observed is the discontinuity in the bulk-silicon curves at the 6 μm pitch threshold, which represents the point where two adjacent TSVs are actually in contact. This is because vias have a $4\mu m$ diameter, plus, only for the bulk-silicon case, an insulating coating $1\mu m$ thick. Below the $6\mu m$ threshold, we assume that TSVs are dug into a solid SiO_2 structure, and are therefore only separated by a thin oxide layer; above the threshold, a silicon "screen" appears in the middle as each TSV is the result of a separate etching in the silicon substrate. The presence or absence of the silicon layer changes substantially the parasitic capacitance behaviour.

The complete extracted circuit model gives maximum accuracy in electrical simulation, but good insight can be gained by modeling the delay

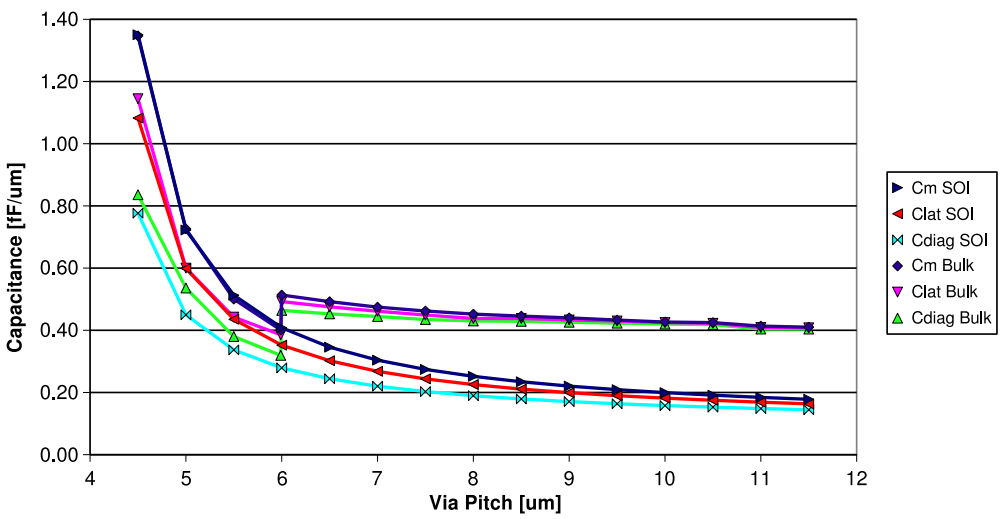


Figure 8.4: Capacitance trend when sweeping the pitch of vias having a constant diameter. Figures are reported for SOI and bulk silicon. C_m: $C_{1,1}$; C_{lat}: average of $C_{2,2}$ to $C_{5,5}$ (N, S, W, E vias); C_{diag}: average of $C_{6,6}$ to $C_{9,9}$ (SW, NW, NE, SE vias).

with the well-known RC approximation:

$$t_D = 0.35 \times R \times C \tag{8.4}$$

In the formula, contact resistance and load capacitance (*e.g.* buffers or flip flop at the end of the line) should be taken into account. Since TSVs are interconnected by means of metal bonding, we estimate the contact resistance [236] to be $100m\Omega$ per layer. Delay estimates using Equation (8.4) are in good agreement with SPICE simulations. For example, once the contact resistance and load capacitance are taken into account, $16ps$ to $18.5ps$ of delay (for SOI and bulk silicon, respectively) are found when the TSV diameter is set to $4\mu m$ and the pitch to $8\mu m$.

To put these results in perspective, the maximum un-repeated planar line length in Metal 2 and Metal 3, in the same technology, is $1.5mm$. Using a planar inter-switch link of this length as a reference, we observe that vertical links exhibit roughly one order of magnitude lower capacitive load. Roughly the same ratio can be found for resistance. As a consequence, even after taking coupling effects of tightly packed TSV bundles into account, vertical links turn out to be substantially faster and more energy efficient than moderate size planar links.

8.3

Integration of TSVs within NoC Switches

NoC components and NoC design tools require modifications to support vertical links implemented with TSVs. As discussed above, 3D designs are likely to expose a large degree of heterogeneity, especially along the vertical axis. We leverage our NoC flow (Chapter 5 on page 115, Chapter 6 on page 145) and build on top of it a semi-automatic 3D flow, from RTL description to layout-level verification.

We leverage the information gathered in Section 8.2 on page 229 to build LEF (Library Exchange Format) descriptions of vertical vias. LEF macros are standard hardware descriptions at the layout level, including information about process technology, cell placement, routing and pins/pads. Based on these macros, TSVs can be accurately inserted within the design during the placement and routing stage; they are simply attached to the input or output pins of a switch port, just as a horizontal bus would. At the RTL level, on the other hand, the design can still be unchanged with respect to a 2D implementation. This brings several advantages: (i) the presence of vertical wires is totally transparent to the architectural and functional views of the architecture; (ii) a chip may feature any degree of connectivity heterogeneity since vertical links can be added or exchanged for horizontal ones; (iii) vertical bandwidth can be added only where needed in the chip, saving switch ports everywhere else; (iv) building upon the savings brought by the previous item, the set of switches with vertical ports, *i.e.* the ones located where vertical bandwidth is really needed, can have ideal performance because they can be implemented as full crossbars.

Thanks to this approach, a complete flow is achieved; this includes the ability to extract and simulate a 3D layout, where all switch ports are exposed to proper timing constraints and load information is available for both horizontal and vertical connections. A depiction of a sample layout featuring a 3x3 switch with vertical ports is presented in Figure 8.5 on the next page. The arrangement of the TSV macros is the one we identified to offer the best timing requirements: close to the pinout of the switch, so as to guarantee minimum length of the wire from the switch to the base of the via, thus reducing parasitics.

The choice of a NoC topology must be performed by taking into account available performance information. Therefore, it is important to build a timing model of the switches. Please recall that ×pipes switches

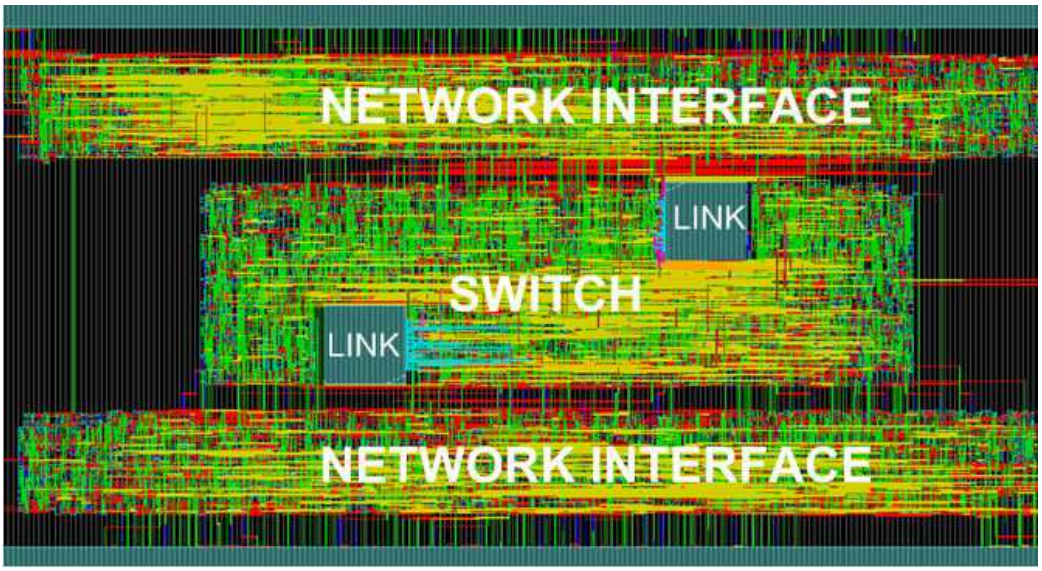


Figure 8.5: Layout detail: a switch is attached to the LEF macros of two vertical links.

come in two radically different variants (Section 4.4 on page 101). One key point to stress here is that, in ACK/NACK, due to the presence of output buffers for retransmission, the NoC links are enclosed between two clocked buffers at the sending and receiving ends. Hence, a whole clock period is available for signal propagation along the wires of the inter-switch links; the link length and the switch logic are decoupled by the output buffer. In contrast, in STALL/GO, to minimize the hardware overhead, only the switch inputs are registered; the switch logic and the link propagation time (up to the following switch or to the first link pipeline stage) contribute to a the same timing path, which becomes the bottleneck for the system. In STALL/GO, the link delay directly impacts the maximum operating frequency of the switches and of the whole NoC.

In Figure 8.6 on the next page, we explore the frequency that STALL/GO and ACK/NACK switches of different cardinalities can achieve when driving horizontal ($1.5mm$) or vertical ($50\mu m$) links. As expected, ACK/NACK switches don't change operating frequency when moving to 3D structures, since their frequency bottleneck is given by the switch logic and is not affected by link performance. STALL/GO is, in general, slightly slower than ACK/NACK due to the contribution of link delay on critical paths; however, when used in combination with TSVs, it regains 30-50 MHz, *i.e.* at 50 to 75% of the frequency gap, while maintaining its low-overhead properties (and single-cycle latency). In other words, the

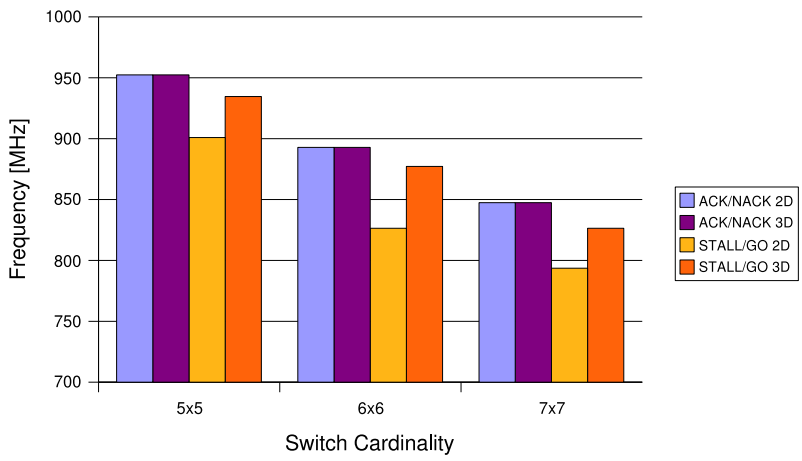


Figure 8.6: Maximum frequency achievable by STALL/GO *vs.* ACK/-NACK switches in 2D and 3D flows, for varying switch cardinalities.

NoC can be clocked faster when slow horizontal links are replaced by fast vertical links.

8.4 Implementation of TSV-Based NoCs

As a validation of our flow, we present a NoC implementation based on a 2D 3x2 quasi-mesh (called simply mesh in the following) and migrate it to a 3D arrangement (Figure 8.7 on the following page and Figure 8.8 on page 239). The 3D mapping is achieved by splitting in two halves the mesh and overlapping them in separate chip layers, with communication achieved through TSVs. The stacked topology has exactly the same functionality of the two-dimensional implementation.

As a first step, we leverage SunFloor and ×pipesCompiler (Chapter 5 on page 115) to instantiate the 2D mesh. There is no need to modify the RTL output of SunFloor in any way. Next, we identify the best partitioning for mapping onto the layer stack. This task is, at present, done manually, due to the large set of constraints involved. These include manufacturing limitations, chip pinout, area considerations, bandwidth demands, thermal requirements, *etc.*. For example, our test 3x2 mesh connects three processors and three memories; since we assume that processors cannot be stacked on top of each other, to avoid the formation of hot spots, we inter-

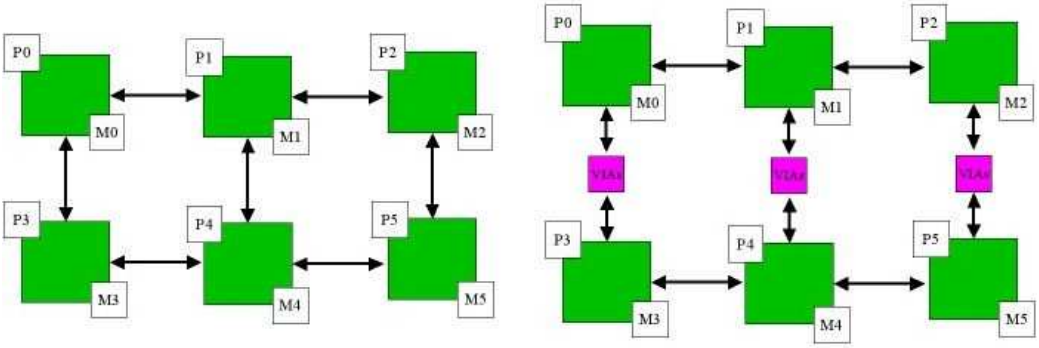


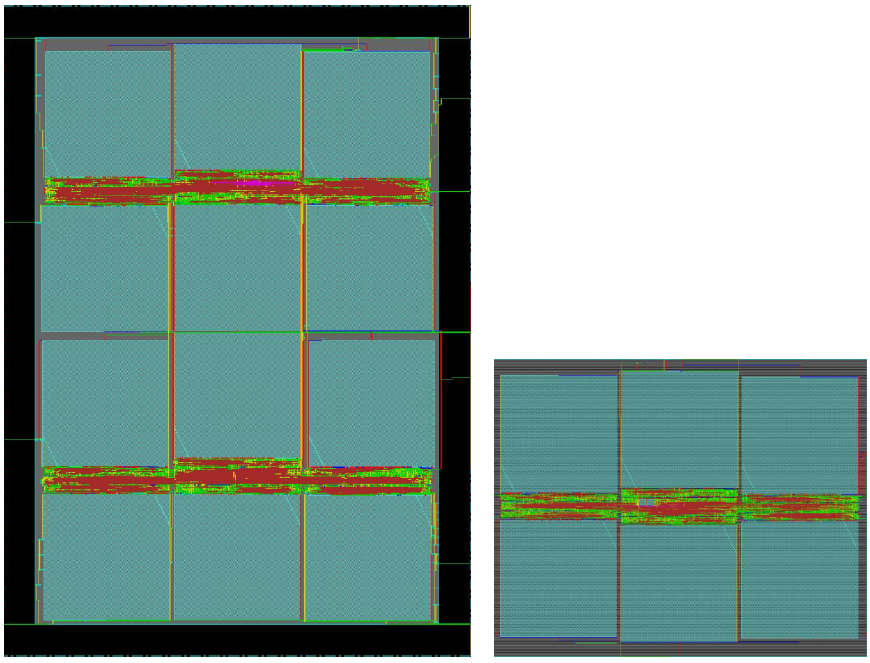
Figure 8.7: 2D 3x2 mesh NoC and possible 3D re-implementation.

leave processors and memories. ×pipes links connect either two different switches or a switch and a network interface; our choice is to cut two-dimensional topologies across switch-to-switch links, replacing the latter with an upstream and a downstream port.

Then we perform synthesis, placement and routing of the RTL in two separate runs, one per design partition. For this study, we instantiate independent clock trees in each layer. In order to come up with a system that is simulatable, we set constraints on both skew and delay, and we apply them to each tree. In this way, the clocks of the two layers are kept synchronous among each other. The final result is a system which is fully simulatable at the layout level, with proper TSV characterization.

During placement, we insert TSV macros at the proper switch boundaries. We choose the minimum TSV diameter ($4\mu m$) and pitch achievable in current technologies. The area overhead of each TSV is $64\mu m^2$ (8x8). For each bidirectional vertical switch port (*e.g.* the Up one) we have $2 \times (5 + DataWidth)$ TSVs, where the factor 2 is due to the presence of one input and one output port for bidirectionality, 5 is the number of control signals, and $DataWidth$ is the width (in bits) of the inter-switch data link. In the example of a 6x6 switch and assuming a $DataWidth$ of 28 bits, the area overhead is about 6% for ACK/NACK and 9% for STALL/GO. In exchange for this small area cost, switches can operate around 10% faster and less buffering can be deployed (saving up to 13% of the sequential area).

8.5



(a) 2D 3x2 mesh. (b) Upper half of the 3D mesh.

Figure 8.8: Layouts of the original 2D mesh and of its 3D re-implementation.

Architecture of a Mesochronous Synchronizer for 3D NoCs

Minimizing the clock skew of a clock tree in a complex 3D structure becomes at least a challenging task. Two of the possible ways to implement a 3D clock tree are (i) the design of a separate clock tree per each 2D layer, and the insertion of a single vertical structure to which to attach all the tree roots, (ii) the design of a single clock tree in one of the layers, and the deployment of many vertical vias at the terminal nodes of this tree, thus distributing the clock to all the layers. According to [167, 168], solution (ii) would be better in terms of power and skew, but unfortunately at the price of an impractical number of vertical vias, severely impacting the cost and the modularity of the design (for instance, this solution does not readily apply to the very desirable scenario where 3D chips are assembled by stacking layers provided by different vendors and possibly built with completely different processes). On the other hand, solution (i) incurs major skew issues, which demand additional synchronization every time data is exchanged among layers. Since unfortunately there is no clear solution to the problem of skew-free and modular clock distribution in 3D chips, the need for clock synchronization at the inter-layer boundaries is well motivated.

Several solutions are potentially available. Totally asynchronous NoCs are, unfortunately, very complex to design, validate and implement. Generic dual-clock FIFOs could be deployed; however, unless they are developed in full custom fashion (which implies design effort and portability drawbacks), their high implementation cost suggests using them only where absolutely needed. For example, instead of using them inside of the NoC topology for mere synchronization among clusters of routers, it may be wiser to instantiate them only at the edges of the NoC, *e.g.* at the interface of a core which is able to perform frequency scaling. To achieve functionality and flexibility at the minimum cost, mesochronous schemes are probably the most effective. We focus on the implementation of mesochronous adapters for 3D NoCs, with emphasis on circuit design, timing properties, flow control support, and implementation cost. It is worth remarking that nothing prevents the proposed scheme from also being deployed in traditional planar designs.

We leverage the baseline architecture proposed in [182]. This choice features substantial pros, including minimal complexity, ease of implementation in traditional design flows, and ability to function even during chip testing (which is typically performed at a lower frequency than the

target operating one). It is important to notice, however, that the reference work is aimed especially at handling mesochronous communication over very long and slow links (where it provides variation tolerance and high performance as additional benefits), but does not focus on short-range mesochronous synchronization, such as the one likely to be happening across 3D NoC vertical links. Therefore, it does not provide a sufficiently in-depth discussion about two issues that are crucial for any such implementation:

- Timing margins, which are key to assessing circuit robustness and to the tuning of the low-level details of the design, are not studied in enough depth in a real NoC test case, therefore preventing the related optimizations.
- Support for bidirectional communication, *i.e.* for flow control, is lacking. Mesochronous signaling is useless if proper backwards flow control cannot be issued.

Exploring and quantifying the tradeoffs required by these features is clearly key to assessing the viability of the overall approach.

8.5.1 Circuit Description

The proposed scheme is based on a synchronization circuit at the receiving end of a mesochronous link (see Figure 8.9 on the next page for a slightly simplified depiction) [182]. The circuit receives as its inputs a bundle of NoC wires representing a regular NoC link, carrying data and/or flow control commands, and a copy of the clock signal of the sender. Since the latter wire experiences the same propagation delay as the data and flow control wires, it can be used as a strobe signal for them.

The circuit is composed of a *front-end* and a *back-end*. The front-end is driven by the incoming clock signal, and strobes the incoming data and flow control wires onto a set of parallel latches in a rotating fashion, based on a counter. The back-end of the circuit leverages the local clock, and samples data from one of the latches in the front-end thanks to multiplexing logic which is also based on a counter. The rationale is to temporarily store incoming information in one of the front-end latches, using the incoming clock wire to avoid any timing problem related to the clock phase offset. Once the information stored in the latch is stable, it can be read by the target clock domain and sampled by a regular flip-flop. The counters in the front-end and back-end are initialized upon reset, after observing

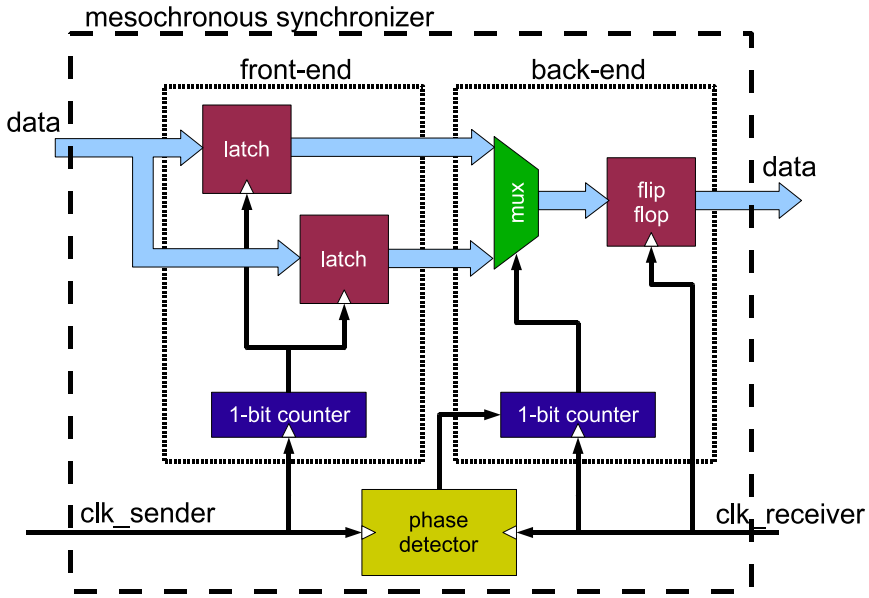


Figure 8.9: Proposed mesochronous synchronizer circuit.

the actual clock skew among the sender and receiver with a phase detector [182], so as to establish a proper offset. This is to guarantee that information can safely settle in the front-end latches before being sampled on the target domain clock. The phase detector only operates upon the system reset, but given the mesochronous nature of the link, its findings hold equally well during normal operation; the advantage is that power consumption in normal mode is negligible.

With respect to the baseline scheme [182], we apply several changes, tuning the architecture to the problem at hand. One clear feature of the 3D NoC scenario, for example, is that vertical inter-switch links are typically short and feature extremely small propagation delays, as seen above. Therefore, there is typically no need for the synchronizer to support multi-cycle propagation delays. As a result, one of the most notable architectural changes is the presence of only two latches, thus also dramatically simplifying the structure of the front-end and back-end counters to 1-bit elements. This change, which allows for large area savings, is allowed by the timing properties discussed in the following. Shall the need arise, more latches could still be deployed in case of a mesochronous link spanning over a very long distance, and requiring multiple clock cycles for signal propagation.

Figure 8.10 on the facing page summarizes the intended configuration for a system with two layers and two vertical links, one going upwards,

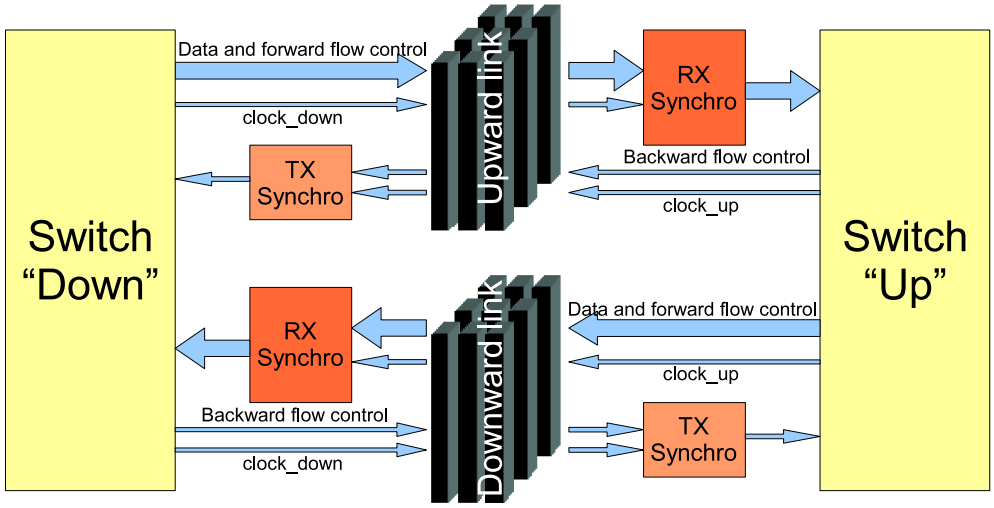


Figure 8.10: Proposed scheme for two-way synchronization across two layers.

one going downwards. For each such link, one main synchronizer (“RX Synchronizer”) must be deployed to adjust the incoming information to the new clock signal. Since a few flow control wires are travelling backwards, a smaller “TX Synchronizer” is also needed to handle them. A single block, grouping the TX and RX synchronizers by the same side of the vertical link, could be designed; this would optimize the resource usage, for instance allowing to share the phase detector. Separate synchronizers on the other hand allow for the instantiation of unidirectional (*e.g.* upwards only) links without unneeded control logic.

8.5.2 Timing Margins of the Proposed Circuit

In order not to incur metastability and not to lose data within the mesochronous synchronizer, timing constraints must be met at two points in the circuit: (i) the front-end must latch incoming data safely, (ii) the back-end must sample incoming data when it is stable.

To fulfill condition (i), the latches must become transparent at the right point in time. The ideal control signal `latch_enable` to do so would be perfectly aligned with the strobe clock `clk_sender`, upon which `data` is designed to be sampled. Unfortunately, such an ideal condition is impossible to reproduce. First, `clk_sender` must be conditioned by local signals in the mesochronous synchronizer (namely, the output `count` of the

front-end counter), which introduces a delay t_{cond}. Second, `clk_sender` and `data` may not be perfectly in sync any more if the vertical link among the sending switch and the mesochronous synchronizer is not ideal, *e.g.* if the wires/vertical vias carrying `clk_sender` are slower than those carrying `data` by $t_{routingskew}$. This means that `latch_enable` has a worst-case offset, with respect to the ideal edge on which `data` should be sampled, of $t_{cond} + t_{routingskew}$; this is an advance if $t_{routingskew}$ is negative and larger than t_{cond} (`clk_sender` wires much faster than `data` wires), and a delay otherwise.

On the other hand, the good news is that `data` is not supposed to be switching extremely close in time to the clock edges of `clk_sender`. Even if `data` were to be the direct output of registers in the sending switch, it would still take the propagation time of a flip flop before any transition could be noticed. In practice, it is likely that output buffers in the sending switch may also have some additional logic downstream of such registers, such as multiplexers to select the output of one of multiple buffer locations. Similarly, the `data` propagation delay must be designed to allow for at least a flip-flop setup time before the following clock edge, and probably a bit more to account for a bit of extra logic at the receiving buffer, such as multiplexers again. In general, the minimum transition delay of `data` after the previous clock edge of `clk_sender` can be called $t_{data_{min}}$, and the maximum can be called $t_{data_{max}}$.

In order to generate as robust a circuit as possible, we propose the circuit of Figure 8.11 on the next page to generate `latch_enable`; example waveforms are in Figure 8.12 on page 246. This circuit is an improvement with respect to [182]. Since two latches are enough to implement the front-end (see below), the counter is 1-bit, and therefore a single flip-flop, while the logic to check the counter output against a fixed value becomes a single XOR. The circuit evaluates the counter output `count` on the positive edges of `clk_sender`, but only asserts `latch_enable` when `clk_sender` goes low, *i.e.* half a clock cycle later. This shortens the critical path among `clk_sender` edges and `latch_enable` edges, *i.e.* t_{cond}, to the delay of a single NOR gate, irrespective of the delay of the counter and comparison logic - as long as these fit within a clock semiperiod, which is trivial. With this arrangement, the latches in the front-end are only transparent for one clock semiperiod every two clock periods. The conditions for correct functionality can then be summarized as:

$$t_{cond} + t_{routingskew} + t_{latchhold} < t_{data_{min}} \tag{8.5}$$

$$t_{clk} + t_{cond} + t_{routingskew} > t_{data_{max}} + t_{latchsetup} \tag{8.6}$$

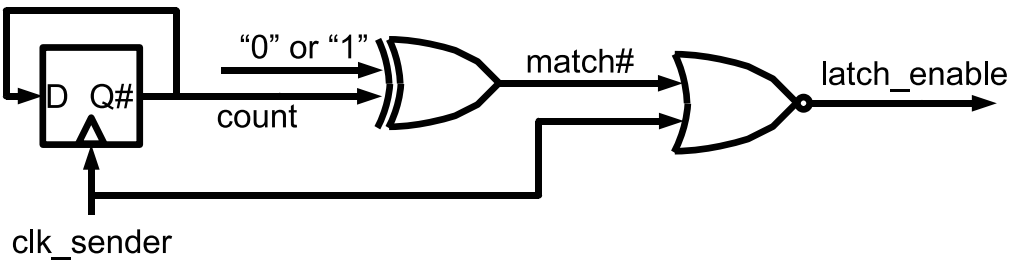


Figure 8.11: Circuit to generate the `latch_enable` control wire.

$$t_{counter} + t_{comp} < \frac{t_{clk}}{2} \tag{8.7}$$

Equation (8.5) on the preceding page expresses the fact that `latch_enable` should come early enough not to let the following piece of `data` slip in the front-end latch by mistake. Equation (8.6) on the facing page ensures that `latch_enable` comes late enough to actually let `data` settle down before latching it in the front-end. Finally, Equation (8.7) ensures that the critical path for the generation of `latch_enable` is indeed determined by the edges of `clk_sender` plus a NOR delay. Experimental results validating that these equations are actually holding will be presented in Section 8.6.2 on page 251.

Condition (ii) is easy to fulfill given a proper initialization of the counters at reset. It is a degree of freedom whether to have the back-end sample data from the upper latch at "even" clock edges and the lower latch at "odd" clock edges, or vice versa, based on the initial value imposed to the back-end counter during reset. Since the latches in the front-end are transparent one semiperiod every two periods, and opaque (frozen) for the remaining three semiperiods, it is always possible to choose a counter setup where the sampling clock edge in the back-end captures the output of the latches in a stable condition, even accounting for a large timing margin to neutralize jitter. Please note that this discussion also proves that no more than two latches in parallel are needed in the front-end, at least as long as the link propagation time remains shorter than a single clock period.

8.5.3 Adding Support for Backwards Flow Control

A key open issue to understanding whether the circuit can be used to implement a useful link for a 3D NoC is to check the overhead it mandates for a design with flow control. In fact, a unidirectional mesochronous link

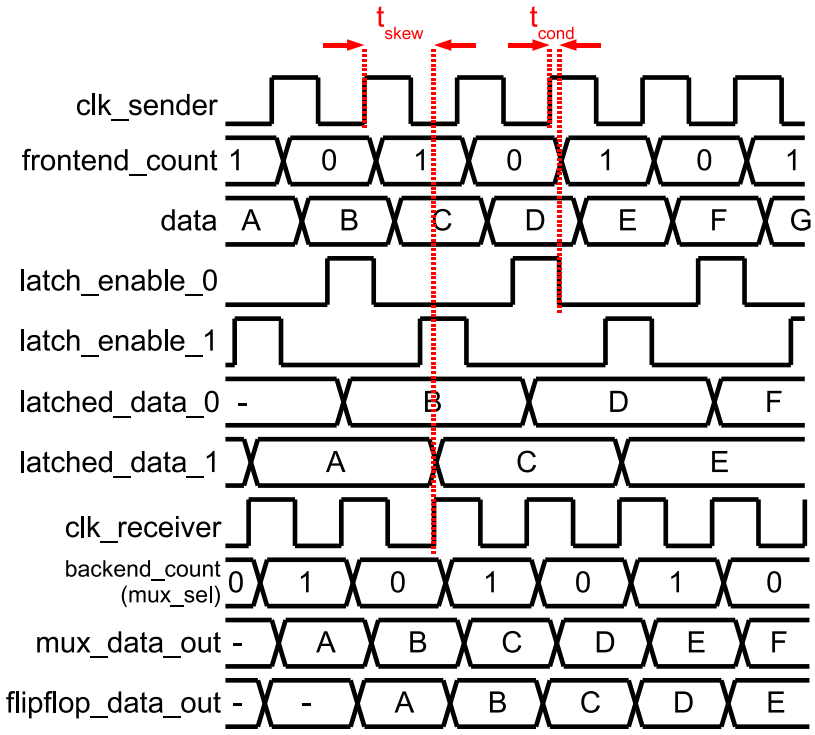


Figure 8.12: Example of the waveforms in the proposed synchronizer.

is relatively straightforward to design; once bidirectional communication must be taken into account, the implementation details and the related resource overhead become crucial. We see two properties that the system must feature to define a proper implementation of flow control over mesochronous links: (i) the system must never incur data loss or corruption, (ii) if the receiver is not busy for independent reasons (such as contention for the same switch output port), the system must be able to sustain a transfer bandwidth of one flit per clock cycle.

The solution to be applied depends on the flow control deployed in the platform, but is anyway based on the main observation that the maximum added time to convey flow control signals across a vertical link, and to resynchronize them, is in any case less than two clock cycles. Based on this information, the following solutions can be envisioned.

Backwards Flow Control in ACK/NACK

In the ACK/NACK flow control, in absence of flow control information heading back (either ACKs or NACKs), the sender "optimistically" pushes flits out. Since a copy of each flit must be stored locally, the maximum number of outstanding flits is as many as the output buffer can hold. When flow control information is eventually received, in case of NACKs, old flits are resent; if, on the other hand, it is an ACK which makes its way back to the sender, an old flit can be discarded from the output buffer, and a new one can be stored and sent.

Strictly speaking, the ACK/NACK flow control protocol does not require any corrective action to handle the timing changes introduced by a mesochronous synchronizer. The synchronizer merely delays the reception of flow control signals; this introduces no critical change in behaviour, and data safety is still guaranteed. However, changes need to be performed to support maximum bandwidth over the mesochronous link. The added latency of two clock cycles on each way (forwards and backwards) means that flits will reach their destination two cycles later, and ACKs will bounce back four cycles later than normal. To cope with this condition, output buffers in the sender need to be extended by four entries, *e.g.* from four (the minimum buffer depth to support maximum throughput in normal circumstances) to eight. This does not require any architectural change; a parameter adjustment in the output buffer is sufficient. A block diagram of the modified ACK/NACK switch is presented in Figure 8.13 on the next page. The area cost related to this change will be presented in Section 8.6.3 on page 252. No changes are required to the receiving switch, at the other side of the vertical link, unless a link in the opposite direction

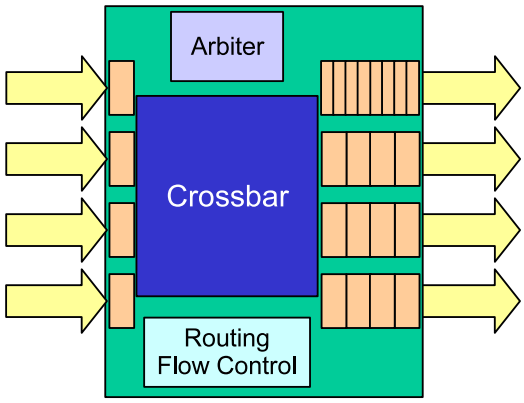


Figure 8.13: ACK/NACK modified switch block diagram. The port upstream of the vertical link has a deeper output buffer.

is also desired.

Backwards Flow Control in STALL/GO

In STALL/GO, flits are sent only as long as the STALL feedback wire is deasserted. This has two implications, which are the opposite of the ones for the ACK/NACK case. On the one hand, if STALLs are never injected by the receiver, the sender never receives them, and full transmission bandwidth can always be sustained; no circuit change is needed to meet this criterion. On the other hand, data safety is critical. STALLs are the only way the receiver can withhold the flow of flits from the sender in case they cannot be processed (such as in case of lost arbitration for a switch output port). If STALLs cannot reach the sender in time, namely within one clock cycle, flits leaving the sender while the receiver is busy simply get lost.

To cope with this situation, we extend regular input buffers by two entries (from two, which is the minimum to provide full bandwidth, to four) and change their control logic. Instead of raising the STALL wire when the buffer is actually full, we raise it when two locations are still available. This approach is conservative; for example, a 4-deep STALL/GO buffer could in principle operate forever and at full bandwidth with three or four of its locations full, provided that, at each clock cycle, a flit can be extracted to make room for a new incoming one. However, if the same buffer were to be this full and were to experience further downstream congestion, there would simply be no way to notify the sender in time and to store the flits in flight anywhere. Thus, we choose instead to raise STALLs in advance, so that, by the time the sender is notified of the congestion, at

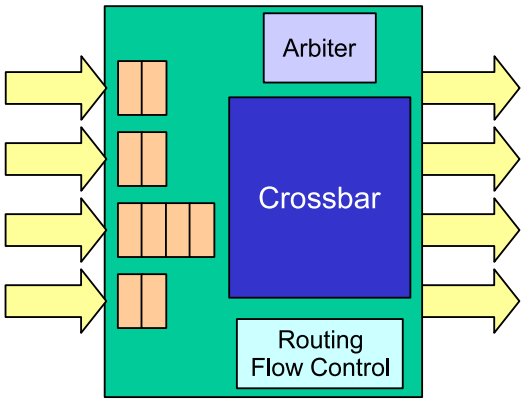


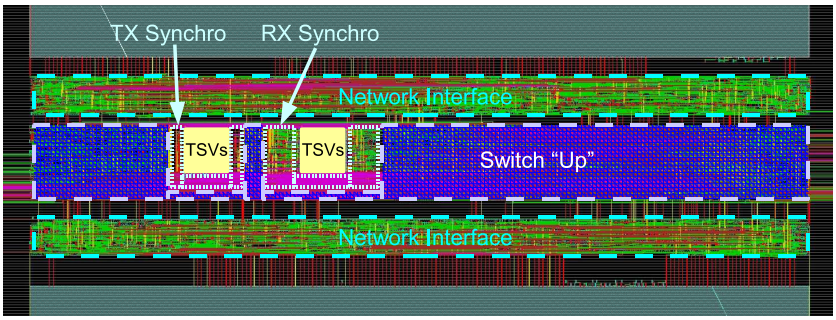
Figure 8.14: STALL/GO modified switch block diagram. The port downstream of the vertical link has a deeper input buffer and modified control logic.

most two flits are in flight, and they can still be stored. A block diagram of the modified STALL/GO switch is presented in Figure 8.14. The area cost related to this change will be presented in Section 8.6.3 on page 252. No changes are required to the sending switch, at the other side of the vertical link, unless a link in the opposite direction is also desired.

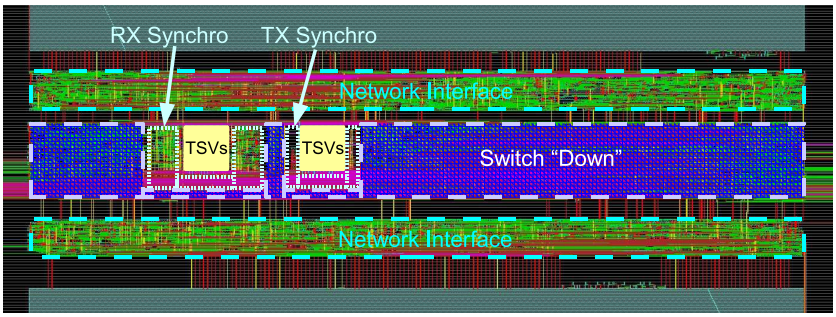
8.6 Experimental Results on Synchronizer Implementation

8.6.1 Example Mesochronous Link Implementation

We synthesize the proposed circuit with the UMC 130nm technology library and insert it into a 3D chip stack floorplan, then perform routing. Figure 8.15 on the next page summarizes the result. It is possible to see the layout of the upper and lower layers, each featuring a switch and two NIs. Two obstructions model the vertical vias (one "Up" link and one "Down") interconnecting the layers. The RX and TX synchronizers are wrapped around the via bases, and are swapped among the layers. This layout is found to be very area-efficient.



(a) Top layer.



(b) Bottom layer.

Figure 8.15: Layout of a 3D chip stack with a mesochronous NoC link. Bundles of 7x7 vias are laid among the layers, supporting 32-bit links plus flow control connections, and leaving some spare vias.

8.6.2 Timing Properties of the Synchronizer

In Section 8.5.2 on page 243, a set of conditions to be fulfilled for proper operation have been presented. Our experiments on a post-routing netlist show the following:

- Equation (8.5) on page 244 is easily fulfilled. Thanks to our optimized design (Section 8.5.2 on page 243), we measure t_{cond} values of about $60ps$. The propagation time skew among different wires of a NoC link is very low, typically yielding a $t_{routingskew}$ below $20ps$. The typical latch hold time $t_{latchhold}$ is roughly $60ps$. On the other hand, for our NoC, we measure a $t_{data_{min}}$ of about $370ps$, irrespective of whether the flow control is ACK/NACK or STALL/GO. The constraint is therefore fulfilled. (Please note that $t_{data_{max}}$, however, is dependent on the chosen flow control due to the reasons explained in Section 8.3 on page 235, and can be of up to $900ps$, imposing an operating frequency of $1GHz$ at most).

- Equation (8.6) on page 244 poses no issue. This is because the condition $t_{clk} > t_{data_{max}} + t_{latchsetup}$ is automatically met by any fully synchronous circuit. On the other hand, the term $t_{cond}+t_{routingskew}$, which appears because of the mesochronous synchronizer logic, never becomes negative in any of our test layouts. In other words, the propagation time difference among `data` and `clk_sender` is normally negligible, and in no case `clk_sender` is so much faster than `data` so as to more than offset t_{cond} (also see the bullet above). Therefore Equation (8.6) on page 244 is always verified in our tests. Please note that, even in case of a violation of this condition, the circuit could still be made to work safely by slightly increasing t_{clk}, *i.e.* slightly decreasing the operating frequency.

- Equation (8.7) on page 245 is fulfilled by a very large margin. The typical clock period of our reference NoC is larger than $1ns$ in 130nm technology, yielding a semiperiod of at least $500ps$. Given the simplicity of the counter and comparator logic for a front-end with just two latches, we observe $t_{counter} + t_{comp}$ times of less than $200ps$, well within the desired range.

Given these results, the proposed architecture proves robust under all circumstances.

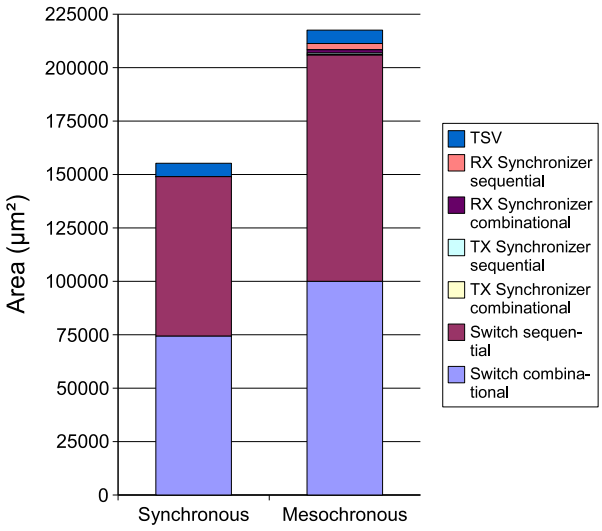
8.6.3 Silicon Cost of Proposed Synchronizer

Figure 8.16 on the next page summarizes the area overhead for the implementation of the proposed mesochronous synchronizer. The numbers refer to a post-routing circuit model. The "Synchronous" baseline comprises the area of two 32-bit 5x5 switches, with the minimum buffering required for sustaining maximum throughput under no congestion, and that of the vertical obstruction required for a unidirectional vertical link (counted twice: once per chip layer). The "Mesochronous" figures add the area overhead for supporting mesochronous clocking over such a link, namely, the buffer depth increase in one of the switches and the two TX and RX Synchronizers. It is possible to notice that the synchronizers themselves feature minimal overhead, thanks to the drastic simplification in logic allowed by the implementation of 2-latch front-ends. The RX Synchronizer is about five times larger than the TX Synchronizer, since it must handle many more wires. The largest area overhead for mesochronous clocking support is within the switches themselves, and, as expected, is mostly accounted for in the sequential area budget. ACK/NACK incurs a much larger penalty than STALL/GO, since four extra buffers have to be deployed instead of just two. Overall, the global area overhead is about 13% of the baseline configuration for STALL/GO and about 40% of the baseline configuration for ACK/NACK. Especially in the STALL/GO scenario, the area cost is minimum and the implementation seems to be clearly affordable.

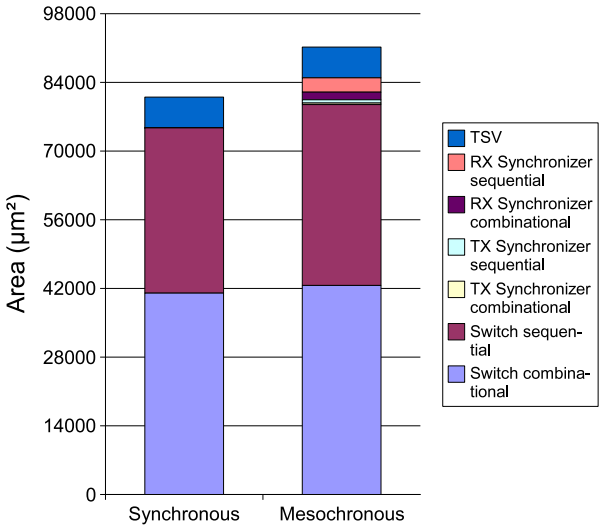
8.7 Conclusions

Stacked chips represent a promising avenue to keep pushing chip density barriers. However, being a relatively novel technology, many unknowns still exist on how to best exploit the opportunities they offer, and on how to work around their limitations. NoCs are a breaktrough technology which can help in tackling stacked chip complexity and heterogeneity, while reducing to a minimum the yield cost to be paid because of TSVs.

We have studied the performance and system-level impact of through-silicon vias as one of the possible ways to implement vertical NoC links in a highly dense manner. We have shown that, even when accounting for the coupling effects in dense vertical link bundles, the parasitics associated with TSVs are one order of magnitude smaller than those of tradi-



(a) ACK/NACK.



(b) STALL/GO.

Figure 8.16: Area cost to implement mesochronous synchronization.

tional horizontal wires, making 3D NoCs a very promising approach. We have shown how to design NoC switches with vertical ports. We have also shown that our semi-automated flow is capable of generating layouts of 3D NoCs which are fully compatible with accurate post-layout timing, area and power analysis.

We have then proposed a detailed implementation of a mesochronous synchronizer for usage in a three-dimensional chip with a NoC backbone. Since completely synchronous designs seem to be hard or impossible to achieve in stacked chips, such a device is key to correct functionality. Starting from a baseline circuit scheme, we have customized it, verified its timing properties, added flow control facilities on top of the basic circuit, and assessed the area overhead of the whole. Key advantages of the baseline circuit have been kept, such as simplicity and ability to operate correctly even during chip test at a low frequency. The experimental results show that the proposed scheme is robust and that its area cost is minimal, proving the viability of this architecture for 3D chip implementations based on NoCs.

Research on 3D NoCs is just now being undertaken, and much work remains to be done. Among the areas requiring more attention, the issue of developing software tools to perform 3D system partitioning and 3D topology design looks prominent.

CHAPTER 9

Conclusions and Future Work

9.1 Summary of Contributions

This dissertation provides a broad range of contributions to the scientific community:

- A NoC architectural library, called ×pipes. This NoC is fully featured, yet optimized for minimum area and power consumption.
- A complete, top-to-bottom design flow for the implementation of NoCs. This set of tools empowers designers to generate an optimized application-specific network, given just a set specifications which is easy to collect. The design flows includes architectural simulation and physical implementation.
- A set of techniques enabling the use of NoCs in new environments, such as in fault-tolerant applications and in 3D chips.
- Insight on the potential and on the shortcomings of NoCs, both when compared existing interconnection architectures, and when just assessing the benefits and costs of alternative NoC implementations.

9.2

Conclusions

Emerging consumer applications demand high performance, low power, low cost and high reliability from the current and the upcoming generations of embedded devices. MPSoCs, *i.e.* highly integrated, often heterogeneous chips, have a leading role in fulfilling these demands. However, their complexity is becoming so challenging that new techniques and mechanisms to design their on-chip interconnection are greatly needed. NoCs have enjoyed increasing acceptance as the most scalable answer to these shortcomings. However, as a technology of recent inception, the consensus on their full maturity has not been reached yet.

This dissertation proves fully that NoCs are a viable, efficient, scalable, predictable, flexible answer to the on-chip interconnection woes. We have shown a complete design flow (Figure 9.2 on the facing page) that, leveraging upon a library of NoC components (Figure 9.1 on the next page), is capable of generating placed&routed instances of a NoC. Our proposed design flow is conceived to optimize design time. Thanks to a detailed modeling activity, by taking into account aspects such as the resulting chip floorplan, and with careful analysis of on-chip wiring, we dramatically increase the design closure chances and thus the need for project re-spins. Our NoC instances have been verified as fully functional. We have also shown some implementations in 65nm technology, and outlined some scaling properties of these networks.

We also concretely extend the domain of applicability of NoCs, by presenting techniques for increased system fault tolerance and by outlining the path towards having NoCs as the backbone for stacked 3D chips.

9.3

Future Work

Despite the large body of research devoted to NoCs in the last years, many open challenges remain. We identify several major areas for upcoming research:

- There is a consensus on the system-level need of integrating blocks at different operating frequencies, which is also often called a GLOBALLY ASYNCHRONOUS, LOCALLY SYNCHRONOUS (GALS) paradigm [17]. However, as of today, the dilemma of which NoC

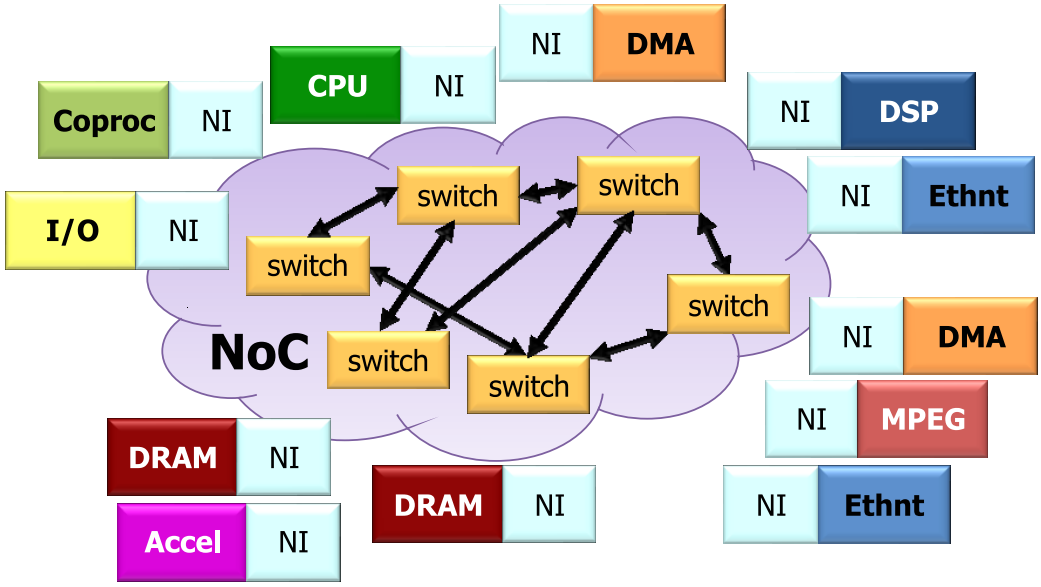


Figure 9.1: Conceptual view of a Network-on-Chip.

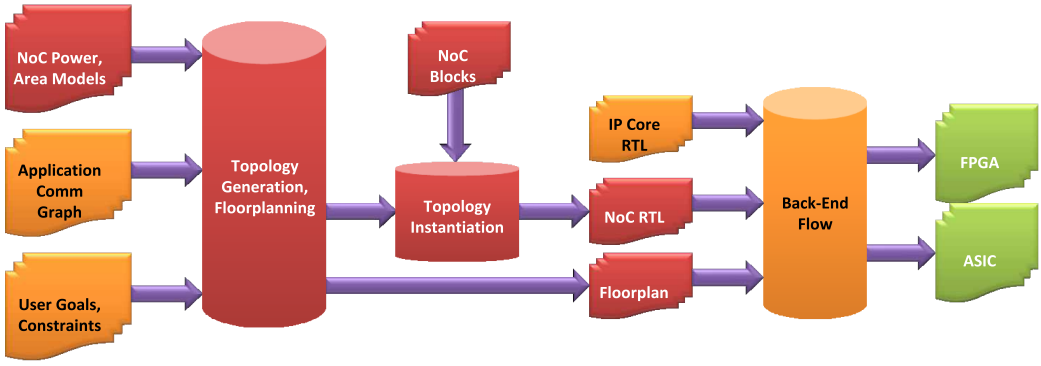


Figure 9.2: The complete proposed NoC design flow.

alternative to choose in this context has not been fully cleared: synchronous choices (interspersed with a minimum amount of clock converters) have been claiming rapid development times and minimum overhead, while the alternatives claim, for example, superior robustness to process variance [19, 237, 171]. Furthermore, to the best of our knowledge, none of the contending approaches has yet comprehensively tackled the issue of dynamic frequency (and possibly also voltage) scaling [7, 21]; some open questions in this research area include how to best devise mechanisms for NoC partitioning, how to best control the operating parameters of each NoC partition statically (*e.g.* with CAD tooling for heterogeneous NoCs) or at runtime, *etc.*.

- NoCs for 3D chips are an emerging trend [233, 238, 239]. Work remains to be done. Moreover, since the manufacturing technology is not fully mature, many crucial design parameters (such as the attainable density, yield and speed of vertical interconnects) remain unclear. Depending on these parameters, many different NoC architectures and topologies can be envisioned. Some of the key upcoming challenges in this field include the potential need for pervasive fault tolerance, the design of a new generation of CAD tools for NoC topology exploration, and the issue of clock domain synchronization (if not even of bridging among different signaling methods) across stack layers.

- NoC reconfiguration is also a broad topic [21]. While several approaches have been published to reconfiguring NoCs (either completely, on FPGAs, or just with new routing tables, in ASICs) [240, 241], our impression is that many links are missing before fully reconfigurable, hazard- and deadlock-free, low-resource-overhead NoCs can be available. Furthermore, even then, the larger problem of efficiently deciding how to reconfigure NoCs at runtime, based on mutating application demands, will be very challenging.

APPENDIX A

Author's Relevant Publications

The appendix lists the works in which the author has been involved and which are relevant to the dissertation's subject.

Journal papers:

- **F. Angiolini**, P. Meloni, S. Carta, L. Raffo, L. Benini, "A Layout-Aware Analysis of Networks-on-Chip and Traditional Interconnects for MPSoCs", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Mar 2007, Vol. 26 Issue 3, pp. 421-434*

- R. Tamhankar, S. Murali, S. Stergiou, A. Pullini, **F. Angiolini**, L. Benini, G. De Micheli, "Timing Error Tolerant Network-on-Chip Design Methodology", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Jul 2007, Vol. 26 Issue 7, pp. 1297-1310*

- P. Meloni, I. Loi, **F. Angiolini**, S. Carta, M. Barbaro, L. Raffo, L. Benini, "Area and Power Modeling for Networks-on-Chip with Layout Awareness", *VLSI Design, Vol. 2007, Article ID 50285, 12 pages*

- A. Pullini, **F. Angiolini**, S. Murali, D. Atienza, G. De Micheli, L. Benini, "Bringing NoCs to 65nm", *IEEE Micro, Sep-Oct 2007, Vol. 27 Issue 5, pp. 75-85*

- S. Mahadevan, **F. Angiolini**, J. Sparsø, L. Benini, J. Madsen, "A Reactive and Cycle-True IP Emulator for MPSoC Exploration", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Jan 2008, Vol. 27 Issue 1, pp. 109-122*

- D. Atienza, **F. Angiolini**, S. Murali, A. Pullini, L. Benini, G. De Micheli, "Network-On-Chip Design and Synthesis Outlook", *to be published in Elsevier Integration, the VLSI Journal*

Conference papers:

- M. Loghi, **F. Angiolini**, D. Bertozzi, L. Benini, R. Zafalon, "Analyzing On-Chip Communication in a MPSoC Environment", *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition 2004, Paris, France, Feb 16-20, 2004, pp. 752-757*

- M. Ruggiero, **F. Angiolini**, F. Poletti, D. Bertozzi, L. Benini, R. Zafalon, "Scalability Analysis of Evolving SoC Interconnect Protocols", *Proceedings of the 2004 International Symposium on System-on-Chip, Tampere, Finland, Nov 16-18, 2004, pp. 169-172*

- S. Mahadevan, **F. Angiolini**, M. Storgaard, R. G. Olsen, J. Sparsø, J. Madsen, "A Network Traffic Generator Model for Fast Network-on-Chip Simulation", *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition 2005, Munich, Germany, Mar 7-11, 2005, pp. 780-785*

- S. Stergiou, **F. Angiolini**, S. Carta, L. Raffo, D. Bertozzi, G. De Micheli, "×pipes Lite: A Synthesis Oriented Design Flow For Networks on Chips", *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition 2005, Munich, Germany, Mar 7-11, 2005, pp. 1188-1193*

- A. Pullini, **F. Angiolini**, D. Bertozzi, L. Benini, "Fault Tolerance Overhead in Network-on-Chip Flow Control Schemes", *Proceedings of 18th Annual Symposium on Integrated Circuits and System Design (SBCCI) 2005, Florianópolis, Brazil, Sep 4-7, 2005, pp. 224-229*

- **F. Angiolini**, P. Meloni, D. Bertozzi, L. Benini, S. Carta, L. Raffo, "Networks on Chips: A Synthesis Perspective", *Proceedings of the Parallel Computing (ParCo) Conference 2005, Málaga, Spain, Sep 13-16, 2005*

- S. Srinivasan, **F. Angiolini**, M. Ruggiero, N. Vijaykrishnan, L. Benini, "Simultaneous Memory and Bus Partitioning for SoC Architectures", *Proceedings of the IEEE International SOC Conference (SOCC) 2005, Washington (DC), USA, Sep 25-28, 2005, pp. 125-128*

- **F. Angiolini**, S. Mahadevan, J. Madsen, L. Benini, J. Sparsø, "Realistically Rendering SoC Traffic Patterns with Interrupt Awareness", *Proceedings of the IFIP VLSI-SOC Conference 2005, Perth (WA), Australia, Oct 17-19, 2005, pp. 211-216*

- **F. Angiolini**, P. Meloni, S. Carta, L. Benini, L. Raffo, "Contrasting a NoC and a Traditional Interconnect Fabric with Layout Awareness", *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition 2006, Munich, Germany, Mar 6-10, 2006, pp. 124-129*

- **F. Angiolini**, J. Ceng, R. Leupers, F. Ferrari, C. Ferri, L. Benini, "An Integrated Open Framework for Heterogeneous MPSoC Design Space Exploration", *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition 2006, Munich, Germany, Mar 6-10, 2006, pp. 1145-1150*

- P. Meloni, S. Carta, R. Argiolas, L. Raffo, **F. Angiolini**, "Area and Power Modeling Methodologies for Networks-on-Chip", *Proceedings of the Nano-Net Conference 2006, Lausanne, Switzerland, Sep 14-16, 2006*

- **F. Angiolini**, D. Atienza, S. Murali, L. Benini, G. De Micheli, "Reliability Support for On-Chip Memories Using Networks-on-Chip", *Proceedings of the International Conference on Computer Design (ICCD) 2006, San José (CA), USA, Oct 1-4, 2006*

- S. Murali, P. Meloni, **F. Angiolini**, D. Atienza, S. Carta, L. Benini, G. De Micheli, L. Raffo, "Designing Message-Dependent Deadlock Free Networks on Chips for Application-Specific Systems on Chips", *Proceedings of the IFIP VLSI-SOC Conference 2006, Nice, France, Oct 16-18, 2006, pp. 158-163*

- S. Murali, P. Meloni, **F. Angiolini**, D. Atienza, S. Carta, L. Benini, G. De Micheli, L. Raffo, "Designing Application-Specific Networks on Chips with Floorplan Information", *Proceedings of the International Conference on Computer Aided Design (ICCAD) 2006, San Jos (CA), USA, Nov 5-9, 2006, pp. 355-362*

- S. Murali, R. Tamhankar, **F. Angiolini**, A. Pullini, D. Atienza, L. Benini, G. De Micheli, "Comparison of a Timing-Error Tolerant Scheme with a Traditional Re-transmission Mechanism for Networks on Chips", *Proceedings of the 2006 International Symposium on System-on-Chip, Tampere, Finland, Nov 13-16, 2006, pp. 27-30*

- A. Pullini, **F. Angiolini**, P. Meloni, D. Atienza, S. Murali, L. Raffo, G. De Micheli, L. Benini, "NoC Design and Implementation in 65nm Technology", *Proceedings of the 1st ACM/IEEE International Symposium on Networks-on-Chip, Princeton (NJ), USA, May 7-9, 2007, pp. 273-282*

- I. Loi, **F. Angiolini**, L. Benini, "Supporting vertical links for 3D networks on chip: toward an automated design and analysis flow", *Proceedings of the Nano-Net Conference 2007, Catania, Italy, Sep 24-26, 2007*

- I. Loi, **F. Angiolini**, L. Benini, "Developing Mesochronous Synchronizers to Enable 3D NoCs", *to be published in the Design, Automation and Test in Europe Conference and Exhibition 2008*

Patents:

- **F. Angiolini**, S. Murali, D. Atienza, G. De Micheli, "Mechanisms to Add Reliability, Power Management and Load Balancing to Networks-on-Chip", *PPA 2006*

Bibliography

[1] Infineon, "XMM 6080 Platform (HEDGE)," 2008, http://www.infineon.com.

[2] *AMBA Specification*, ARM Inc., May 1999.

[3] SIA Semiconductor Industry Association, "The international technology roadmap for semiconductors," SIA Semiconductor Industry Association, Tech. Rep., 2002, http://public.itrs.net/.

[4] "AMBA 3 AXI overview," ARM Ltd., 2005, http://www.arm.com/products/solutions/AMBA3AXI.html.

[5] "STBus," STMicroelectronics, http://www.st.com/.

[6] D. Wingard, "Micronetwork-based integration for SOCs," in *Proceedings of Design Automation Conference (DAC)*, 2001, pp. 673–677.

[7] W. Wolf, "The Future of Multiprocessor Systems-on-Chips," in *Proc. DAC*, June 2004, pp. 681–685.

[8] "Philips nexperia - highly integrated programmable system-on-chip (mpsoc)," 2004, http://www.semiconductors.philips.com/products/nexperia.

[9] Texas Instruments (TI), "OMAP Platform," 2004, http://focus.ti.com/omap/docs/.

[10] "St nomadik multimedia processor," 2004, http://www.st.com/stonline/prodpres/dedicate/proc/proc.htm.

[11] "AMBA AHB overview," ARM Ltd., 2005, http://www.arm.com/products/solutions/AMBA_Spec.html.

[12] OCP International Partnership (OCP-IP), "Open core protocol standard," 2003, http://www.ocpip.org/home.

[13] MIPS Technologies, "MIPS Cores," http://www.mips.com/products/cores/.

[14] N. S. Kim, T. Austin, D. Blaauw, T. Mudge, K. Flautner, J. Hu, M. Irwin, M. Kandemir, and N. Vijaykrishnan, "Leakage current: Moore's law meets static power," *Computer*, vol. 36, no. 12, pp. 65–77, December 2003.

[15] L. Benini and G. De Micheli, "Networks on chip: a new SoC paradigm," *IEEE Computer*, vol. 35, no. 1, pp. 70–78, January 2002.

[16] W. J. Dally and B. Towles, "Route packets, not wires: On-chip interconnection networks," in *Proceedings of the 38th Design Automation Conference*, June 2001, pp. 684–689.

[17] A. Jantsch and H. Tenhunen, Eds., *Networks on chip*. Hingham, MA, USA: Kluwer Academic Publishers, 2003.

[18] A. Adriahantenaina, H. Charlery, A. Greiner, L. Mortiez, and C. Zeferino, "Spin: a scalable, packet switched, on-chip micro-network," in *Proceedings of the IEEE Design and Test in Europe Conference (DATE)*, 2003, pp. 70–73.

[19] K. Goossens, J. Dielissen, and A. Radulescu, "Aethereal network on chip: Concepts, architectures, and implementations," *IEEE Design and Test of Computers*, vol. 22, no. 5, pp. 414–421, September-October 2005.

[20] F. Angiolini, P. Meloni, S. Carta, L. Raffo, and L. Benini, "A layout-aware analysis of networks-on-chip and traditional interconnects for MPSoCs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 26, pp. 421–434, March 2007.

[21] L. Benini and G. D. Micheli, Eds., *Networks on chips: Technology and Tools*. San Francisco, CA, USA: Morgan Kaufmann Publishers, 2006.

[22] H. Zhang, V. Prabhu, V. George, M. Wan, M. Benes, A. Abnous, , and J. Rabaey, "A 1-v heterogenous reconfigurable dsp ic for wireless baseband digital signal processing," *IEEE Journal of Solid State Circuits*, vol. 35, no. 11, pp. 1697–1704, Nov 2000.

[23] W. J. Dally and S. Lacy, "Vlsi architecture: Past, present, and future," in *ARVLSI '99: Proceedings of the 20th Anniversary Conference on Advanced Research in VLSI*. Washington, DC, USA: IEEE Computer Society, 1999, p. 232.

[24] S. Kolson, A. Jantsch, J.-P. Soininen, M. Forsell, M. Millberg, J. Oberg, K. Tiensyrja, and A. Hemani, "A network on chip architecture and design methodology," in *Proceedings of IEEE Annual Symposium on VLSI*. IEEE Computer Society, April 2002, pp. 105–112.

[25] P. Guerrier and A. Greiner, "A generic architecture for on-chip packet-switched interconnections," in *Design Automation and Test in Europe, DATE'00*, March 2000, pp. 250 – 256.

[26] D. Wiklund and D. Liu, "SoCBUS: Switched network on chip for hard real time embedded systems," in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS03)*. IEEE, 2003.

[27] F. Karim, A. Nguyen, S. Dey, and R. Rao, "On-chip communication architecture for OC-768 network processors," in *Proceedings of the Design Automation Conference (DAC)*, 2001, pp. 678 – 683.

[28] S.-J. Lee, S.-J. Song, K. Lee, J.-H. Woo, S.-E. Kim, B.-G. Nam, and H.-J. Yoo, "An 800MHz star-connected on-chip network for application to systems on a chip," in *Digest of Technical Papers of the 2003 IEEE International Solid-State Circuits Conference (ISSC)*. IEEE Computer Society, 2003, pp. 468–469.

[29] A. Radulescu, J. Dielissen, K. Goossens, E. Rijpkema, and P. Wielage, "An efficient on-chip network interface offering guaranteed services, shared-memory abstraction, and flexible network configuration," in *Proceedings of the 2004 Design, Automation and Test in Europe Conference (DATE)*. IEEE, 2004.

[30] D. Siguenza-Tortosa and J. Nurmi, "Proteo: a new approach to network-on-chip," in *Proceedings of IASTED International Conference on Communication Systems and Network*, Malaga (Spain), 2002.

[31] A. Andriahantenaina and A. Greiner, "Micro-network for SoC : Implementation of a 32-port spin network," in *The Proceedings of Design, Automation and Test in Europe Conference and Exhibition*. IEEE, 2003, pp. 1128– 1129.

[32] K. Lee, S.-J. Lee, S.-E. Kim, H.-M. Choi, D. Kim, S. Kim, M.-W. Lee, and H.-J. Yoo, "A 51mW 1.6GHz on-chip network for low-power heterogeneous SoC platform," in *Digest of Technical Papers of the 2004 IEEE International Solid-State Circuits Conference (ISSC)*. IEEE Computer Society, 2004, pp. 152–518.

[33] C. A. Zeferino and A. A. Susin, "SoCIN: A parametric and scalable network-on-chip," in *Proceedings of the 16th Symposium on Integrated Circuits and Systems Design (SBCCI03)*, 2003, pp. 34–43.

[34] J. Xu, W. Wolf, J. Henkel, and S. Chakradhar, "A methodology for design, modeling, and analysis of Networks-on-Chip," in *Proceedings of the 2005 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE Computer Society, 2005, pp. 1778–1781.

[35] G. Sassatelli, S. Riso, L. Torres, M. Robert, and F. Moraes, "Packet-switching network-on-chip features exploration and characterization," in *Proceedings of the International Conference on Very Large Scale Integration System-on-Chip (VLSI-SoC 2005)*, 2005, pp. 403–408.

[36] W. Hang-Sheng, P. Li-Shiuan, and S. Malik, "A technology-aware and energy-oriented topology exploration for on-chip networks," in *Proceedings of the Design Automation and Test in Europe Conference (DATE)*. IEEE, March 2005.

[37] E. Nilsson, M. Millberg, J. Oberg, and A. Jantsch, "Load distribution with the proximity congestion awareness in a networks on chip," 2003.

[38] M. MillBerg, E. Nilsson, R. Thid, and A. Jantsch, "Guaranteed bandwidth using looped containers in temporally disjoint networks within the Nostrum network on chip," in *Proceedings of the Design, Automation and Testing in Europe Conference (DATE'04)*. IEEE, 2004.

[39] J. Liu, L. Zheng, and H. Tenhunen, "A guaranteed-throughput switch for network-on-chip," in *International Symposium on System-on-Chip*, Nov 2003, pp. 31–34.

[40] V. Chandra, A. Xu, H. Schmit, and L. Pileggi, "An interconnect channel design methodology for high performance integrated circuits," in *DATE '04: Proceedings of the conference on Design, automation and test in Europe*. Washington, DC, USA: IEEE Computer Society, 2004, p. 21138.

[41] S. Khorsandi and A. Leon-Garcia, "Robust non-probabilistic bounds for delay and throughput in credit-based flow control," in *INFOCOM*, 1996, pp. 577–584.

[42] W. Dally and B. Towles, *Principles and Practices of Interconnection Networks*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003.

[43] C. A. Zeferino, M. E. Kreutz, L. Carro, and A. A. Susin, "A study on communication issues for systems-on-chip," in *SBCCI '02: Proceedings of the 15th symposium on Integrated circuits and systems design*. Washington, DC, USA: IEEE Computer Society, 2002, p. 121.

[44] A. Rădulescu, J. Dielissen, S. González Pestana, O. P. Gangwal, E. Rijpkema, P. Wielage, and K. Goossens, "An efficient on-chip network interface offering guaranteed services, shared-memory abstraction, and flexible network programming," *IEEE Transactions on CAD of Integrated Circuits and Systems*, vol. 24, no. 1, pp. 4–17, Jan. 2005.

[45] D. Rostislav, V. Vishnyakov, E. Friedman, and R. Ginosar, "An asynchronous router for multiple service levels networks on chip," in *Proceedings of the 11th IEEE International Symposium on Asynchronous Circuits and Systems*, 2005, pp. 44–53.

[46] C. A. Zeferino, F. G. M. E. Santo, and A. A. Susin, "Paris: a parameterizable interconnect switch for networks-on-chip," in *SBCCI '04: Proceedings of the 17th symposium on Integrated circuits and system design*. New York, NY, USA: ACM, 2004, pp. 204–209.

[47] E. Rijpkema, K. Goossens, A. Rădulescu, J. Dielissen, J. van Meerbergen, P. Wielage, and E. Waterlander, "Trade-offs in the design of a router with both guaranteed and best-effort services for networks on chip," *IEE Proceedings: Computers and Digital Techniques*, vol. 150, no. 5, pp. 294–302, Sept. 2003.

[48] J. Chan and S. Parameswaran, "Nocgen:a template based reuse methodology for networks on chip architecture," in *Proceedings of the 17th International Conference and VLSI Design*, 2004, pp. 717 – 720.

[49] R. R. Tamhankar, S. Murali, and G. D. Micheli, "Performance driven reliable link design for networks on chips," in *ASP-DAC '05: Proceedings of the 2005 conference on Asia South Pacific design automation*. New York, NY, USA: ACM, 2005, pp. 749–754.

[50] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A framework for architectural-level power analysis and optimizations," in *Proceedings of the 27th International Symposium on Computer Architecture (ISCA)*, 2000, pp. 83–94.

[51] W. Ye, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin, "The design and use of simplepower: A cycle-accurate energy estimation tool," in *Proceedings of the 34th Design Automation Conference (DAC'00)*, 2000, pp. 340–345.

[52] A. Bona, V. Zaccaria, and R. Zafalon, "System level power modeling and simulation of high-end industrial network-on-chip," in *Proceedings of Design, Automation and Testing in Europe Conference 2004 (DATE04)*. IEEE, Febuary 2004, pp. 318–323.

[53] J. S. Kim, M. B. Taylor, J. Miller, and D. Wentzlaff, "Energy characterization of a tiled architecture processor with on-chip networks," in *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, 2003, pp. 424–427.

[54] H.-S. Wang, X. Zhu, L.-S. Peh, and S. Malik, "Orion: a power-performance simulator for interconnection networks," in *Proceedings of 35th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE/ACM, November 2002, pp. 294–305.

[55] H.-S. Wang, L.-S. Peh, and S. Malik, "A technology-aware and energy-oriented topology exploration for on-chip networks," in *Proceedings of Design, Automation and Testing in Europe Conference 2005 (DATE05)*. IEEE, March 2005, pp. 1238–1243.

[56] N. Eisley and L.-S. Peh, "High-level power analysis of on-chip networks," in *Proceedings of the 7th International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, 2004, pp. 104–115.

[57] J. Chan and S. Parameswaran, "NoCEE: Energy macro-model extraction methodology for network on chip routers," in *Proceedings of the IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, 2005, pp. 254–259.

[58] G. Palermo and C. Silvano, "PIRATE: A framework for power/performance exploration of network-on-chip architectures," in *Proceedings of the 14th Intl. Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, 2004, pp. 521–531.

[59] T. T. Ye, L. Benini, and G. D. Micheli, "Analysis of power consumption on switch fabrics in network routers," in *Proceedings of the 36th Design Automation Conference (DAC'02)*, 2002, pp. 524–529.

[60] C. S. Patel, S. M. Chai, S. Yalamanchili, and D. E. Schimmel, "Power constrained design of multiprocessor interconnection networks," in *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD)*, 1997, pp. 408–416.

[61] Y. Zhang and M. Irwin, "Power and performance comparison of crossbars and buses as on-chip interconnect structures," in *Asilomar Conf. on Signals, Systems and Computers*, October 1999, pp. 378–383.

[62] F. Petrot, D. Hommais, and A. Greiner, "A simulation environment for core based embedded systems," in *Annual Simulation Symposium*, Atlanta, April 1997, pp. 86–91.

[63] M. Caldari, M. Conti, M. Coppola, S. Curaba, L. Pieralisi, and C. Turchetti, "Transaction-level models for AMBA bus architecture using SystemC 2.0," in *Proceedings of the IEEE Design and Test in Europe Conference (DATE)*, 2003, pp. 26–31.

[64] X. Zhu and S. Malik, "A hierarchical modeling framework for on-chip communication architectures," in *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design (ICCAD '02)*. New York, NY, USA: ACM Press, 2002, pp. 663–671.

[65] M. Coppola, S. Curaba, M. Grammatikakis, and G. Maruccia, "IP-SIM: SystemC 3.0 enhancements for communication refinement," in *Proc. of IEEE Design and Test in Europe Conference (DATE)*, 2003, pp. 106–111.

[66] J. Rowson and A. Sangiovanni-Vincentelli, "Interface-based design," in *Design Automation Conference (DAC)*, June 1997, pp. 178–183.

[67] K. Hines and G. Borriello, "Optimizing communication in embedded system cosimulation," in *Workshop on Hardware/Software Codesign*, 1997, pp. 121–125.

[68] M. Gasteier and M. Glesner, "Bus-based communication synthesis," *ACM Trans. on Design Automation of Electronic Systems*, vol. 4, pp. 1–11, January 1999.

[69] T. Yen and W. Wolf, "Communication synthesis for distributed embedded systems," in *Int. Conf. on Cumputer-Aided Design*, November 1995, pp. 288–294.

[70] K. Lahiri, A. Raghunathan, and S. Dey, "System-level performance analysis for designing on-chip communication architectures," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 20, pp. 768–783, June 2001.

[71] K. Richter, M. Jersak, and R. Ernst, "A formal approach to MPSoC performance verification," *IEEE Computer*, vol. 36, pp. 60–67, April 2003.

[72] V. D'Silva, S. Ramesh, and A. Sowmya, "Synchronous protocol automata: a framework for modelling and verification of SoC communication architectures," in *Proc. of Design, Automation and Test in Europe Conf. (DATE)*, 2004, pp. 390–395.

[73] J. Ou, C. Seonil, and V. Prasanna, "Performance modeling of reconfigurable SoC architectures and energy-efficient mapping of a class of application," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2003, pp. 241–250.

[74] K. Lahiri, A. Raghunathan, and S. Dey, "Evaluation of the traffic-performance characteristics of systems-on-chip communication architectures," in *Conf. on VLSI Design*, January 2001, pp. 29–35.

[75] V. Lahtinen, E. Salminen, K. Kuusilinna, and T. Hamalainen, "Comparison of synthesized bus and crossbar interconnection architectures," in *Int. Symp. on Circuits and Systems (ISCAS)*, May 2003, pp. (5): 25–28.

[76] K. Ryu, E. Shin, and V. Mooney, "A comparison of five different multiprocessor SoC bus architectures," in *EUROMICRO*, September 2001, pp. 202–209.

[77] M. Loghi, F. Angiolini, D. Bertozzi, L. Benini, and R. Zafalon, "Analyzing on-chip communication in a mpsoc environment," in *Proceedings of the conference on Design, automation and test in Europe*. Paris, France: IEEE Computer Society, February 2004, p. 20752.

[78] M. Ruggiero, F. Angiolini, F. Poletti, D. Bertozzi, L. Benini, and R. Zafalon, "Scalability analysis of evolving SoC interconnect protocols," in *Proceedings of the 2004 International Symposium on System-on-Chip*, 2004, pp. 169–172.

[79] L. Benini, D. Bertozzi, A. Bogliolo, F. Menichelli, and M. Olivieri, "Mparm: Exploring the multi-processor SoC design space with SystemC," *The Journal of VLSI Signal Processing*, vol. 41, no. 2, pp. 169–182, September 2005.

[80] SimpleScalar LLC, "SimpleScalar," SimpleScalar LLC, http://www.simplescalar.com.

[81] The SoCLib Partners, "SoCLib," The SoCLib Partners, http://soclib.lip6.fr.

[82] The Liberty Research Group, "The Liberty Simulation Environment," The Liberty Research Group, http://liberty.princeton.edu/.

[83] The MicroLib Community, "MicroLib," The MicroLib Community, http://microlib.org/.

[84] The SkyEye Community, "SkyEye," The SkyEye Community, http://www.skyeye.org.

[85] Synopsys, "System Studio," http://www.synopsys.com.

[86] CoWare Inc., "ConvergenSC," CoWare Inc., http://www.coware.com/products/.

[87] ARM Ltd., "RealView MaxSim," ARM Ltd., http://www.arm.com/products/DevTools/MaxSim.html.

[88] Prosilog, "Magillem," Prosilog, www.prosilog.com.

[89] Virtio, "Virtual Platforms," Virtio, www.virtio.com.

[90] P. Mishra, N. Dutt, and A. Nicolau, "Functional abstraction driven design space exploration of heterogeneous programmable architectures," in *ISSS '01: Proceedings of the 14th international symposium on Systems synthesis*. New York, NY, USA: ACM Press, 2001, pp. 256–261.

[91] CoWare Inc., "LISATek," CoWare Inc., http://www.coware.com/products/.

[92] M. B. Sandro Rigo, Guido Araujo and R. Azevedo, "Archc: A systemc-based architecture description language," in *16th Symposium on Computer Architecture and High Performance Computing - Foz do Iguacu, Brazil*, October 2004.

[93] Target Compiler Technologies N.V., "The Chess/Checker Retargetable DSP Environment," http://www.retarget.com.

[94] N. Cheung, J. Henkel, and S. Parameswaran, "Rapid configuration and instruction selection for an asip: A case study," in *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*. Washington, DC, USA: IEEE Computer Society, 2003, p. 10802.

[95] A. Kitajima, M. Itoh, J. Sato, A. Shiomi, Y. Takeuchi, and M. Imai, "Effectiveness of the ASIP design system PEAS-III in design of pipelined processors," in *ASP-DAC '01: Proceedings of the 2001 conference on Asia South Pacific design automation*. New York, NY, USA: ACM Press, 2001, pp. 649–654.

[96] K. Lahiri, A. Raghunathan, and S. Dey, "System level performance analysis for designing on-chip communication architectures," *Trans. on Computer Aided-Design of Integrated Circuits and Systems*, vol. 20, no. 6, pp. 768–783, 2001.

[97] D. Wiklund, S. Sathe, and D. Liu, "Network on chip simulations for benchmarking," in *Proceedings of the 4th IEEE International Workshop on System-on-Chip for Real-Time Applications (IWSOC)*. IEEE, 2004.

[98] E. Bolotin, I. Cidon, R. Ginosar, and A. Kolodny, "QNoC: QoS architecture and design process for network on chip," in *Journal of Systems Architecture*. Elsevier, 2004.

[99] G. V. Varatkar and R. Marculescu, "On-chip traffic modeling and synthesis for MPEG-2 video applications," *Transactions on Very Larget Scale Integration (VLSI) Systems*, vol. 12, no. 1, pp. 108–119, JANUARY 2004.

[100] S. Avallone, A. Pescape, and G. Ventre, "Analysis and experimentation of internet traffic generator," in *Proceedings of FTDCS*, 2004.

[101] N. Genko, D. Atienza, G. De Micheli, J. M. Mendias, R. Hermida, and F. Catthoor, "A complete network-on-chip emulation framework," in *In the Proceedings of Design, Automation and Test in Europe (DATE '05)*. Munich, Germany: IEEE Press, March 2005.

[102] L. Cai and D. Gajski, "Transaction level modeling in system level design," Center for Embedded Computer Systems, Information and Computer Science, University of California, Irvine," CECS Technical Report 03-10, March 2003.

[103] F. Fummi, P. Gallo, S. Martini, G. Perbellini, M. Poncino, and F. Ricciato, "A timing-accurate modeling and simulation environment for networked embedded systems," in *Proceedings of the 42th Design Automation Conference (DAC)*, June 2003, pp. 42–47.

[104] T. Grötker, S. Liao, G. Martin, and S. Swan, *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[105] O. Ogawa, S. B. de Noyer, P. Chauvet, K. Shinohara, Y. Watanabe, H. Niizuma, T. Sasaki, and Y. Takai, "A practical approach for bus architecture optimization at transaction level," in *Proceedings of Design, Automation and Testing in Europe Conference 2003 (DATE)*. IEEE, March 2003.

[106] S. Pasricha, N. Dutt, and M. Ben-Romdhane, "Extending the transaction level modeling approach for fast communication architecture exploration," in *Proceedings of 38th Design Automation Conference (DAC)*. ACM, 2004, pp. 113–118.

[107] M. Sgroi, M. Sheets, A. Mihal, K. Keutzer, S. Malik, J. Rabaey, and A. Sangiovanni-Vincentelli, "Addressing the System-on-Chip interconnect woes through communication-based design," in *Proceedings of the 38th Design Automation Conference (DAC'01)*, June 2001, pp. 667 – 672.

[108] P. Lieverse, P. van der Wolf, and E. Deprettere, "A trace transformation technique for communication refinement," in *Proceedings of 9th International Symposium on Hardware/Software Codesign (CODES)*. ACMS, April 2001, pp. 134–139.

[109] S. Schneider, U. Mueller, and D. Tiegelbekkers, "A reactive workload generation framework for simulation-based performance engineering of system interconnects," in *Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, September 2005.

[110] Synopsys, "OpenVERA Technology Backgrounder," Synopsys, 2001, http://www.open-vera.com/.

[111] J. Hu and R. Marculescu, "Exploiting the routing flexibility for energy/performance aware mapping of regular noc architectures," in *Proceedings of the conference on Design, Automation and Test in Europe (DATE)*. Washington, DC, USA: IEEE Computer Society, 2003, p. 10688.

[112] J.-M. Daveau, T. B. Ismail, and A. A. Jerraya, "Synthesis of system-level communication by an allocation-based approach," in *ISSS '95: Proceedings of the 8th international symposium on System synthesis*. New York, NY, USA: ACM, 1995, pp. 150–155.

[113] S. Murali and G. De Micheli, "An application-specific design methodology for STBus crossbar generation," in *Proceedings of Design, Automation and Test in Europe Confer ence and Exhibition (DATE'05)*, Munchen, Germany, March 2005.

[114] J. Hu, Y. Deng, and R. Marculescu, "System-level point-to-point communication synthesis using floorplanning information," in *Proceedings of the 2002 conference on Asia South Pacific design automation/VLSI Design (ASP-DAC '02)*. Washington, DC, USA: IEEE Computer Society, 2002, p. 573.

[115] K. Lahiri, A. Raghunathan, and S. Dey, "Design space exploration for optimizing on-chip communication architecture," *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems (T-CAD)*, vol. 23, no. 6, pp. 952–961, June 2004.

[116] S. Murali and G. D. Micheli, "Sunmap: a tool for automatic topology selection and generation for nocs," in *DAC '04: Proceedings of the 41st annual conference on Design automation*. New York, NY, USA: ACM Press, 2004, pp. 914–919.

[117] A. Hansson, K. Goossens, and A. Rǎdulescu, "A unified approach to constrained mapping and routing on network-on-chip architectures," in *Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis (CODES+ISSS '05)*, 2005, pp. 75–80.

[118] S. Murali, L. Benini, and G. D. Micheli, "Mapping and physical planning of networks-on-chip architectures with quality-of-service guarantees," in *Proceedings of the 2005 conference on Asia South Pacific design automation (ASP-DAC '05)*. New York, NY, USA: ACM Press, 2005, pp. 27–32.

[119] S. Murali, M. Coenen, A. Radulescu, K. Goossens, and G. D. Micheli, "A methodology for mapping multiple use-cases onto networks on chips," in *DATE '06: Proceedings of the conference on Design, automation and test in Europe*. 3001 Leuven, Belgium, Belgium: European Design and Automation Association (EDAA), 2006, pp. 118–123.

[120] R. Ravi, M. V. Marathe, S. S. Ravi, D. J. Rosenkrantz, and H. B. H. III, "Approximation algorithms for degree-constrained minimum-cost network-design problems," *Algorithmica*, vol. 31, no. 1, pp. 58–78, 2001.

[121] A. Pinto, L. Carloni, and A. Sangiovanni-Vincentelly, "Efficient synthesis of networks on chip," in *Proceedings of 21st International Conference in Computer Design*, October 2003, pp. 146 – 150.

[122] W. H. Ho and T. M. Pinkston, "A Methodology for Designing Efficient On-Chip Interconnects on Well-Behaved Communication Patterns," in *The Ninth International Symposium on High-Performance Computer Architecture (HPCA'03)*, Feb. 2003, p. 377.

[123] K. Srinivasan, K. S. Chatha, and G. Konjevod, "An automated technique for topology and route generation of application specific on-chip interconnection networks," in *Proceedings of the 2005 IEEE/ACM International conference on Computer-aided design (ICCAD '05)*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 231–237.

[124] T. Ahonen, D. A. Siguenza-Tortosa, H. Bin, and J. Nurmi, "Topology optimization for application-specific networks-on-chip," in *Proceedings of the 2004 international workshop on System level interconnect prediction (SLIP '04)*. New York, NY, USA: ACM Press, 2004, pp. 53–60.

[125] C. J. Glass and L. M. Ni, "The turn model for adaptive routing," *SIGARCH Comput. Archit. News*, vol. 20, no. 2, pp. 278–287, 1992.

[126] G.-M. Chiu, "The odd-even turn model for adaptive routing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 11, no. 7, pp. 729–738, 2000.

[127] J. Duato, "A new theory of deadlock-free adaptive routing in wormhole networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 4, no. 12, pp. 1320–1331, 1993.

[128] W. Dally and H. Aoki, "Deadlock-free adaptive routing in multicomputer networks using virtual channels," *IEEE Transactions on Parallel and Distributed Systems*, vol. 04, no. 4, pp. 466–475, 1993.

[129] D. Starobinski, M. Karpovsky, and L. A. Zakrevski, "Application of network calculus to general topologies using turn-prohibition," *IEEE/ACM Transactions on Networking*, vol. 11, no. 3, pp. 411–421, June 2003.

[130] Y. H. Song and T. M. Pinkston, "A progressive approach to handling message-dependent deadlock in parallel computer systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 14, no. 3, pp. 259–275, 2003.

[131] Y. Choi, "Deadlock recovery based router architectures for high performance networks," Ph.D. dissertation, University of Southern California, June 2001.

[132] A. Hansson, K. Goossens, and A. Radulescu, "UMARS: A unified approach to mapping and routing on a combined guaranteed service and best-effort network-on-chip architecture," Philips Research Technical Report 2005/00340, Tech. Rep., April 2005.

[133] S. Murali, P. Meloni, F. Angiolini, D. Atienza, S. Carta, L. Benini, and G. D. Micheli, "Designing application-specific networks on chips with floorplan information," in *Proceedings of the 2006 International Conference on Computer-Aided Design (ICCAD)*. New York, NY, USA: ACM Press, 2006, pp. 355–362.

[134] S. L. Scott and G. Thorson, "Optimized routing in the cray t3d," in *PCRCW '94: Proceedings of the First International Workshop on Parallel Computer Routing and Communication*. London, UK: Springer-Verlag, 1994, pp. 281–294.

[135] S. Scott and G. Thorson, "The Cray T3E network: Adaptive routing in a high performance 3D torus," in *Proceedings of Hot Interconnects IV*, Stanford, CA, 1996.

[136] J. Carbonaro and F. Verhoorn, "Cavallino: the teraflops router and NIC," in *Proceedings of Hot Interconnects IV*, Stanford, CA, 1996.

[137] S. S. Mukherjee, P. Bannon, S. Lang, A. Spink, and D. Webb, "The alpha 21364 network architecture," *IEEE Micro*, vol. 22, no. 1, pp. 26–35, 2002.

[138] J. Laudon and D. Lenoski, "The sgi origin: a ccnuma highly scalable server," *SIGARCH Comput. Archit. News*, vol. 25, no. 2, pp. 241–251, 1997.

[139] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy, "The directory-based cache coherence protocol for the dash multiprocessor," in *ISCA '90: Proceedings of the 17th annual international symposium on Computer Architecture*. New York, NY, USA: ACM, 1990, pp. 148–159.

[140] B. Gebremichael, F. W. Vaandrager, M. Zhang, K. Goossens, E. Rijpkema, and A. Radulescu, "Deadlock prevention in the æthereal protocol," in *CHARME*, 2005, pp. 345–348.

[141] V. Agarwal, S. Keckler, and D. Burger, "The effect of technology scaling on microarchitectural structures," Technical Report TR2000-02, University of Texas at Austin, USA, Tech. Rep., 2002.

[142] P. Shivakumar, S. W. Keckler, C. R. Moore, and D. Burger, "Exploiting microarchitectural redundancy for defect tolerance," in *ICCD '03: Proceedings of the 21st International Conference on Computer Design*. Washington, DC, USA: IEEE Computer Society, 2003, p. 481.

[143] P. Stanley-Marbell and D. Marculescu, "Dynamic fault-tolerance management in failure-prone battery-powered systems," in *Proceedings of 12th International Workshop on Logic and Synthesis (IWLS 2003)*, Laguna Beach, CA., May 2003, pp. 370–381.

[144] T. M. Austin, "DIVA: A reliable substrate for deep submicron microarchitecture design," in *International Symposium on Microarchitecture*, 1999, pp. 196–207.

[145] T. Austin, V. Bertacco, D. Blaauw, and T. Mudge, "Opportunities and challenges for better than worst-case design," in *Asia-South Pacific Design Automation Conference*, January 2005.

[146] R. Aitken, N. Dogra, D. Gandhi, and S. Becker, "Redundancy, repair, and test features of a 90nm embedded SRAM generator," in *Proceedings of the 18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, 2003.

[147] Y. Zorian, E. J. Marinissen, and S. Dey, "Sram design on 65-nm cmos technology with dynamic sleep transistor for leakage reduction," *IEEE Computer*, vol. 32, no. 6, pp. 52–60, June 1999.

[148] M. Blaum, R. Goodman, and R. McEliece, "The reliability of single-error protected computer memories," *IEEE Trans. Comput.*, vol. 37, no. 1, pp. 114–119, 1988.

[149] N. S. Bowen and D. K. Pradhan, "The effect of program behavior on fault observability," *IEEE Trans. Comput.*, vol. 45, no. 8, pp. 868–880, 1996.

[150] J. Ohtani, T. Ooishi, T. Kawagoe, M. Niiro, M. Maruta, and H. Hidaka, "A shared built-in selfrepair analysis for multiple embedded memories," in *Proc. IEEE Custom Integrated Circuits Conf. (CICC)*, vol. 4, 2001, pp. 187–190.

[151] C.-T. Huang, C.-F. Wu, J.-F. Li, and C.-W. Wu, "Built-in redundancy analysis for memory yield improvement," *IEEE Transactions on Reliability*, vol. 52, no. 4, pp. 386–399, 2003.

[152] M. Choi, N. Park, F. Lombardi, Y. Kim, and V. Piuri, "Optimal spare utilization in repairable and reliable memory cores," in *Records of the 2003 International Workshop on Memory Technology, Design and Testing*, 2003.

[153] A. Morgenshtein, E. Bolotin, I. Cidon, R. Ginosar, and A. Kolodny, "Micro-modem - reliability solution for NoC communications," in *Proc. of the 11th IEEE Intl. Conf. on Electronics, Circuits and Systems (ICECS)*, Dec. 2004, pp. 483–486.

[154] D. Bertozzi and L. Benini, "Xpipes: A network-on-chip architecture for gigascale systems-on-chip," *IEEE Circuits and Systems Magazine*, vol. 4, no. 2, pp. 18–31, 2004.

[155] C.-L. Su, R.-F. Huang, and C.-W. Wu, "A processor-based built-in self-repair design for embedded memories," in *Proceedings of the 12th Asian Test Symposium (ATS)*, 2003.

[156] B. Rajendran, R. S. Shenoy, D. J. Witte, N. S. Chokshi, R. L. DeLeon, and G. S. Tompa, "Cmos transistor processing compatible with monolithic 3-d integration," in *Proc. VLSI Interconnection (VMIC)*, 2005, pp. 76–82.

[157] Ziptronix, *Ziptronix target vertical scalability*, 2005.

[158] S. Christiansen, R. Singh, and U. Gosele, "Wafer direct bonding: From advanced substrate engineering to future applications in micro/nanoelectronics," in *Proceedings of the IEEE*, December 2006, pp. 2060–2106.

[159] K. Lee, "Wafer-stacked package technology for high-performance system," in *RTI Int. technology Venture Forum*, 2005.

[160] S.Spiesshoefer and et al, "Z-axis interconnects using fine pitch, nanoscale through-silicon vias: Process development," in *Electronic Components and Technology Conference*, 2004.

[161] R. S. Patti, "Three-dimensional integrated circuits and the future of system-on-chip designs," *Proceedings of the IEEE*, vol. 94, no. 6, June 2006.

[162] A. W. Topol, J. D. C. La Tulipe, L. Shi, D. J. Frank, K. Bernstein, S. E. Steen, A. Kumar, G. U. Singco, A. M. Young, K. W. Guarini, and M. Ieong, "Three-dimensional integrated circuits," *IBM Journal of Research and Development*, vol. 50, no. 4/5, pp. 491–506, July/September 2006.

[163] S. Fujita, K. Nomura, K. Abe, and T. Lee, "3d on-chip networking technology based on post-silicon devices for future networks-on-chip," in *Nano-Networks and Workshops*, September 2006, pp. 1–5.

[164] V. F. Pavlidis and E. G. Friedman, "3-D topologies for networks-on-chip," in *Proceedings of the IEEE SoC Conference (SOCC)*. IEEE Computer Society, 2006, pp. 285–288.

[165] B. Feero and P. P. Pande, "Performance evaluation for three-dimensional networks-on-chip," in *Proceedings of the IEEE Annual Symposium on VLSI (ISVLSI)*. IEEE Computer Society, 2007, pp. 305–310.

[166] J. Kim, C. Nicopoulos, D. Park, R. Das, Y. Xie, N. Vijaykrishnan, M. S. Yousif, and C. R. Das, "A novel dimensionally-decomposed router for on-chip communication in 3d architectures," in *Proceedings of the 34th International Symposium on Computer Architecture (ISCA)*, 2007.

[167] C. A. Mineo, "Clock tree insertion and verification for 3D integrated circuits," North Carolina State University at Raleigh, Tech. Rep., 2005.

[168] M. Mondal, A. J. Ricketts, S. Kirolos, T. Ragheb, G. Link, N. Vijaykrishnan, and Y. Massoud, "Thermally robust clocking schemes for 3D integrated circuits," in *Proceedings of the 2007 Design, Automation and Test in Europe conference (DATE)*. New York, NY, USA: ACM Press, 2007, pp. 1206–1211.

[169] J. Bainbridge and S. Furber, "Chain: a delay-insensitive chip area interconnect," *IEEE Micro*, vol. 22, no. 5, pp. 16–23, Sep/Oct 2002.

[170] T. Bjerregaard and J. Sparsø, "Scheduling discipline for latency and bandwidth guarantees in asynchronous network-on-chip," in *Proceedings of the 11th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, 2005, pp. 34–43.

[171] A. Sheibanyrad, I. M. Panades, and A. Greiner, "Systematic comparison between the asynchronous and the multi-synchronous implementations of a network on chip architecture," in *DATE '07: Proceedings of the conference on Design, automation and test in Europe*. San Jose, CA, USA: EDA Consortium, 2007, pp. 1090–1095.

[172] A. Lines, "Asynchronous interconnect for synchronous soc design," *IEEE Micro*, vol. 24, no. 1, pp. 32–41, Jan-Feb 2004.

[173] E. Beigne, F. Clermidy, P. V. A. Clouard, and M. Renaudin, "An asynchronous noc architecture providing low latency service and its multi-level design framework," in *11th IEEE International Symposium on Asynchronous Circuits and Systems*, 2005, pp. 54–63.

[174] E. Nigussie, J. Plosila, and J. Isoaho, "Delay-insensitive on-chip communication link using low-swing simultaneous bidirectional signaling," in *Proceedings of the 2006 Emerging VLSI Technologies and Architectures (ISVLSI06)*, 2006, p. 217.

[175] E. Grass, F. Winkler, M. Krstic, A. Julius, C. Stahl, and M. Piz, "Enhanced gals techniques for datapath applications," in *PATMOS*, 2005, pp. 581–590.

[176] M. B. Stensgaard, T. Bjerregaard, J. Sparsø, and J. H. Pedersen, "A simple clockless network-on-chip for a commercial audio DSP chip," in *Proceedings of the 9th EUROMICRO Conference on Digital System Design*, 2006, pp. 641–648.

[177] L. A. Plana, W. J. Bainbridge, and S. B. Furber, "The design and test of a smartcard chip using a CHAIN self-timed network-on-chip," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, 2004, p. 274.

[178] M. Krstic, E. Grass, C. Stahl, and M. Piz, "System integration by request-driven gals design," *IEE Proceedings on Computers and Digital Techniques*, vol. 153, no. 5, pp. 362–372, September 2006.

[179] D. Lattard and et al, "A telecom baseband circuit-based on an asynchronous network-on-chip," in *Proc. of the International Solid State Circuits Conference, ISSCC2007*, 2007, pp. 258–259.

[180] K. Lee, S.-J. Lee, and H.-J. Yoo, "Low-power network-on-chip for high-performance soc design," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 2, pp. 148–160, February 2006.

[181] S.-J. Lee, "Cost-optimization and chip implementation of on-chip network," KAIST, Tech. Rep., 2005.

[182] M. Ghoneima, Y. Ismail, M. Khellah, and V. De, "Variation-tolerant and low-power source-synchronous multicycle on-chip interconnection scheme," *VLSI Design*, vol. 2007, 2007.

[183] D. Mangano, R. Locatelli, A. Scandurra, C. Pistritto, M. Coppola, L. Fanucci, F. Vitullo, and D. Zandri, "Skew insensitive physical links for network on chip," in *1st International Conference on Nano-Networks (NanoNet)*, 2006, pp. 1–5.

[184] Synopsys, "Cocentric," http://www.synopsys.com.

[185] "SystemC Community," 2003, http://www.systemc.org.

[186] "Amba buses overview," ARM Ltd., 2005, http://www.arm.com/products/solutions/AMBAOverview.html.

[187] "The uClinux embedded linux/microcontroller project," http://www.uclinux.org/.

[188] On-Line Application Research (OAR), "RTEMS, open-source real-time operating system for multiprocessor systems," 2002, http://www.rtems.org.

[189] O. Schliebusch, A. Chattopadhyay, R. Leupers, G. Ascheid, H. Meyr, M. Steinert, G. Braun, and A. Nohl, "RTL processor synthesis for architecture exploration and implementation," in *Proceedings of the 2004 Design, Automation and Test in Europe Conference (DATE'04)*. IEEE, 2004, pp. 156–160.

[190] A. Hoffmann, H. Meyr, and R. Leupers, *Architecture Exploration for Embedded Processors with Lisa*. Norwell, MA, USA: Kluwer Academic Publishers, 2002.

[191] A. Wieferink, T. Kogel, R. Leupers, G. Ascheid, H. Meyr, G. Braun, and A. Nohl, "A system level processor/communication co-exploration methodology for multi-processor system-on-chip platforms," in *Proceedings of the 2004 Design, Automation and Test in Europe Conference (DATE'04)*. IEEE, 2004, pp. 1256–1263.

[192] F. Angiolini, F. Menichelli, A. Ferrero, L. Benini, and M. Olivieri, "A post-compiler approach to scratchpad mapping of code," in *Proceedings of the 2004 ACM International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, 2004, pp. 259–267.

[193] K. Lahiri, A. Raghunathan, and S. Dey, "Evaluation of the traffic-performance characteristics of System-on-Chip communication architectures," in *Proceedings of the 14th International Conference on VLSI Design*, 2001, pp. 29–35.

[194] F. Fummi, S. Martini, G. Perbellini, M. Poncino, F. Ricciato, and M. Turolla, "Heterogeneous co-simulation of networked embedded systems," in *Proceedings of Design, Automation and Testing in Europe Conference 2004 (DATE)*. IEEE, Febuary 2004.

[195] S. Mahadevan, F. Angiolini, M. Storgaard, R. G. Olsen, J. Sparsø, and J. Madsen, "A network traffic generator model for fast network-on-chip simulation," in *Proceedings of Design, Automation and Testing in Europe Conference 2005 (DATE)*. IEEE, March 2005, pp. 780–785.

[196] F. Angiolini, S. Mahadevan, J. Madsen, L. Benini, and J. Sparsø, "Realistically rendering SoC traffic patterns with interrupt awareness," in *IFIP International Conference on Very Large Scale Integration (VLSI-SoC)*, September 2005.

[197] G. V. Varatkar and R. Marculescu, "On-chip traffic modeling and synthesis for MPEG-2 video applications," *Transactions on Very Larget Scale Integration (VLSI) Systems*, vol. 12, no. 1, pp. 108–119, JANUARY 2004.

[198] W. Wolf, *Computers as Components:Principles of Embedded Computing System Design*. Morgan Kaufmann, 2001.

[199] N. Genko, D. Atienza, G. D. Micheli, L. Benini, J. M. Mendias, R. Hermida, and F. Catthoor, "A novel approach for network on chip emulation," in *International Symposium on Circuits and Systems*. IEEE, 2005, pp. 2365–2368.

[200] S. Kuenzli, F. Poletti, L. Benini, and L. Thiele, "Combining simulation and formal methods for system-level performance analysis," in *Proceedings of Design, Automation and Testing in Europe Conference 2006 (DATE)*. IEEE, March 2006, pp. 236–242.

[201] S. Pasricha, N. Dutt, and M. Ben-Romdhane, "Extending the transaction level modeling approach for fast communication architecture exploration," in *Proceedings of 38th Design Automation Conference (DAC)*. ACM, 2004, pp. 113–118.

[202] G. Palermo, C. Silvano, G. Mariani, R. Locatelli, and M. Coppola, "Application-specific topology design customization for stnoc," in *DSD '07: Proceedings of the 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 547–550.

[203] D. Siguenza-Tortosa and J. Nurmi, "Vhdl-based simulation environment for proteo noc," in *Proceedings of High-Level Design Validation and Test Workshop*, October 2002, pp. 1 – 6.

[204] A. Pullini, F. Angiolini, D. Bertozzi, and L. Benini, "Fault tolerance overhead in network-on-chip flow control schemes," in *Proceedings of the 18th Annual Symposium on Integrated Circuits and System Design (SBCCI)*, 2005, pp. 224–229.

[205] L. Carloni, K. McMillan, and A. Sangiovanni Vincentelli, "Theory of latency-insensitive design," *IEEE Transactions on CAD*, vol. 20, no. 9, 2001.

[206] K. Banerjee and A. Mehrotra, "A power-optimal repeater insertion methodology for global interconnects in nanometer designs," *IEEE Transactions on Electron Devices*, vol. 49, pp. 2001–2007, November 2002.

[207] V. Chandra, A. Xu, H. Schmit, and L. Pileggi, "An interconnect channel design methodology for high performance integrated circuits," in *DATE '04: Proceedings of the conference on Design, automation and test in Europe*. Washington, DC, USA: IEEE Computer Society, 2004, p. 21138.

[208] J. Hu and R. Marculescu, "Energy-aware communication and task scheduling for network-on-chip architectures under real-time constraints," in *DATE '04: Proceedings of the conference on Design, automation and test in Europe*. Washington, DC, USA: IEEE Computer Society, 2004, p. 10234.

[209] K. Goossens, J. Dielissen, O. P. Gangwal, S. G. Pestana, A. Radulescu, and E. Rijpkema, "A design flow for application-specific networks

on chip with guaranteed performance to accelerate soc design and verification," in *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 1182–1187.

[210] S. S. Mukherjee, P. Bannon, S. Lang, A. Spink, and D. Webb, "The alpha 21364 network architecture," in *HOTI '01: Proceedings of the The Ninth Symposium on High Performance Interconnects (HOTI '01)*. Washington, DC, USA: IEEE Computer Society, 2001, p. 113.

[211] L. Widdoes and S. Correll, "The S-1 project: Developing high performance computers," in *IEEE Spring Compcon*, Feb 1980, pp. 282–291.

[212] B. Hendrickson and R. Leland, "The chaco user's guide: Version 2.0," Sandia Tech Report SAND94–2692, Tech. Rep., 1994, http://www.cs.sandia.gov/~bahendr/chaco.html.

[213] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. London, England: MIT Press - McGraw-Hill Book Company, 1994.

[214] I. M. Saurabh Adya, "Fixed-outline floorplanning : Enabling hierarchical design," *IEEE Trans. on VLSI*, vol. 11, no. 6, pp. 1120–1135, December 2003.

[215] A. Jalabert, S. Murali, L. Benini, and G. De Micheli, "×pipescompiler: A tool for instantiating application specific networks on chip," in *Proceedings of Design, Automation and Test in Europe Conference and Exhibition (DATE'04)*, vol. 4. IEEE, February 2004, pp. 1 – 6.

[216] D. Bertozzi, A. Jalabert, S. Murali, R. Tamhankar, S. Stergiou, L. Benini, and G. D. Micheli, "Noc synthesis flow for customized domain specific multiprocessor systems-on-chip." *IEEE Trans. Parallel Distrib. Syst.*, vol. 16, no. 2, pp. 113–129, 2005.

[217] Cadence, "SoC Encounter," http://www.cadence.com.

[218] Synopsys, "Astro," http://www.synopsys.com.

[219] ——, "PrimeTime," http://www.synopsys.com.

[220] F. Angiolini, P. Meloni, S. Carta, L. Benini, and L. Raffo, "Contrasting a NoC and a traditional interconnect fabric with layout awareness," in *Proceedings of the DATE '06 Conference*, Munich, Germany, 2006.

[221] Synopsys, "Physical Compiler," http://www.synopsys.com.

[222] ——, "coreTools," http://www.synopsys.com.

[223] ——, "Design Compiler," http://www.synopsys.com.

[224] ——, "PrimePower," http://www.synopsys.com.

[225] S. Stergiou, F. Angiolini, S. Carta, L. Raffo, D. Bertozzi, and G. D. Micheli, "×pipes lite: A synthesis oriented design library for networks on chips," in *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 1188–1193.

[226] G. E. Box, J. S. Hunter, and W. G. Hunter, *Statistics for Experimenters: Design, Innovation, and Discovery*. Wiley, 2005.

[227] M. Graphics, "ModelSim," http://www.model.com.

[228] H. Wang, M. Miranda, W. Dehaene, F. Catthoor, and K. Maex, "Systematic analysis of energy and delay impact of very deep submicron process variability effects in embedded sram modules," in *Proceedings of DATE*, 2005, pp. 914–919.

[229] S. Xiao, X. Shi, G. L. Feng, and T. R. N. Rao, "A generalization of the single b-bit byte error correcting and double bit error detecting codes for high-speed memory systems," *IEEE Transactions on Computer*, vol. 45, no. 4, pp. 508–511, 1996.

[230] I. S. et al., "Scalable wavelet coding for synthetic and natural hybrid images," *IEEE Transactions on Circuits and Systems For Video Technologhy*, vol. 9, no. 2, March 1999.

[231] M. Pollefeys, R. Koch, M. Vergauwen, and L. Van Gool, "Metric 3D surface reconstruction from uncalibrated image sequences," in *Lecture Notes in Computer Science*, vol. 1506, Proceedings SMILE Workshop (post-ECCV'98). Springer-Verlag, 1998, pp. 139 – 153.

[232] "Target jr," 2002, http://computing.ee.ethz.ch/sepp.

[233] J. Kim, C. Nicopoulos, D. Park, R. Das, Y. Xie, N. Vijaykrishnan, M. Yousif, and C. Das, "A novel dimensionally-decomposed router for on-chip communication in 3d architectures," in *Proceedings of ISCA*, June 2007.

[234] I. Loi, F. Angiolini, and L. Benini, "Supporting vertical links for 3d networks-on-chip: Toward an automated design and analysis flow," in *Proceedings of the Nano-Net Conference 2007*, 2007.

[235] Ansoft Corp., "Q3d extractor," 2007, http://www.ansoft.com.

[236] K. N. Chen, A. Fan, and C. S. T. ans R. Reif, "Contact resistance measurament of bonded copper interconnects for three-dimensional integration technology," *IEEE ELECTRON DEVICE LETTERS*, vol. 25, no. 1, January 2005.

[237] T. Bjerregaard and J. Sparsø, "A scheduling discipline for latency and bandwidth guarantees in asynchronous network-on-chip," in *ASYNC '05: Proceedings of the 11th IEEE International Symposium on Asynchronous Circuits and Systems*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 34–43.

[238] F. Li, C. Nicopoulos, T. Richardson, Y. Xie, V. Narayanan, and M. Kandemir, "Design and management of 3d chip multiprocessors using network-in-memory," in *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 130–141.

[239] T. Kgil, S. D'Souza, A. G. Saidi, N. L. Binkert, R. G. Dreslinski, T. N. Mudge, S. K. Reinhardt, and K. Flautner, "Picoserver: using 3d stacking technology to enable a compact energy efficient chip multiprocessor," in *ASPLOS*, 2006, pp. 117–128.

[240] G. M. Link and N. Vijaykrishnan, "Hotspot prevention through runtime reconfiguration in network-on-chip," in *Proceedings of DATE*, vol. 01. Los Alamitos, CA, USA: IEEE Computer Society, 2005, pp. 648–649.

[241] T. Marescaux, J.-I. Mignolet, A. Bartic, W. Moffat, D. Verkest, S. Vernalde, and R. Lauwereins, "Networks on chip as hardware components of an os for reconfigurable systems," in *In Proceedings of Field-Programmable Logic and Applications (FPL'03)*, September 2003.